Towards Formal Relaxed Equivalence Checking in Approximate Computing Methodology

Lukáš Holík, Ondřej Lengál, Adam Rogalewicz
Lukáš Sekanina, Zdeněk VAŠÍČEK, Tomáš Vojnar

Faculty of Information Technology
Brno University of technology
vasicek@fit.vutbr.cz

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Outline

- Approximate computing
- Functional approximations
- Relaxed equivalence checking
- Towards relaxed equivalence checking

The idea of approximate computing

- The requirement of exact numerical or Boolean equivalence between the specification and implementation of a circuit is relaxed in order to achieve improvements in performance or energy efficiency [Venkatesan et al., 2011].

- The requirement of exactness can be relaxed because of:
  1. limited perceptual capability of humans
  2. a golden result is impossible (or difficult) to define
  3. worst-case design would lead to large power consumption (process parameter variations are large in sub-45 nm technologies)

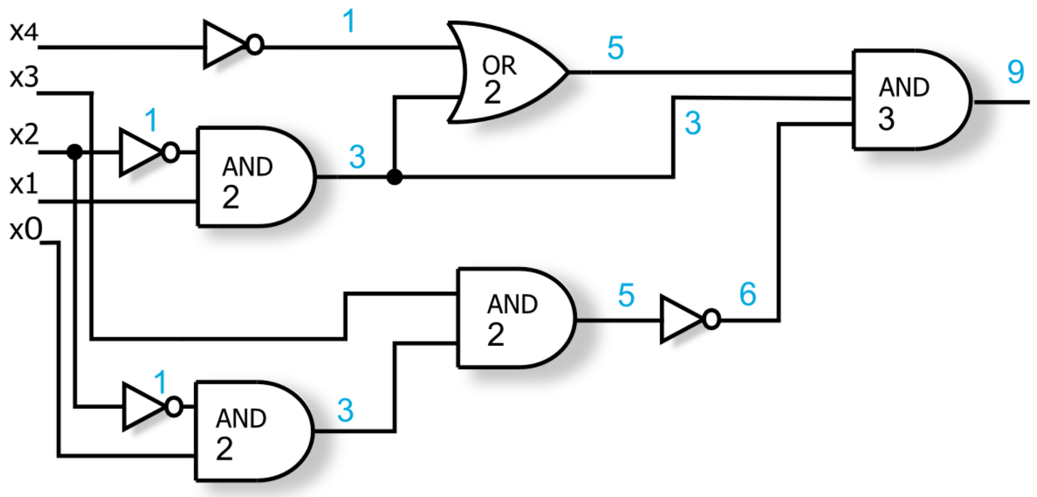
Approximate computing

- The concept of approximate computing has been developed in different ways and at various levels of computing stack, for example

- Software-level approximations
  - EnerJ: approximate data types in JAVA [Sampson el al., 2011]
  - Neural network replaces general purpose code [Esmaeilzadeh el a., 2013]
  - Axilog: language annotations [Yazdanbakhs, 2015]
- Specialized processors supporting approximate computing
  - Improving Efficiency of Extensible Processors by Using Approximate Custom Instructions, [Kamal et al., 2014]
- Hardware-level approximations
  - over-scaling based approximations
  - functional approximations

Over-scaling based approximations

- Principle: Approximations are introduced by over-clocking or voltage over-scaling (i.e. the circuits operate correctly under normal conditions).



- Limits: The over-scaling based approach works well when there exists few long paths in the target circuit.

Functional approximations

- Goal: To implement a slightly different Boolean function that has faster or more power-efficient implementation.

