Problem A: X-value Equivalence Checking

Chih-Jen (Jacky) Hsu, Chi-An (Rocky) Wu, Ching-Yi Huang, and Chung-Han Chou
Cadence Design Systems, Inc.

1. Introduction

In this contest, we formulate a problem "X-value Equivalence Checking". Contestants need to check if two combinational netlists are equivalent, where the netlists have X value in addition to binary values, 0 and 1. For evaluation, correctness is required, and the team solving the most cases wins.

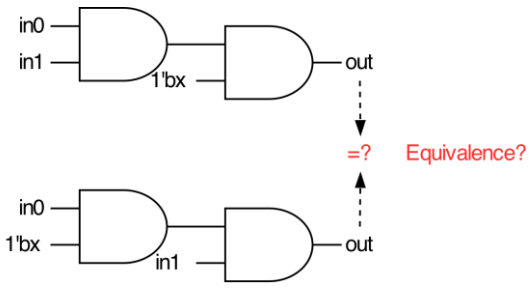

Figure 1. Illustration of X-value equivalence checking.

Equivalence checking [1][2] plays a crucial step in ASIC design flow. Equivalence means that two different design implementations perform the same function. Therefore, equivalence checking verifies the correctness of optimizations and allows synthesis tools to perform aggressive optimizations. In ASIC design flow, a design implementation must pass equivalence checking to guarantee that it meets the original specification. In industry, equivalence checking applies formal methods to compare two designs. One of the key challenges is that the netlist has X-value.

For industrial applications, X-value would increase the complexity of formal verification. The previous techniques and heuristics [1][2] are effective for equivalence checking on binary-value netlists, but are not applicable for netlists with X-value. Although don't-care modeling in [3] utilizes don't-care circuits which can provide the correct model of X-value equivalence checking, it is not scalable when the synthesized netlist has X-value optimization. Therefore, formal verification for netlists with X-value is still a challenging topic and there are wide applications that can gain benefits from the techniques of X-value equivalence checking.

Low-power equivalence checking is one of the applications, which can gain benefits from the techniques of X-value equivalence checking. Low-power equivalence checking is a problem of checking if two netlists are equivalent, where both netlists would generate X-value under the power-shutoff behavior [4]. When a gate works on the power-on state, it outputs a binary value, 0 or 1, depending on its input values. When a gate works on the power-off state, it outputs 1'bx, the corruption value. Therefore, to verify the equivalence of two low-power netlists, we need to consider the power-corruption values propagating in the netlists.

The problem of comparing an RTL design to the synthesized netlist is also an application of X-value equivalence checking. X-value is used in HDL specification [5]. Implementation tools would synthesize the design with X-value into a logical netlist which exhibits binary-value functions. X-value equivalence checking would verify the synthesis correctness.

In this contest, we look for useful lemmas, techniques, and heuristics that can enable X-value equivalence checking. In the following context, we first introduce the MUX gate and the DC gate that model our input netlists. Next, we define the "equivalence" of netlists with X-value. Then, we describe the input and output formats of the given testcases. Finally, we explain the evaluation criteria: Basically, correctness is a necessary requirement and the team who solves the most testcases wins.

2. Background

Verilog Language Reference Manual (LRM) [5] describes the X-value and its simulation behavior. In this contest, we use the same definition and behavior for simulating the netlist with X-value.

2.1 MUX gate and DC gate

In this contest, we only consider combinational netlists. A combinational netlist is a directed acyclic graph with gates as its nodes. The gates include the primitive gates defined in Verilog LRM [5], MUX gate, and DC gate. MUX gate and DC gate are the essential building blocks for modeling the industrial netlists. Their definitions in the Verilog language are as follows:

```
module _HMUX(O, I0, I1, S);
   output  O;
   input   I0, I1, S;
   assign O = S?I1:I0;
endmodule
```

For MUX gate (_HMUX), when I0 and I1 are the same value $v \in \{0, 1, 1'bx\}$, O would be v regardless of what value S is. For example, when I0=1'b1, I1=1'b1, and S=1'bx, the output O would be 1'b1. Furthermore, when I0=1'b0, I1=1'b1, and S=1'bx, the output O would be 1'bx.

```
module _DC(O, C, D);
   output O;
   input C, D;
   assign O = D?1'bx:C;
endmodule
```

For DC gate (_DC), O would be 1'bx when either $D \in \{1, 1'bx\}$ or C = 1'bx. We will describe the complete truth table in the appendix section.

