SOC DESIGN INTEGRATION BY USING AUTOMATIC INTERCONNECTION RECTIFICATION

Chun-Yao Wang, Shing-Wu Tung and Jing-Yang Jou

Department of Electronics Engineering
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.

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.

1. 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] $\sim$ [10] and those based on simulation techniques [5] [11] $\sim$ [12]. 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) [13] representation. Thus, to verify the interconnection among the IP cores with all description levels (soft, firm, and hard cores) embedded 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.

2. 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 fault free port sequence (**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 faulty port sequences (**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.

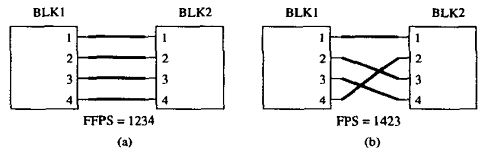


Figure 1: The schematic representation of the FPS

The PUPs representation is a metric used in the AIR to indicate the remaining possible uncorrected ports (**PUPs**) currently in the integrated design. We use Example 2 to demonstrate the PUPs representation.

Example 2: Given an 8-input core, the inputs are numbered from 1 to 8. These port numbers are all possible uncorrected ports and are placed in a pair of parentheses. If these ports are faulty indeed, they must be misplaced with the other ports in the same group only. The group with only one port number represents the port is correct. For example, the (12345678) represents that the port 1 $\sim$ port 8 are all possible uncorrected ports and they could be misplaced with each other, the |PUPs| = 8. The (1)(234)(567)(8) represents that the port 1 and port 8 are correct ports, the port 2 $\sim$ port 4 and the port 5 $\sim$ port 7 are the possible uncorrected ports and they could be misplaced with the other ports in the same group only, the |PUPs| = 6. If the PUPs are induced to (1)(2)(3)(4)(5)(6)(7)(8), the |PUPs| = 0, and the interconnection is correct.

Definition 3 defines the cross-group operation, which is used to calculate the updated PUPs.

Definition 3: Given two PUPs representation P1 and P2 with the same number of ports. The cross-group P3 of P1 and P2 is denoted as P1 $\triangle$ P2 and it satisfies with the following condition: if any two port numbers x and y are both placed in the same group of P1 and P2, they will be placed in the same group of P3; otherwise, they are placed in the different groups of P3.

Example 3: Given two PUPs, P1 = (12)(34) and P2 = (14)(23), the cross-group P3 of P1 and P2 = P1 $\triangle$ P2 = (1)(2)(3)(4).

We also exploit Example 3 to explain the physical meaning of the cross-group operation. P1 represents that if the port 2 and port 3 in a FPS are faulty, they **must not** be misplaced with each other. On the contrary, P2 represent that if the port 2 and port 3 in the same FPS are faulty, they **must** be misplaced with each other. Since the port 2 and port 3 in this FPS cannot be "misplaced" with each other and "not misplaced" with each other simultaneously, the port 2 and port 3 have not to be faulty, and they have to be placed in single groups in the PUPs, respectively. The proposed cross-group operation accomplishes this object indeed.

The environment and mechanism of POF verification, which exploits the IEEE P1500 SECT [15], can be found in [6]. Thus, we omit describing them here.

3. THE AIR ALGORITHM

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 correction, and PUPs calculation as shown in Fig. 2. The pattern generation stage generates valid patterns Si to differentiate the outputs of fault free interconnection and faulty interconnection. The fault detection stage applies Si into the integrated design to examine whether the interconnection are misplaced or not. If the fault effect appears after the fault detection stage, the misplaced interconnection are identified and rectified in the fault diagnosis and correction stage. Otherwise, the PUPs calculation stage is performed. After the fault diagnosis and correction stage, the same PUPs calculation stage is performed as well. The PUPs calculation stage can figure out the possible faulty ports remained in the integrated design. These remaining possible faulty ports will be verified in the subsequent iterations by additional verification patterns. Since the fault diagnosis and correction stage usually cannot correct all misplaced ports in one iteration, the rectified interconnection has to be verified (detection, diagnosis, and correction) by Si again until the fault effect disappears after the fault detection stage. These iterative procedures are presented with bold lines in the AIR flow as shown in Fig. 2.