- Two main methodologies were developed
  - Ad hoc methods optimizing a particular component, e.g. multipliers [Kulkarni, 2011], adders [Gupta, 2013], median filters [Monajati, 2015]
  - Design automation methods
    SASIMI: Substitute-and-simplify [Venkataramani et al, 2013]
    SALSA: Systematic logic synthesis using Quality Constraint Circuits [Venkataramani et al, 2012]
    ABACUS: AST-based approach [Nepal et al., 2014]

Design automation methods – issues and limits

- Although approximate computing techniques have shown significant promise, moving them to the mainstream will require several issues to be addressed, foremost among which is the issue of modeling and analysis of accuracy. [Venkatesan, 2011]

Approach	Circuits	Accuracy measured using
ABACUS 2014	FIR filter, perceptron, block matcher	training data
ASLAN 2014	MPEG encoder	sequential quality constraint circuit
SASIMI 2013	Benchmarks, multipliers, adders, SAD	training data
SALSA 2012	Adders, multipliers, FIR, IIR, DCT	quality constraint circuit

Design automation methods – issues and limits

- Several approximate designs have been proposed that compromise accuracy in different ways; unfortunately there is no simple and systematic analysis methodology to evaluate quality of candidate designs and compare them with conventional designs or with each other.

The evaluation of quality based on training data cannot

- guarantee that a given approximate implementation meets my accuracy requirements.
- ensure that there are no bugs in an approximate implementation.
- be applied when only a neglible error is acceptable.

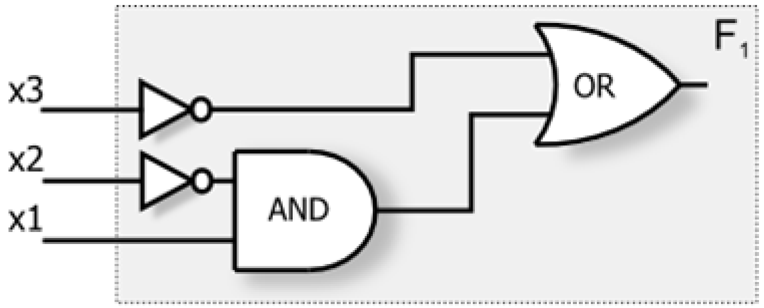
Relaxed equivalence checking

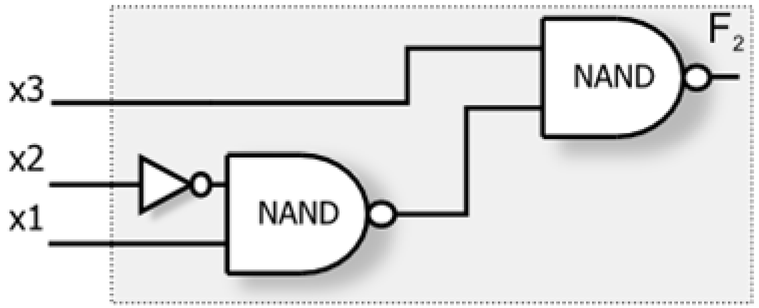
- Traditional formal verification techniques (combinational and sequential equivalence checking) are designed to prove exact Boolean equivalence between specification and implementation and do not directly address the previous questions.

- There is a need for relaxed equivalence checking, an approach which is able to prove the equivalence up to some bound.

