# Equivalence Checking using Cryptol

Sean Weaver

# Overview

- Recent equivalence checking technology
- Using Cryptol to evaluate FPGA-based cryptographic cores
- Some of our results

# Equivalence Checking

- Given two Boolean functions, prove whether or not two they are functionally equivalent

- This talk focuses specifically on the mechanics of checking the equivalence of pairs of combinational circuits

# Types of Circuits

- Combinational Circuit
  - No state-holding elements
  - No feedback loops
  - Output is a function of the current input
- Sequential Circuit
  - Can have state-holding elements
  - Can have feedback loops
  - Must transform (e.g. BMC) into a combinational circuit for equivalence checking

# Circuit Equivalence Checking

- Checking the equivalence of a pair of circuits
  - For all possible input vectors ($2^{\#\text{input bits}}$), the outputs of the two circuits must be identical
  - Equivalence Checking is coNP-Hard
  - However, the equivalence check of circuits with "similar" structure is easy [1]
  - We must be able to
    - identify shared structure and
    - need a tool that can efficiently solve NP-Complete problems (Satisfiability solver, BDDs, Gröbner Basis solver, etc.)

1. E. Goldberg, Y. Novikov. How good can a resolution based SAT-solver be? SAT-2003, LNCS 2919, pp. 35-52.

# Equivalence Checking Uses

- Formal Verification
  - Prove whether a low level implementation matches a high level, or mathematical, specification
- Verifying Compiler
  - Maintain the functionality of generated code
- Version Control
  - Use previous implementations to maintain the correctness of future implementations
- Design Synthesis and Optimization
  - Automatically reduce and optimize netlists
- Inversion
  - Prove whether encode and decode functions are inverses of each other

# Functional Verification of Hardware Design

**A reasonable** *functional specification* **of any 1-bit adder**:

$$(X \Leftrightarrow (A \wedge \bar{B} \wedge \bar{C}) \vee (\bar{A} \wedge B \wedge \bar{C}) \vee (\bar{A} \wedge \bar{B} \wedge C) \vee (A \wedge B \wedge C)) \wedge$$
$$(Y \Leftrightarrow (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)).$$

**A proposed** *implementation* **of a 1-bit adder**:

$$\begin{aligned}
&(u \Leftrightarrow (A \wedge \bar{B}) \vee (\bar{A} \wedge B)) \wedge \\
&(v \Leftrightarrow u \wedge C) \wedge \\
&(w \Leftrightarrow A \wedge B) \wedge \\
&(X \Leftrightarrow (u \wedge \bar{C}) \vee (\bar{u} \wedge C)) \wedge \\
&(Y \Leftrightarrow w \vee v).
\end{aligned}$$

Call these formulas $\psi_S(A, B, C, X, Y)$ and $\psi_I(A, B, C, X, Y, u, v, w)$.

The theorem we are trying to prove is:

$$\psi_S(A, B, C, X, Y) \Leftrightarrow \exists u, v, w : \psi_I(A, B, C, X, Y, u, v, w).$$

# Functional Verification of Hardware Design

Given that the input variables (A, B, C) are equivalent, verify output variables (X, Y) are equivalent.

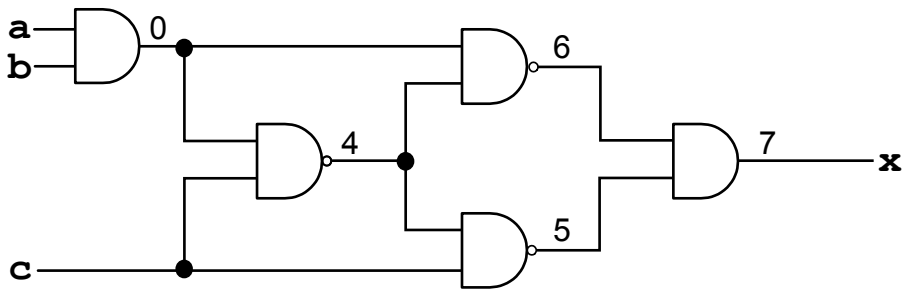
1. Conjoin the specification and implementation,
2. Add the equivalence checking constraint.

