CAD Contest A: X value equivalence checking

[*] Final project of Introduction to Electronic Design Automation, Spring 2020.

1st Wan-Hsuan Lin
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
b06901054@ntu.edu.tw

2nd Yun-Rong Luo
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
b06901073@ntu.edu.tw

3rd Yu-Ling Hsu
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
b06901081@ntu.edu.tw

***Abstract*—We encode AIG network into dual rail AIG, and modify the abc command *dcec* to do compatible equivalence checking. We solved 7 cases out of 9 released Alpha cases. We solved all NEQ cases. As for EQ cases, we solved case1, case5, and case6. Case 3 and 8 timed out due to SAT solving in *dcec*.**

I. Introduction

We first introduce the terminology that will be used.

- CE: compatible equivalence.
- XAIG: XAIG is a 4-bit in 2-bit out gate implementation of dual rail gates using conventional AIG data structure. This implementation preserves all properties of AIG. We can apply all conventional algorithms on AIG circuit to our XAIG circuit. We used 00 and 01 to represent 1'b0 and 1'b1, correspondingly. And we used 10 and 11 to represent 1'bx. In this way, the first rail (LSB) will represents the original AIG circuit. The second rail (MSB) represents the X value. Figure 1 is an example of a small ordinay AIG circuit, and Figure 2 is its corresponding XAIG circuit schematic.

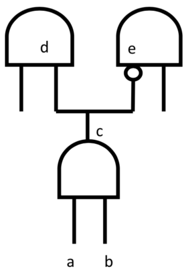

Fig. 1. Ordinary AIG.

- CMiter: We replaced the Miter circuit in *dcec* to our CMiter in order to solve compatible equivalence instead of functional equivalence.
- C1 and C2 circuit: We named the subcircuit that represents the first rail (00 and 01) as C1. And we named the subcircuit that represents the second rail (10 and 11) as C2.

Special thanks to Professor Jie-Hong Roland Jiang.

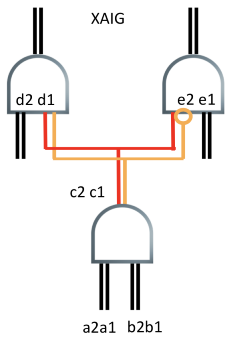

Fig. 2. XAIG.

II. Implementation

A. Reading the verilog files

We used the tool *yosys* to do frontend files reading. We used yosys's command *flatten* to replace the _DC gates and _HMUX gates in gf.v and rf.v into _DC's and _HMUX's corresponding gate-level design. We then used the command *aigmap* to transform the network into AIG circuit. We used the command *setundef* to transform the 1'bx in _DC into constant latches. Since the contest only deals with combinational circuits, we can use constant latches to identify the 1'bx in the transformed AIG circuit. We write out the AIG as ascii-aiger files.

B. Constructing the XAIG circuit

We read in the ascii-aiger files and constructed the C1 circuit. We need to take special care to the latches. Since we encoded 1'bx as 10 and 11, latches in C1 circuit can be connected to either constant-0 or constant-1. We then constructed the C2 circuit by depth-first-search order from primary inputs to primary outputs. C1 and C2 share the same primary input list and constant-0. We maintained a mapping from each C1 signal to C2, to indicate that they belongs to the same XAIG 2-bit output. Primary inputs in C1 maps to constant-0 in C2. Latches in C1 maps to constant-1 in C2. Every AIG gates in C1 are mapped to C2 in the way shown in Figure 3. Note that after yosys generates the ascii-aiger files, the order of primary inputs and primary outputs may be different between the golden and the revised circuit. We used their symbols to unify their orders of primary inputs and primary outputs.

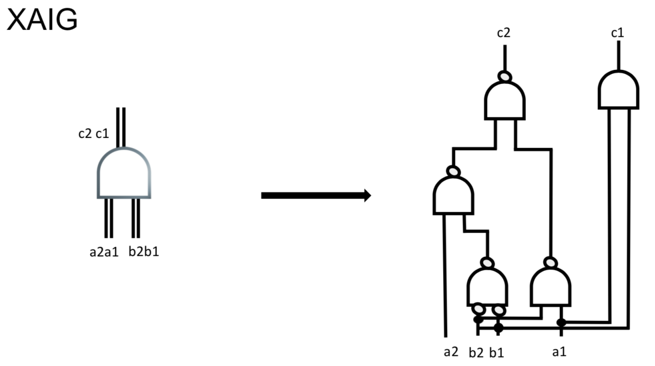

Fig. 3. The implementation of XAIG.