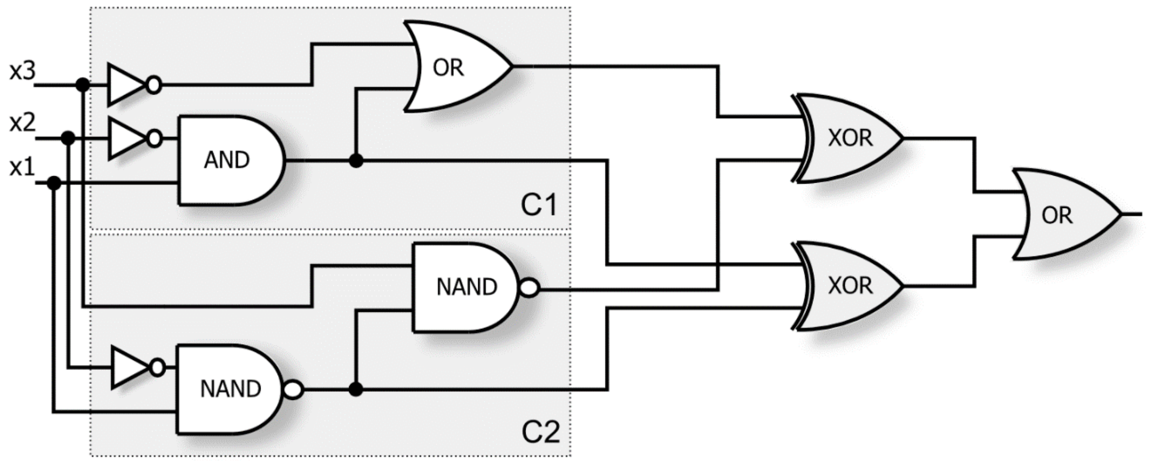
Combinational Equivalence Checking

- Goal: To prove that F1(x) ≡ F2(x) or that F1(x) ≠ F2(x)





SAT-based Combinational Equivalence Checking



CNF

...

$(\overline{x_6} + x_9 + x_{12})(x_6 + \overline{x_9} + x_{12})$
$(\overline{x_6} + \overline{x_9} + \overline{x_{12}})(x_6 + x_9 + \overline{x_{12}})$
$(x_{11} + x_{12} + \overline{x_{13}})(x_{13} + \overline{x_{11}})$
$(x_{13} + \overline{x_{12}})(x_{13})$