$$
\begin{aligned}
&(X \Leftrightarrow (A \wedge \bar{B} \wedge \bar{C}) \vee (\bar{A} \wedge B \wedge \bar{C}) \vee (\bar{A} \wedge \bar{B} \wedge C) \vee (A \wedge B \wedge C)) \wedge \\
&(Y \Leftrightarrow (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)) \wedge \\
&(u \Leftrightarrow (A \wedge \bar{B}) \vee (\bar{A} \wedge B)) \wedge \\
&(v \Leftrightarrow u \wedge C) \wedge \\
&(w \Leftrightarrow A \wedge B) \wedge \\
&(X' \Leftrightarrow (u \wedge \bar{C}) \vee (\bar{u} \wedge C)) \wedge \\
&(Y' \Leftrightarrow w \vee v) \wedge \\
&((X \oplus X') \vee (Y \oplus Y')).
\end{aligned}
$$

This is called a **miter** formula. If unsatisfiable, the specification and implementation are equivalent. A SAT solver can be used to solve this problem.

# Are These Circuits Equivalent?

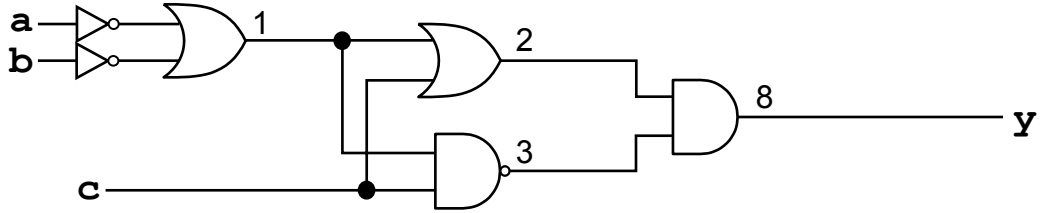
## Circuit #1




**Key to Circuit Symbols**

- *n* — *AND function, output numbered*
- *n* — *NAND function, output numbered*
- *n* — *OR function, output numbered*
- *Inverter (NOT function)*

## Circuit #2



# Example: Outline

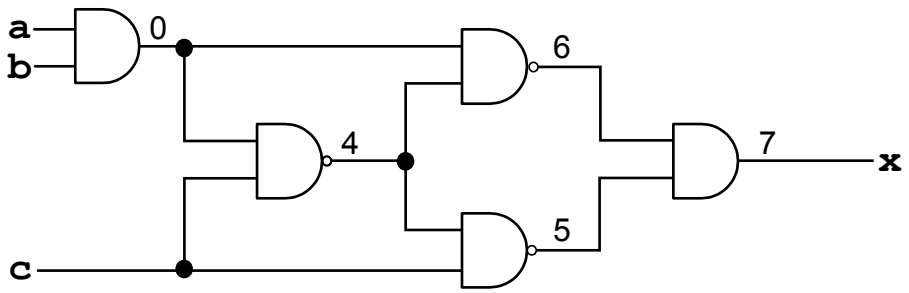
- Random Simulation -
  - Send random vectors through the two circuits, collecting pairs of candidate equivalent nodes
- And/Inverter Graph
  - Find more equivalent nodes by creating the AIG of the circuits
- SAT Sweeping
  - Use candidate equivalent nodes to guide SAT searches, merging AIG nodes which reduces the complexity of future SAT searches

# Identifying Shared Structure

- An internal node in the first circuit may be equivalent to an internal node in the second circuit
- Detect by using random simulation
  - Percolate random vectors through both circuits (fast trick - use 64-bit words)
  - Partition nodes into equivalence classes
  - This can detect potentially many, high probability, candidate equivalent nodes

# Are These Circuits Equivalent?

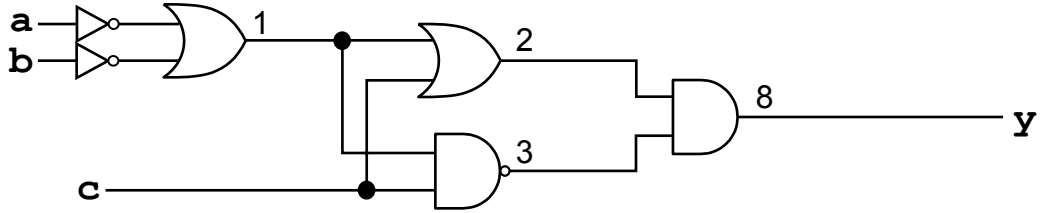
## Circuit #1




**Key to Circuit Symbols**

- *n* — *AND function, output numbered*
- *n* — *NAND function, output numbered*
- *n* — *OR function, output numbered*
- *Inverter (NOT function)*