C. Random simulation on the XAIG to witness the NEQ early

We generated random size_t patterns and do simulation on the XAIG. We compare the golden circuit's C1's and C2's primary outputs signal with that of revised circuit to witness the NEQ. We do 250 iterations of simulation for each cases. For each iteration, we use a size_t signal to simulate 64 patterns. We are able to witness case4 case7, and case9 by random simulation.

D. Modify abc command dcec to do compatible equivalence checking

Originally, the corresponding primary outputs (*PO*s) of two networks were XORed by a function *Aig_Miter()* under command *dcec*. However, in order to perform *CE* checking, we replace the normal Miter circuit of *PO*s with *CMiter* in the way shown in Figure 4.

Note that XAIG has two-bit output, so we need to deal with four bits every time we generate a *CMiter*. Once an output of *CMiter* is 1, the two networks are not compatible equivalence.

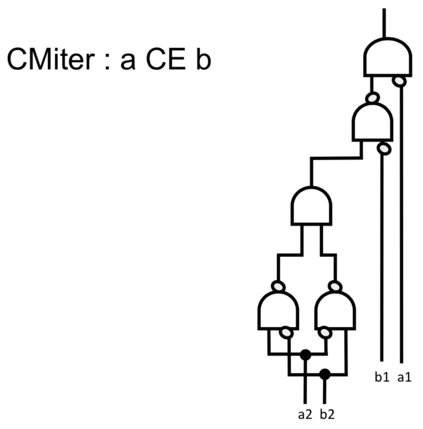

Fig. 4. CMiter.

E. Output counter-example when dcec proves NEQ

We follow the original way how *dcec* implements *FRAIG*, and SAT solver was called to check whether the *CMiter*ed circuit is satisfiable. If it is SAT, we extract the data member *pModel*, where *abc* stores the counter examples of the network, and write it into the output file. Otherwise, we only write out the string "EQ" to the output file as CAD Contest defined.

III. Experimental Analysis

We used C and C++ to implement our algorithm. We used yosys to parse input and convert verilog file to aag format and use abc to do FRAIG and solve SAT problem.

A. Results of Alpha cases

TableI shows our experimental result. The column yosys record the runtime spend on parsing files, circuit construction, and simulation. The column abc record the runtime spend on FRAIG and SAT solving. For NEQ cases, our own simulation before abc can find counterexample in early stage and prove NEQ effectively. For NEQ case that can't be proved by simulation, abc with dcec command can prove NEQ in a short time. For EQ cases, we can't prove EQ in limited time for case 3 and case 8. For EQ cases such as case 1, 3, and 8, most of runtime spend on the SAT solving stage. For EQ case 5 and case 6, we proof EQ in a short time. However, the circuit size of case 1 is mush smaller than case 5 and case 6 while it requires more time to prove EQ. For cases with large circuit size, parsing files and circuit construction also cost a lot of time.

TableII shows our experimental result compared with the result of top three contestants in alpha test. The first column of each rank is their total runtime compared with our total runtime and the second column is their total runtime compared with our runtime cost on abc. For EQ cases, our performance for case 5 and case 6 is much better than other contestants. For NEQ cases, our runtime is more than others. However, if we split our procedure into two stage, yosys and abc, we can see that abc, the main procedure to proof EQ, cost only a little portion of runtime. If we only consider the runtime cost by abc, our performance is much better on average.

To analysis the runtime difference for test case 5 and 6, we do the experiment. In the experiment, we try different command in abc to prove equivalence. The result is in TableIII The first column is used command DCEC in abc to do equivalence checking, while the second column is used CEC to do equivalence checking. We only compared three cases which have entered abc stage in former experiment. The runtime difference for NEQ case is small, while the runtime difference for EQ cases is significant.