2.2 X-value output of combinational netlists

Verilog [5] describes the X-value and its simulation behavior. For a combinational netlist, we use the same simulation behavior of LRM to compute its output values. An output value would be one

of {0, 1, x} transitively propagated by the given input patterns. Therefore, for a combinational netlist, the output values are deterministic to the given input patterns.

Furthermore, to focus on the X-value, we simplify the contest problem:

1. The netlist has no z-value or floating signals.
2. Input patterns only have binary values, 0 and 1.

Given two combinational netlists with the identical primary inputs and primary outputs, we can compute and compare their output values based on the given input patterns.

2.3 Compatible equivalence

We define the compatible equivalence for comparing two values $a, b \in \{0,1,x\}$, we call *a* is compatible equivalent to *b* if

$$a, b \in \{(0,0), (1,1), (x,0), (x,1), (x,x)\}$$

On the contrary, we call *a* is not compatible equivalent to *b* if

$$a, b \in \{(0,1), (1,0), (0,x), (1,x)\}$$

We will use "equivalent" to represent "compatible equivalent" and use "non-equivalent" to represent "not compatible equivalent" in this problem.

2.4 Compatible equivalence of two combinational netlists

Given two combinational netlists (with the identical primary inputs and primary outputs $o_{i=1...N}$), *G* and *R*, *G* is compatible equivalent to *R* if

> *Under all input combinations, the output values of G are compatible equivalent to the output values of R for all primary outputs* $o_{i=1...N.}$

On the contrary, *G* is non-equivalent to *R* if

> *There exists an input pattern such that there is a primary output* o_i, *where the value of* o_i *in G is not compatible equivalent to the value of* o_i *in R.*

Example:

Figure 2 shows a non-equivalent case. For the input pattern {in=1'b1, a=1'b1, b=1'b0}, the output value of netlist *G* is out=1'b0 but the output value of netlist *R* is out=1'bx. Here is the detail implications.

> **Golden:**
> a=1 ∧ b=0 -> en1b=1 -> en1=0 -> n2=0
> a=1 -> en2b=0
> en2b=0 ∧ n2=0 -> **out=1'b0**
> **Revised:**
> a=1 ∧ b=0 -> en1b=1 -> n1=n2=1'bx -> **out=1'bx**

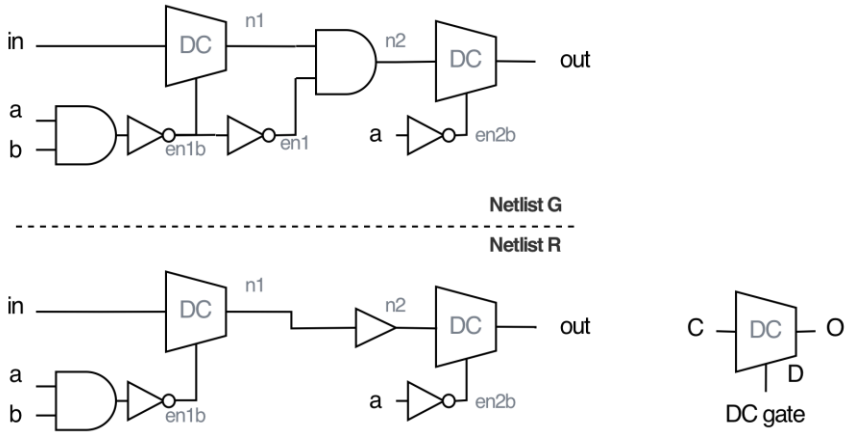

Figure 2. The non-equivalent cases in X-value equivalence checking

Note that the compatible "equivalence" does not have the commutative property, i.e., *a* is equivalent to *b* doesn't imply that *b* is equivalent to *a*. This definition is for the industrial requirement, where we can allow the optimization from X to any function but cannot allow optimize 0/1 value into X value.

2.5 Semantic of 1'bx [optional reading]

In this document, the behavior of 1'bx and its propagation rules follow the simulation semantic. The simulation semantic has well definition like Verilog LRM [5]. By using the definition, we can eliminate ambiguity to 1'bx and have simulators to validate the contest result.

For industrial designs, 1'bx has multiple semantics depending on the design intent. Some behave don't-care, some behave unknown, and some behave errors. Industrial applications like synthesis would process the 1'bx based on the design intent.

3. Problem formulation and input output format

Given two combinational netlists *G* and *R*, contestants need to write a program to answer if *G* is compatible equivalent to *R*. If they are non-equivalent, contestants need to find out an input pattern, called witness, which generates non-equivalent outputs to *R* and *G*.