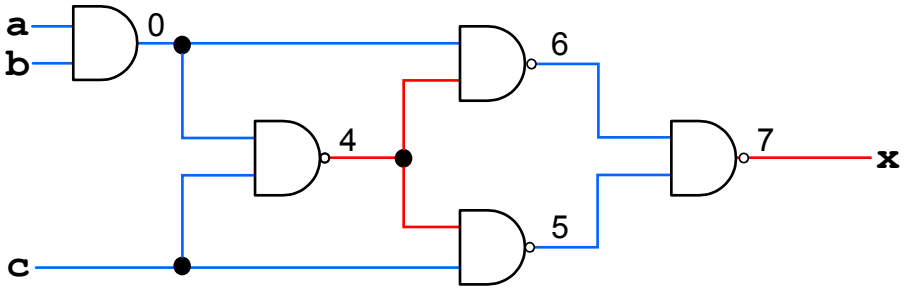
## Circuit #2



# Random Simulation

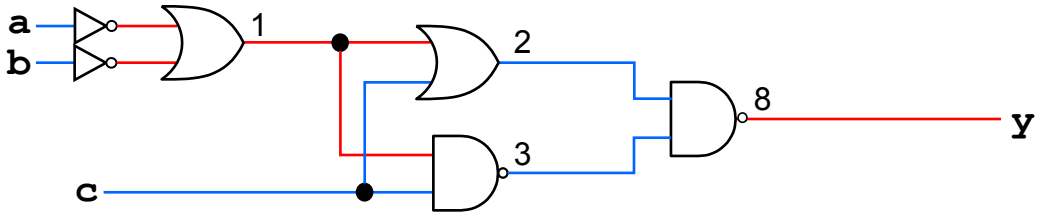
1. Random Vector: {a=**True**, b=**True**, c=**True**}

Circuit #1





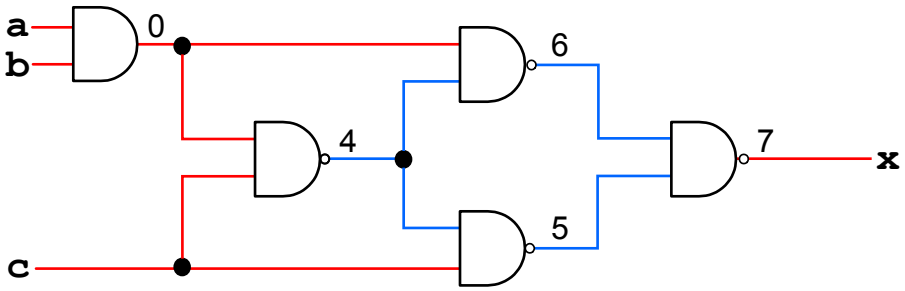
Circuit #2




# Random Simulation

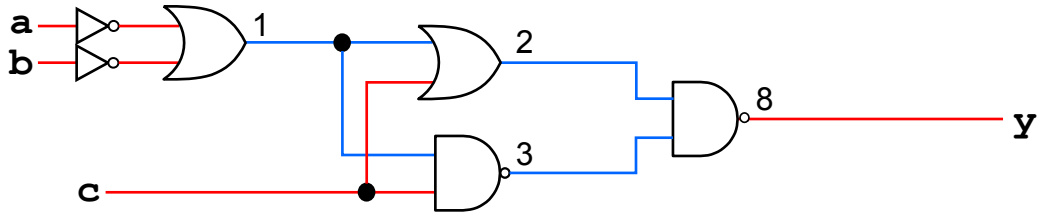
2. Random Vector: {a=**False**, b=**False**, c=**False**}