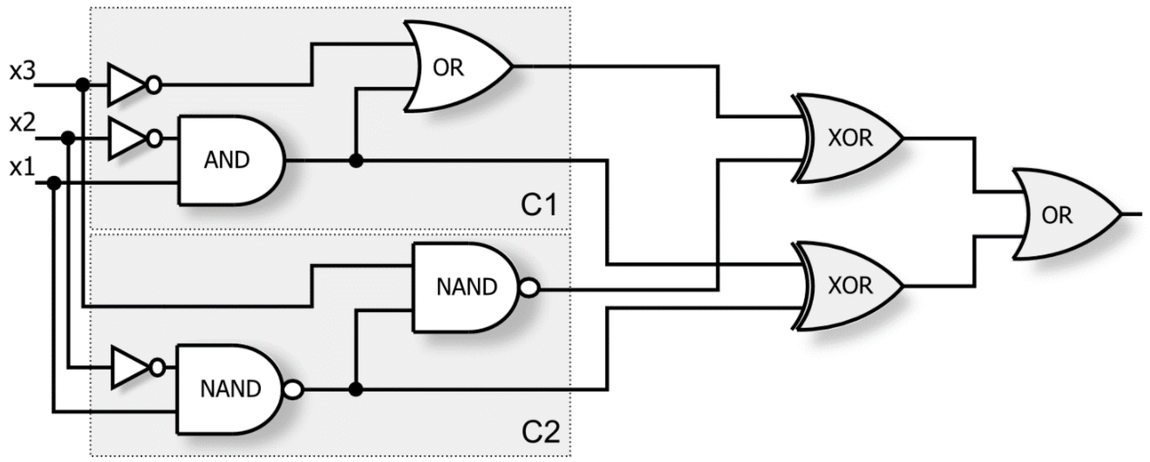
SAT solver

result: SATISFIABLE

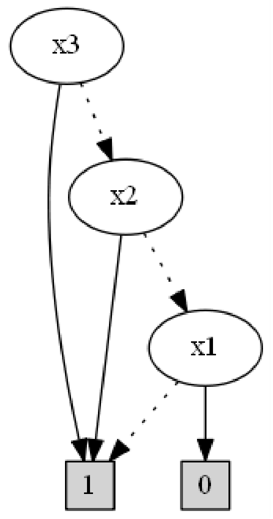
model: 001 111110101

x_1 x_2 x_3

BDD-based Combinational Equivalence Checking



ROBDD

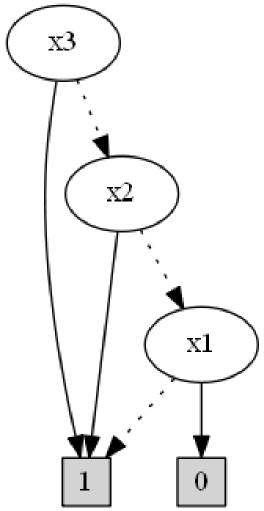


SatOne?

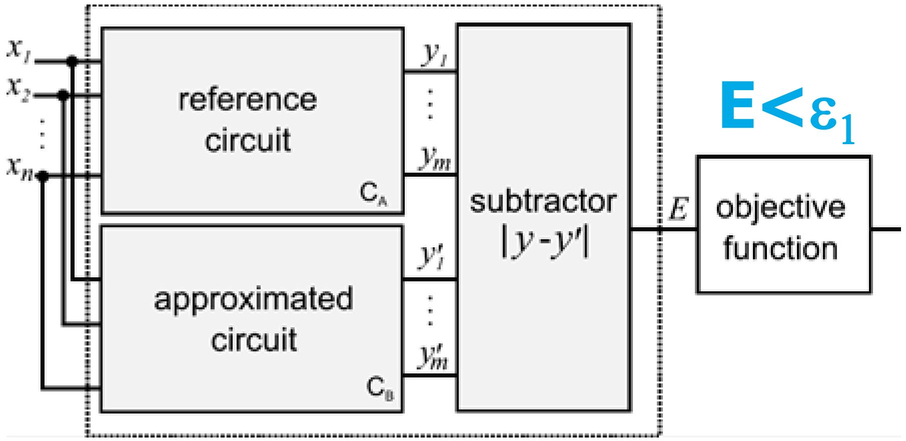
result: TRUE

How to determine arithmetic error?

ROBDD



SatCount ?



$E<\varepsilon_1$

To calculate average arithmetic error, ε_i can be successively increased from 1 to $\varepsilon_{\max}$