3.1 Program requirement

The requested program must be run on a Linux system. The time limit of running each testcase is 1800 seconds. Parallel computation with multiple threads or processes is not allowed. The program accepts three arguments:

```
./xec <golden.v> <revised.v> <output>
```

- <golden.v> and <revised.v> are input files that describe two netlists *G* and *R* in Verilog format, respectively.

- <output> is the output file that shows the checking result: equivalence or non-equivalence with a witness.

3.2 Input format

The netlists <golden.v> and <revised.v> are the combinational netlists composed of

- primitive gates (and, or, nand, nor, not, buf, xor, xnor)
- DC gates and MUX gates,
- constant values (1'b1, 1'b0, and 1'bx)

The netlist is in Verilog language and in the flatten view. The format is as follows:

```
module top ( <name0>, <name1>, … );
input <name0>, <name1>, …;
output <name0>, <name1>,…;
wire <name0>, <name1> , …;
<gate type> <name>( <name0>, <name1>, … );
…
endmodule
```

3.3 Output format

The first line in the output file <output> shows the checking result. "EQ" means equivalence and "NEQ" means non-equivalence. If the result is non-equivalence, contestants also need to show a witness.

The output of equivalence result should be in one line which shows EQ

```
EQ
```

The output of non-equivalence result should follow the following format. For the witness, you need to output the complete pattern, even if some of the primary inputs are not responsible for the non-equivalence.

```
NEQ
<input name1> <0|1>
<input name2> <0|1>
<input name3> <0|1>
...
```

3.4 Example

The following two Verilog files describe the two input netlists, *G* and *R*, in Figure 2.

```
module top(in, a, b, out);
input in, a, b;
output out;
wire n2, n1, en1, en2b, en1b, out, b, a, in;
  nand en1b_ins(en1b, a, b);
  not en2b_ins(en2b, a);
  not en1_ins(en1, en1b);
  _DC buf1(n1, in, en1b);
  and iso(n2, n1, en1);
  _DC buf2(out, n2, en2b);
endmodule
```

Input file: golden.v

```
module top(in, a, b, out);
input in, a, b;
output out;
wire n2, n1, en2b, en1b, out, b, a, in;
  nand en1b_ins(en1b, a, b);
  not en2b_ins(en2b, a);
  _DC buf1(n1, in, en1b);
  buf n2_ins(n2, n1);
  _DC buf2(out, n2, en2b);
endmodule
```

Input file: revised.v

The expected output would be:

```
NEQ
in 1
a 1
b 0
```

Output file: output

4. Evaluation criteria

In this problem, correctness is a necessary requirement. A team will be disqualified from the contest, i.e., won't be ranked, if the team reports a false result (either EQ or NEQ) or an incorrect witness for any testcase.

Here are the evaluation criteria:

- Correct result is required.
- The team solving the most testcases wins.
- When multiple teams solve the same number of testcases, we count the total runtime for only the "solved" testcases. The team spending less runtime wins.
- The time limit for solving a case is 1800 seconds.

We will publish at least 10 testcases with their expected outputs before the beta release. Additionally, we will use hidden testcases for the alpha, beta, and final evaluations. Note that public testcases are excluded in the final evaluation.

5. Appendix:

Here are the truth tables of DC gate and MUX gate

MUX gate: output value of MUX gate for different values of S

S=0

In1 \ In0	0	1	x
0	0	1	x
1	0	1	x
x	0	1	x

S=1

In1 \ In0	0	1	x
0	0	0	0
1	1	1	1
x	x	x	x

S=x

In1 \ In0	0	1	x
0	0	x	x
1	x	1	x
x	x	x	x

DC gate: output value of DC gate:

D \ C	0	1	x
0	0	1	x
1	x	x	x
x	x	x	x

For the truth tables of the other used primitive gates, please refer to [5].

Reference

[1] A. Kuehlmann and F. Krohm. “Equivalence checking using cuts and heaps,“ *Design Automation Conference* (DAC ’97).

[2] E. I. Goldberg, M. R. Prasad and R. K. Brayton, "Using SAT for combinational equivalence checking," Design, Automation and Test in Europe. (DATE 2001)

[3] A. Mishchenko, R. Brayton, J.-H. Jiang, and S. Jang. “Scalable don’t-care-based logic optimization and resynthesis,” International symposium on Field programmable gate arrays (FPGA ’09).

[4] IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems," in IEEE Std 1801-2018

[5] IEEE Standard Verilog Hardware Description Language," in IEEE Std 1364-2001 ,