## Circuit #1





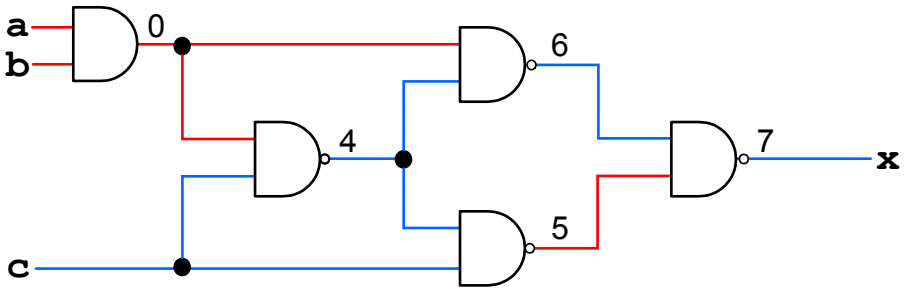
## Circuit #2

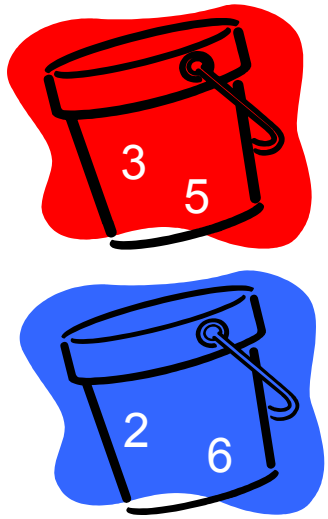



# Random Simulation

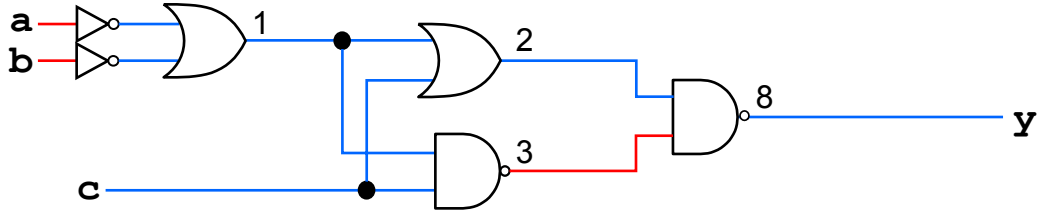
3. Random Vector: {a=**False**, b=**False**, c=**True**}