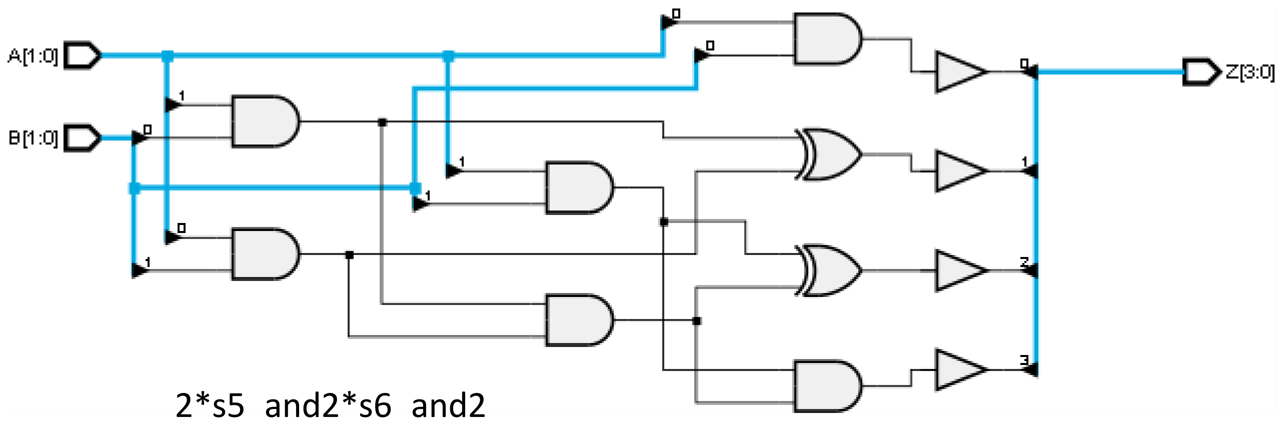
result: 8

Traditional methods and arithmetic circuits

- Known issues of the common methods in verification of arithmetic circuits
  - BDD application to verification of arithmetic circuits is limited by the prohibitively high memory requirement for complex arithmetic circuits, such as multipliers.
  - SAT-based approaches are known to be computationally expensive and not scalable.
- Proposed approach: Use a pseudo-Boolean polynomial representation [Ciesielski, 2015] of arithmetic circuits to determine the quality of an approximation.

Combinational equivalence checking using polynoials

pseudo-Booelan polynomial



2-bit accurate multiplier

z3 = s13_and2

z2 = s12_xor2

z1 = s8_xor2

z0 = s4_and2

s12_xor2 = s7_and2+s9_and2 - 2*s7_and2*s9_and2

s13_and2 = s7_and2 * s9_and2

s8_xor2 = s5_and2+s6_and2 - 2*s5_and2*s6_and2

s9_and2 = s5_and2 * s6_and2

s4_and2 = a0 * b0

s5_and2 = a1 * b0

s6_and2 = a0 * b1

s7_and2 = a1 * b1

Specification:

z0*2^0 + z1*2^1 + z2*2^2 + z3*2^3 - (a0*b0+2*a0*b1+2*a1*b0+4*a1*b1)

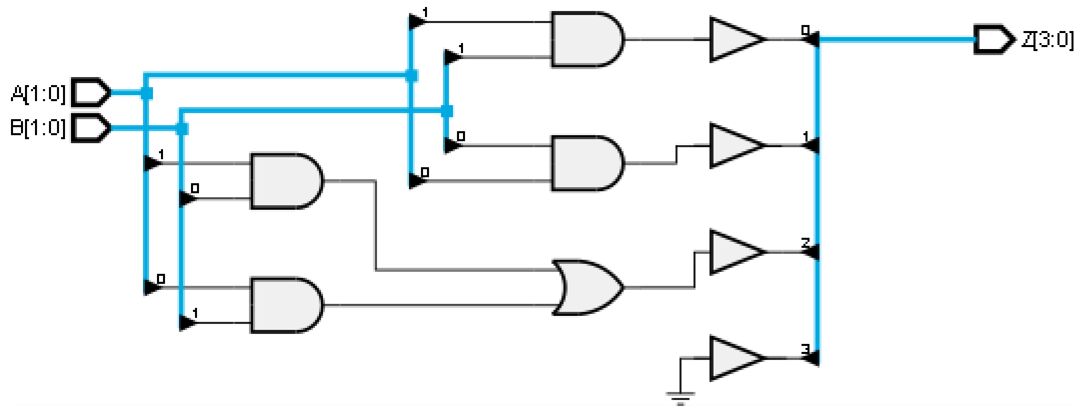
pBSolver

result: 0

Verification of Gate-level Arithmetic Circuits by Function Extraction [M. Ciesielski, C. Yu, D. Liu, and W. Brown, 2015]

Combinational equivalence checking using polynoials

pseudo-Booelan polynomial



z3 = 0
z2 = s5_and2
z1 = s8_or2
z0 = s6_and2
s8_or2 = s4_and2 + s7_and2 - s4_and2 * s7_and2

s7_and2 = b0 * a1
s6_and2 = a0 * b0
s5_and2 = b1 * a1
s4_and2 = b1 * a0

2-bit approximate multiplier [Kulkarni et al., 2011]

Specification:

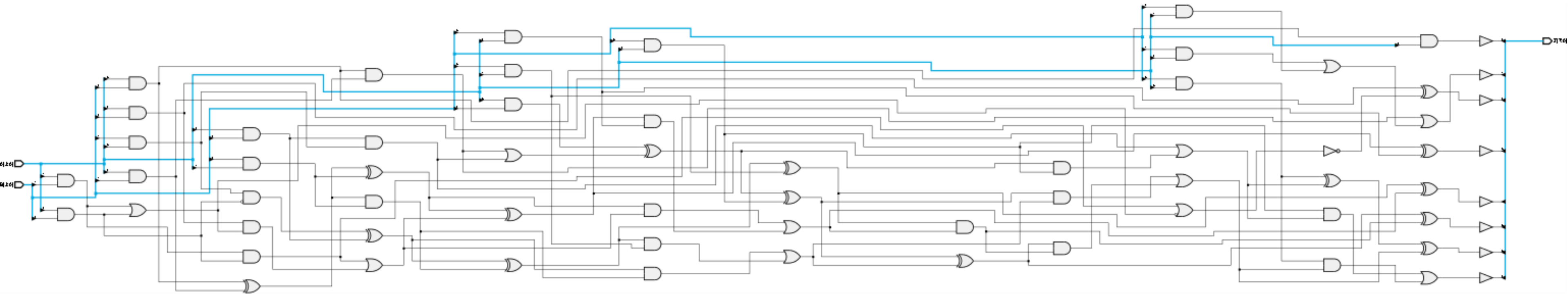
z0*2^0 + z1*2^1 + z2*2^2 + z3*2^3 - (a0*b0+2*a0*b1+2*a1*b0+4*a1*b1)

pBSolver

result: -2*a0*a1*b0*b1

Error is equal to -2 IFF a0=a1=b0=b1=1

4-bit approximate multiplier [Vasicek, 2014]



z7 = s71_or2
z6 = s69_xor2
z5 = s66_xor2
z4 = s60_xor2
z3 = s56_xor2
z2 = s34_xor2
z1 = s48_or2
z0 = s29_and2
s71_or2 = s65_and2 + s70_and2 - s65_and2 * s70_and2
s70_and2 = s44_and2 * s68_or2
s69_xor2 = s64_xor2+s68_or2 - 2*s64_xor2*s68_or2
s68_or2 = s63_and2 + s67_and2 - s63_and2 * s67_and2
s67_and2 = s62_xor2 * s61_and2
s66_xor2 = s62_xor2+s61_and2 - 2*s62_xor2*s61_and2
s65_and2 = s37_and2 * s59_or2
s64_xor2 = s44_and2+s59_or2 - 2*s44_and2*s59_or2
s63_and2 = s55_xor2 * s58_or2
s62_xor2 = s55_xor2+s58_or2 - 2*s55_xor2*s58_or2
s61_and2 = s53_xor2 * s57_or2
s60_xor2 = s53_xor2+s57_or2 - 2*s53_xor2*s57_or2
s59_or2 = s32_and2 + s54_and2 - s32_and2 * s54_and2
s58_or2 = s52_and2 + s51_and2 - s52_and2 * s51_and2
s57_or2 = s50_and2 + s46_and2 - s50_and2 * s46_and2

s56_xor2 = s45_xor2+s13_and2 - 2*s45_xor2*s13_and2
s55_xor2 = s49_xor2+s37_and2 - 2*s49_xor2*s37_and2
s54_and2 = s49_xor2 * s37_and2
s53_xor2 = s47_xor2+s16_and2 - 2*s47_xor2*s16_and2
s52_and2 = s40_xor2 * s39_and2
s51_and2 = s47_xor2 * s16_and2
s50_and2 = s38_xor2 * s42_or2
s49_xor2 = s22_and2+s35_or2 - 2*s22_and2*s35_or2
s48_or2 = s25_and2 + s30_or2 - s25_and2 * s30_or2
s47_xor2 = s40_xor2+s39_and2 - 2*s40_xor2*s39_and2
s46_and2 = s45_xor2 * s13_and2
s45_xor2 = s38_xor2+s42_or2 - 2*s38_xor2*s42_or2
s44_and2 = b3 * a3
s42_or2 = s27_and2 + s26_and2 - s27_and2 * s26_and2
s40_xor2 = s20_and2+s18_and2 - 2*s20_and2*s18_and2
s39_and2 = s28_xor2 * s19_and2
s38_xor2 = s28_xor2+s19_and2 - 2*s28_xor2*s19_and2
s37_and2 = b3 * a2
s35_or2 = s25_and2 + s32_and2 - s25_and2 * s32_and2
s34_xor2 = s11_and2+s31_nor2 - 2*s11_and2*s31_nor2
s32_and2 = s18_and2 * s15_and2
s31_nor2 = 1-s27_and2-s24_inva+s27_and2*s24_inva
s30_or2 = s21_and2 + s8_and2 - s21_and2 * s8_and2