B. Discussion

According our experimental results, parsing files and circuit construction account for a large portion of our runtime. The main procedure to do compatible equivalence checking only accounts for a small portion. Thus, if we improve our method to do parsing, our implementation is expected to be more

TABLE I
RUNTIME ANALYSIS

	Result	yosys	abc	Total
case1	EQ	0.96	1259.31	1260.27
case2	NEQ	1.2	0.1	1.3
case3	TO	-	-	-
case4	NEQ	5.46	0	5.46
case5	EQ	10.12	3.52	13.64
case6	EQ	9.7	3.15	12.85
case7	NEQ	4.02	0	4.02
case8	TO	-	-	-
case9	NEQ	12.264	0	12.264

efficient. The reason that case 3 and 8 is failed to be proven is that the SAT solver can't prove the CNF. We will modify the rule to add learnt clauses and reduce the number of literal in CNF to improve the SAT solver.

IV. FUTURE WORK

The following are the future work we would like to do to improve our algorithms and implementation.

- The current implementation does three times file reading and three times file writing, which is too redundant. In the future, we want to implement our entire algorithm in *abc* and reduce the number of file reading and file writing.
- We would like to use the Theorem proposed in the Appendix to witness EQ before calling *dcec*. We will use a container to store signal pairs $< g_1, g_2 >$, which means that g_1 is CE to g_2. We will collect all $< g_1, g_2 >$ pairs by depth-first-search order and check whether all primary outputs of the golden circuit and the revised circuit is in the CE container. In this way, we can try to witness EQ cases before calling *dcec* and doing SAT solving.
- *dcec* implements a function called balancing XOR sto balance the depth of each Miter circuit. In our case, we modified the Miter into CMiter; therefore, *abc* cannot recognize the CMiter circuit and do balancing. We would also like to modify this part and make *abc* capable of recognizing CMiter and do balancing.

TABLE II
RUNTIME ANALYSIS COMPARED WITH THE TOP THREE CONTESTANT IN ALPHA TEST

	rank0		rank1		rank2		Our	
case1	0.1504	0.1505	0.1579	0.158	0.5391	0.5395	1	0.9992
case2	0.5153	6.7	0.7076	9.2	1.6307	21.2	1	0.0769
case4	0.3333	0	0.8846	0	0.9212	0	1	0
case5	3.0586	11.852	4.554	17.648	5.5087	21.347	1	0.258
case6	3.2817	13.387	4.623	18.86	5.9992	24.473	1	0.2451
case7	0.1293	0	0.1915	0	0.7786	0	1	0
case9	0.0220	0	0.1076	0	0.6555	0	1	0

TABLE III
RUNTIME WITH DIFFERENT COMMAND OF ABC

	abc with DCEC	abc with CEC
case2	0.1	0.09
case5	3.52	32.48
case6	3.15	35.38

V. CONTRIBUTION

Contribution of authors.

- **Wan-Hsuan Lin**:
  (1) Bridging the surface of *yosys* and *abc*
  (2) Tracing *abc* and making *CMiter*
  (3) Experiment (4) PPT (5) Report
- **Yun-Rong Luo**:
  (1) Tracing and managing the part of *yosys*
  (2) Transfering the aig network to XAIG one
  (3) Writing the simulation function (4) PPT (5) Report
- **Yu-Ling Hsu**:
  (1) Tracing *abc* and making *CMiter*
  (2) Writing out the counter example result
  (3) Experiment
  (4) Report

REFERENCES

[1] ICCAD Contest 2020: Problem A,
`http://iccad-contest.org/2020/problems.html`
[2] Yosys,
`http://www.clifford.at/yosys/`
[3] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification.*
`https://people.eecs.berkeley.edu/ alanmi/abc/`

APPENDIX

Theorem: O1 CE O2 if V1 CE V2 and V1' CE V2'.

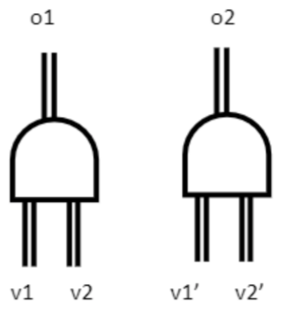

Fig. 5. Theorem that can be used to improve EQ checking.