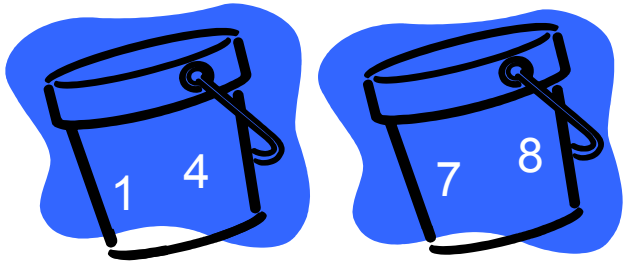
Circuit #1



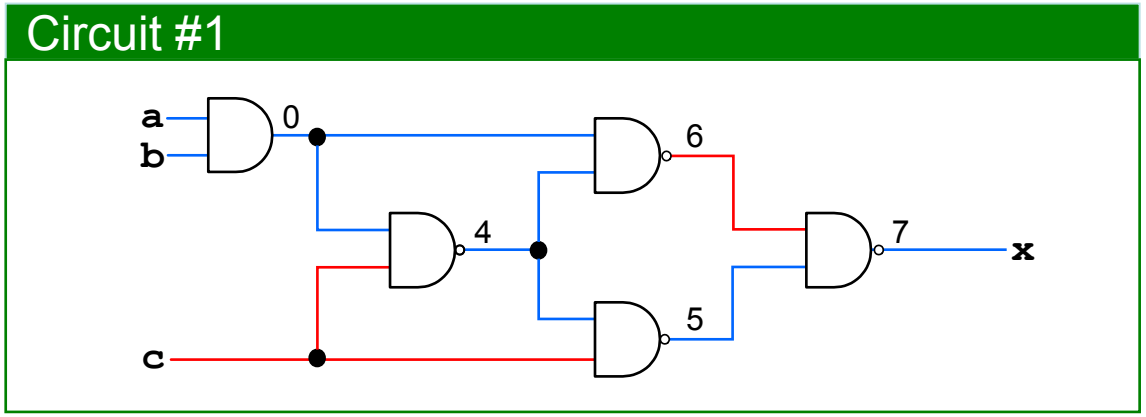

Circuit #2

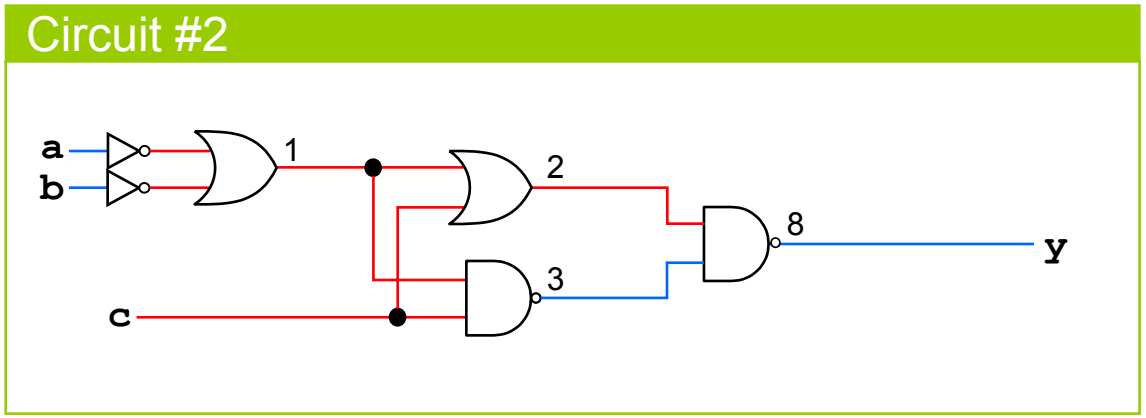


# Random Simulation

4. Random Vector: {a=**True**, b=**True**, c=**False**}








# Example: Outline

- Random Simulation -
  - Send random vectors through the two circuits, collecting pairs of candidate equivalent nodes
- And/Inverter Graph
  - Find more equivalent nodes by creating the AIG of the circuits
- SAT Sweeping
  - Use candidate equivalent nodes to guide SAT searches, merging AIG nodes which reduces the complexity of future SAT searches

# Identifying Shared Structure

- Random simulation is probabilistic
- And/Inverter Graph (AIG) [2]
  - Simple data structure used to represent combinational circuits
  - Nodes represent AND gates
  - Edges represent inputs to an AND gate and may be inverted
  - Operations are fast (add node, merge nodes)

2. A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai. Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. IEEE Trans. CAD, Vol. 21, No. 12, pp. 1377-1394 (2002)

# Model Generation

## Key to Circuit Symbols

*AND function, output numbered*

*NAND function, output numbered*

*OR function, output numbered*


*Inverter (NOT function)*

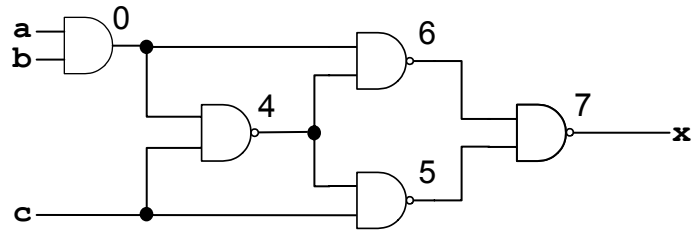
## Key to AIG symbols

*an AND node, numbered as in the circuit*

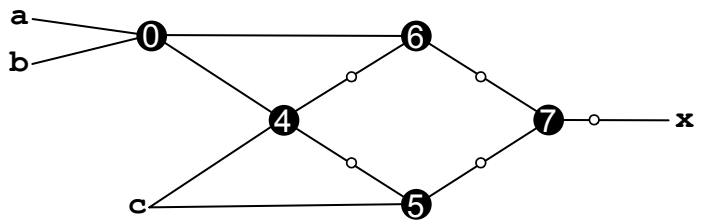

*input to or output from a node*

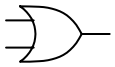
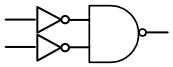