3.1. Pattern Generation

Definition 4: The set consists of all patterns with m 1s and $(N\text{-}m)$ 0s is denoted as Θ_m^N, where $m \in [0, 1, 2, \cdots, N-1, N]$. The size of Θ_m^N is the number of patterns in Θ_m^N and is denoted as $|\Theta_m^N|$.

Example 4: For a 4-input core, Θ_0^4={0000}, $|\Theta_0^4|$ =1. Θ_1^4={1000, 0100, 0010, 0001}, $|\Theta_1^4|$ =4.

Theorem 1: Θ_m^N *can activate all (N!-1) POFs where m* $\in$ *[1, 2,* $\cdots$*, N-2, N-1].* [6]

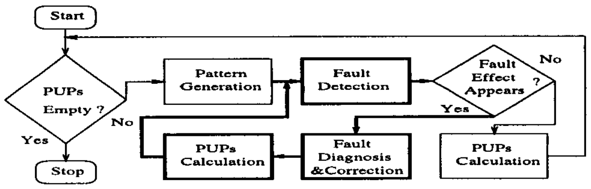
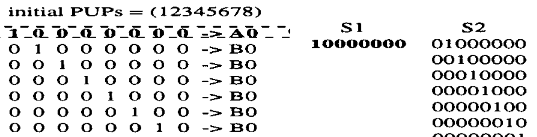


Figure 2: The AIR flow

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

We use an example to demonstrate the pattern generation stage. This example will be used to demonstrate the other stages in the AIR as well later. Given an 8-input combinational core, the initial PUPs are (12345678). Assume the simulation results of Θ_1^8 are shown in Fig. 3 and are represented in symbolic output representation. The patterns with the same output are grouped into one set. Θ_1^8 patterns can be grouped into two sets S1 and S2. Either S1 or S2 can be selected as the verification patterns. Here we select the smaller set S1 as the verification patterns [6].



```
initial PUPs = (12345678)
                                        S1          S2
 1 0 0 0 0 0 0 0 -> A0               10000000    01000000
 0 1 0 0 0 0 0 0 -> B0                           00100000
 0 0 1 0 0 0 0 0 -> B0                           00010000
 0 0 0 1 0 0 0 0 -> B0                           00001000
 0 0 0 0 1 0 0 0 -> B0                           00000100
 0 0 0 0 0 1 0 0 -> B0                           00000010
 0 0 0 0 0 0 1 0 -> B0                           00000001
 0 0 0 0 0 0 0 1 -> B0
```

Figure 3: The simulation outputs of Θ_1^8

The system integrators do not know how the cores are connected exactly in the actual integrated design. However, to demonstrate the fault detection, diagnosis, and correction procedures on the interconnection verification, we assume the FPS λ of this example is given and is 83762451.

3.2. Fault Detection

We apply S1 {10000000} into the design with the FPS λ 83762451, and find that the corresponding output of {10000000} is B0 as shown in Fig. 4(a). Since the fault free output is A0, the fault effect appears and λ is detected by S1.

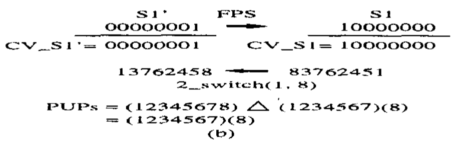
3.3. Fault Diagnosis and Correction

Definition 5: 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.

We apply S1 into the integrated design, and realize that the real applied input is not {10000000} by observing the unexpected output B0. Since there are seven patterns that produce B0 output, we do not know exactly what the actual applied pattern is. Nevertheless, **we know the actual applied pattern which produces A0 output instead**. Therefore, 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. 4(a). From Fig. 4(a), we find that when

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)

S1	Outputs	
	Fault Free	FPS 13762458
10000000	A0	A0

PUPs = (1234567)(8) △ (1)(2345678)
= (1)(234567)(8)

(c)

Figure 4: Rectification processes of Θ_1^8

the last pattern {00000001} is applied, 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}, and we calculate CV_S1'=00000001 and CV_S1=10000000. Then, the misplaced ports can be identified by comparing CV_S1' and CV_S1.

Definition 6: The exchange of the i^{th} port with the j^{th} port is denoted as 2_switch(i, j).