s29_and2 = s9_and2 * a0
s28_xor2 = s9_and2+s17_and2 - 2*s9_and2*s17_and2
s27_and2 = s10_and2 * s12_and2
s26_and2 = s11_and2 * s23_or2
s25_and2 = s9_and2 * s17_and2
s24_inva = 1-s23_or2
s23_or2 = s10_and2 + s12_and2 - s10_and2 * s12_and2
s22_and2 = a3 * b2
s21_and2 = b1 * a0
s20_and2 = s15_and2 * s14_inva
s19_and2 = b2 * a1
s18_and2 = a2 * b2
s17_and2 = a2 * b1
s16_and2 = b3 * a1
s15_and2 = b1 * a3
s14_inva = 1-s12_and2
s13_and2 = b3 * a0
s12_and2 = a2 * b0
s11_and2 = a0 * b2
s10_and2 = a1 * b1
s9_and2 = a3 * b0
s8_and2 = b0 * a1

Specification:

z0*2^0 + z1*2^1 + z2*2^2 + z3*2^3 + z4*2^4 + z5*2^5 + z6*2^6 + z7*2^7 - (a0*b0+2*a0*b1+2*a1*b0+4*a0*b2+4*a1*b1+4*a2*b0+8*a0*b3+8*a1*b2+8*a2*b1+8*a3*b0+16*a1*b3+16*a2*b2+16*a3*b1+32*a2*b3+32*a3*b2+64*a3*b3)

4-bit approximate multiplier [Vasicek, 2014]

Result:

E = -a0*b0 + a2*a3*b0*b1 + 2*a0*a1*b0*b1

Input variables: a0,a1,a2,a3 b0,b1,b2,b3

Quality metrics:

Worst-case error: 2

Number of invalid responses: 4 · 17 = 68

Average error: 4 · 18/68 = 1.05

#	a_3	a_2	a_1	a_0	b_3	b_2	b_1	b_0	E
1	0	0	0	1	x	x	0	1	-1
2	0	0	1	1	x	x	0	1	-1
3	0	1	0	1	x	x	0	1	-1
4	0	1	1	1	x	x	0	1	-1
5	1	0	0	1	x	x	0	1	-1
6	1	0	1	1	x	x	0	1	-1
7	1	1	0	1	x	x	0	1	-1
8	1	1	1	1	x	x	0	1	-1
9	0	0	0	1	x	x	1	1	-1
10	0	0	1	1	x	x	1	1	1
11	0	1	0	1	x	x	1	1	-1
12	0	1	1	1	x	x	1	1	1
13	1	0	0	1	x	x	1	1	-1
14	1	0	1	1	x	x	1	1	1
15	1	1	0	0	x	x	1	1	1
16	1	1	1	0	x	x	1	1	1
17	1	1	1	1	x	x	1	1	2

Conclusion

- A notion of formal relaxed equivalence checking has been briefly introduced.
- Even if some of the equivalence checking methods can be directly extended to support relaxed equivalence checking, the scalability of common approaches represents a limiting factor.
- A suitable formal relaxed equivalence checking method needs to be developed.

Thank You For Your Attention !