Comparing CV_S1' and CV_S1, we observe that the 1^{st} digit and the 8^{th} digit of CV_S1' and CV_S1 are different. We know that if the FPS is the FFPS, CV_S and CV_S' will be identical. **The FFPS will keep CV being intact for all patterns. Therefore, switching the ports with different CV values to preserve CV can move the FPS to the FFPS.** Thus, we apply 2_switch(1, 8) on CV_S1 to let CV_S1 be the same as CV_S1' as shown in Fig. 4(b). Now, the corrected FPS becomes 13762458.

We exploit this property of the FFPS to rectify the misplaced ports throughout the AIR algorithm.

The following lemmas and theorems state the convergency of fault correction procedure.

Theorem 2: *The correct ports in a FPS λ will not be rectified to faulty ones in the fault diagnosis and correction stage.*

Definition 7: 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

(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 is a sequence of CVIFs(S1'). **Theorem 3 guarantees that CVIF(S1') are the only possible faults which we have to deal with in the succeeding iterations.**

3.4. PUPs Calculation

As mentioned above, CVIF(S1') are the remaining possible faults only and they are occurred among the port 1 $\sim$ 7. Thus, the PUPs representation is (1234567)(8). This PUPs has to be cross-grouped with the initial PUPs (12345678) to obtain the updated PUPs. Thus, the updated PUPs = (12345678) △ (1234567)(8) = (1234567)(8). This result is also shown in Fig. 4(b).

Thereafter, we apply S1 into the integrated design again with the corrected FPS 13762458 to examine whether the corrected FPS 13762458 can be further corrected by S1. In Fig. 4(c), we find that the outputs of S1 with the FPS 13762458 are the same as the expected outputs A0, thus, the FPS 13762458 cannot be further corrected by S1. **Please note that this reexamining process also provide the information to reduce the PUPs representation.** The corrected FPS 13762458 does not change the outputs A0, therefore, the actual input pattern is also {10000000} (S1) and the corresponding CV_S1 is unchanged. According to Theorem 3, the remaining possible faults are CVIFs(S1) and they can be expressed as (1)(2345678). Thus, the updated PUPs become (1)(234567)(8), which are obtained from (1234567)(8) △ (1)(2345678). At this time, we move to the next iteration to generate further patterns to detect and rectify the remaining faulty ports among port 2 $\sim$ 7.

3.5. Summary

The FPS is corrected from 83762451 to 13762458, and the PUPs representation is reduced from (12345678) to (1)(234567)(8). These results illustrate that the PUPs 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.

3.6. 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.

4. EXPERIMENTAL RESULTS

The heuristic AIR, which adds the iteration counter to bound the processing time, has been integrated into the SIS [14] environment. Experiments are conducted over a set of ISCAS-85, 89, and ITC-99 benchmarks. These benchmarks are in BLIF format which is a netlist level design description. However, we only use the simulation information to conduct the experiments and therefore, arbitrary level of design description can be used for conducting POF

	parameters		blind connection		
bench	|PI|	lits.	a/b	|PUPs|	time(s)
c17	5	12	4/4	0	0.1
c880	60	703	58/58	0	105
c1355	41	1032	40/40	0	152
c1908	33	1497	32/32	0	149
c432	36	372	36/36	0	41
c499	41	616	39/39	0	57
c3540	50	2934	49/49	0	636
c5315	178	4369	178/178	0	10270
c2670	233	2043	231/231	0	9373
c7552	207	6098	207/207	0	23972
c6288	32	4800	32/32	0	425

Table 1: Experimental results of the heuristic AIR on ISCAS-85

verification. The simulation information of the BLIF benchmarks imitate the simulation model of IP cores. The functionalities of these benchmarks include ALU (c5315), multiplier (c6288), processors (b14, b15), and some ASIC designs, thus, the experiments can represent the realistic SoC design appropriately to some degree.

Table 1 summaries the experimental results of the heuristic AIR on ISCAS-85. The |PI| represents the number of inputs. The number of literals (lits.) indicates the scale of a benchmark. The a/b presents "number of corrected ports/number of faulty ports". These faulty ports in the experiments are caused by the blind connection. The blind connection represents the worst case of the SoC integration. To imitate the actual interconnection faults in the integration, the FPS is generated as follows. For each port i, i from 1 to N, we assign a random number ($\in [1 \sim N]$) to it. If the number has been assigned to port j, where $1 \leq j < i$, we generate another one to the port i until it is not repeated. This process is similar to the real interconnection process with blindness. Since the FPSs in the experiments are generated randomly, the generated FPSs quantify the inject out of order permutations.

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 PUPs representation becomes empty. At the end of AIR, the number of corrected ports, |PUPs|, and CPU time are returned. The number of corrected ports is obtained by comparing the final FPS with the FFPS. The |PUPs| is obtained from the final PUPs representation. The CPU time is measured in second on an Ultra Sparc II workstation.

Note that since we greatly concern about how many faulty ports are injected and corrected rather than the number of verification patterns [6] [7] in the experiments, we do not report the number of the verification patterns in the experimental results.

According to Table 1 and 2, the faulty ports of each benchmark **are all corrected**, the |PUPs| of each benchmark is 0 as well, 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.

5. CONCLUSIONS

In the SoC era, the embedded cores are mixed and integrated to create a system chip. System designers integrate those cores manually and have the possibility of incorrect integration due to the misplaced I/O ports. Furthermore, without the knowledge of the internal structures of the embedded cores, system designers have difficult time to locate the position of having erroneous interconnection. The AIR technique provides an efficient solution to integrate the cores with correct interconnection automatically. Therefore this algorithm can reduce the time on design verification in core-based design methodology.

	parameters			blind connection		
bench	|PI|	lits.	FFs	a/b	|PUPs|	time(s)
s1196	14	1009	18	13/13	0	37.2
s1238	14	1041	18	12/12	0	37.6
s1488	8	1387	6	7/7	0	17.1
s1494	8	1393	6	8/8	0	18.4
s15850	14	13659	597	14/14	0	608
s27	4	18	3	3/3	0	0.1
s5378	35	4212	164	34/34	0	680
s641	35	539	19	35/35	0	136
s713	19	591	19	17/17	0	127
s820	18	757	5	17/17	0	137
s832	18	767	5	16/16	0	147.6
s9234	36	7971	211	34/34	0	1961
s444	3	352	21	3/3	0	3.4
s510	19	424	6	19/19	0	38.5
s344	9	269	15	8/8	0	4.9
s349	9	273	15	9/9	0	5.5
s382	3	306	21	3/3	0	9.2
s386	7	347	6	7/7	0	3.8
s400	3	320	21	3/3	0	4.2
s13207	31	11165	669	29/29	0	2338
s1423	17	1164	74	17/17	0	67.4
s6669	83	5343	239	82/82	0	6985
s4863	49	4092	104	49/49	0	1391
s1269	18	1047	37	17/17	0	66.7
s1512	29	1264	57	28/28	0	218.2
s3271	26	2697	116	25/25	0	403
s3330	40	2816	132	40/40	0	820
s3384	43	2755	183	41/41	0	1327
b10	11	331	17	11/11	0	8.4
b11	7	1078	31	7/7	0	52.6
b12	5	1887	121	5/5	0	19.3
b13	10	507	53	9/9	0	15.8
b14	32	11849	245	31/31	0	2634
b15	37	15856	449	37/37	0	5062

Table 2: Experimental results of the heuristic AIR on sequential benchmarks

6. REFERENCES

[1] H. Chang, et al., "Surviving the SoC revolution - a guide to platform-based design," Kluwer Academic Publishers, 1999.

[2] J. A. Rowson, et al., "Interface-based design," *in Proc. Design Automation Conference*, pp.178-183, Jun. 1997.

[3] J.-Y. Jou, et al., "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] J.-Y. Jou, et al., "On automatic-verification pattern generation for SoC with port-order fault model," *IEEE Transactions on Computer-Aided Design*, pp.466-479, vol.21, no.4, Apr. 2002.

[7] J.-Y. Jou, et al., "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] P.-Y. Chung, Y.-M. Wang, and I. N. Hajj, "Logic design error diagnosis and correction," *IEEE Transaction on VLSI Syst.*, vol.2, pp.320-332, Sep. 1994.

[11] 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.

[12] 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.

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

[14] E. M. Sentovich, et al., "Sequential circuit design using synthesis and optimization," *in Proc. IEEE International Conference on Computer Design*, pp.328-333, Oct. 1992.

[15] E. J. Marinissen, et al., "Towards a standard for embedded core test:An example," *in Proc. IEEE International Test Conference*, pp.616-627, Sep. 1999.