*inverted input or output*

### Circuit #1

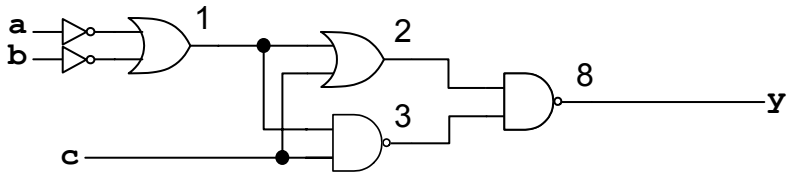



### AIG for Circuit #1

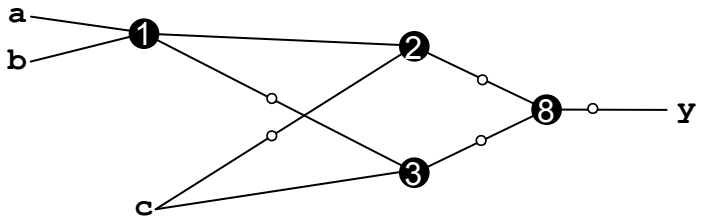



*Recall that*  *is equivalent to* 

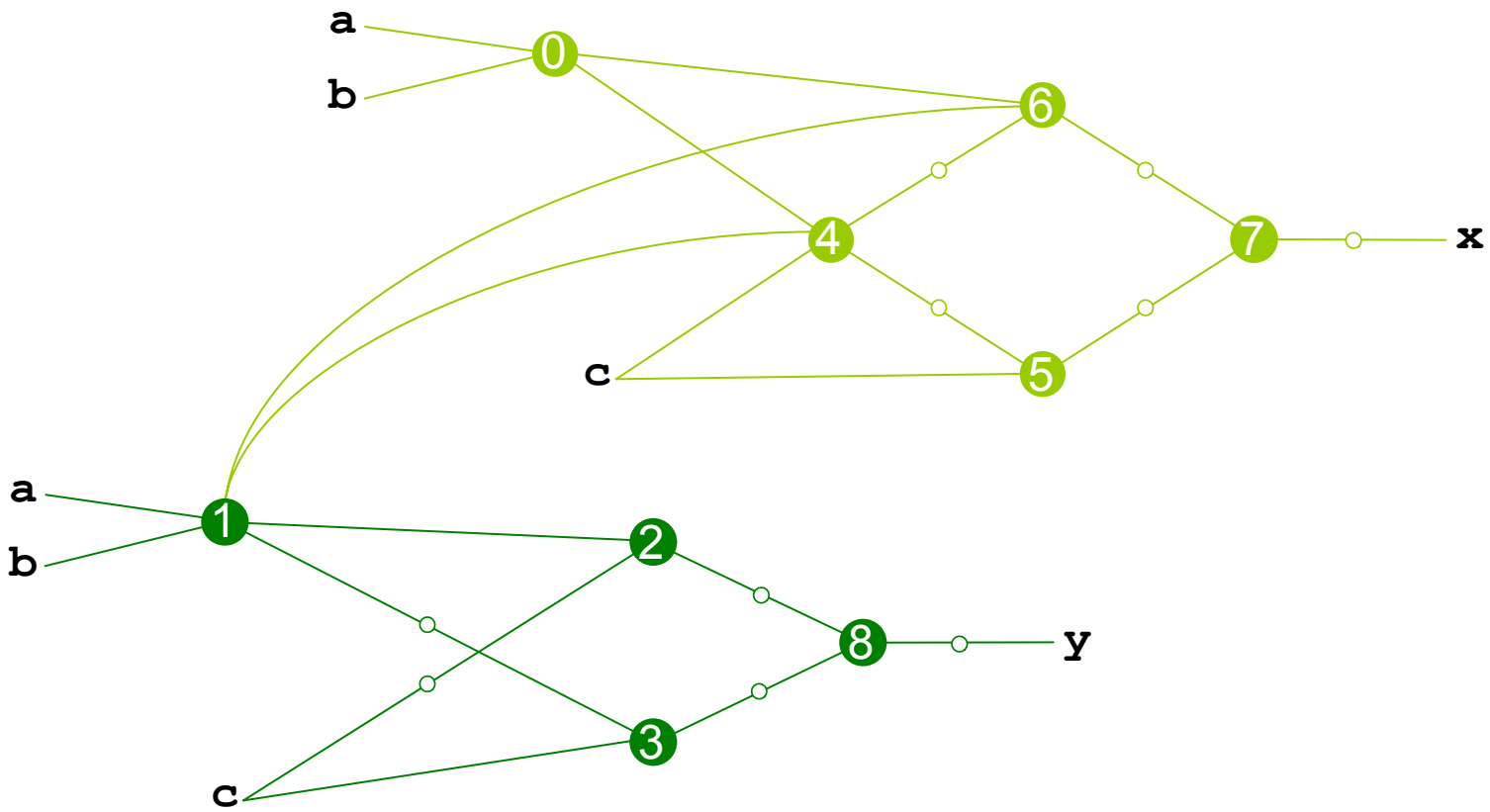
### Circuit #2



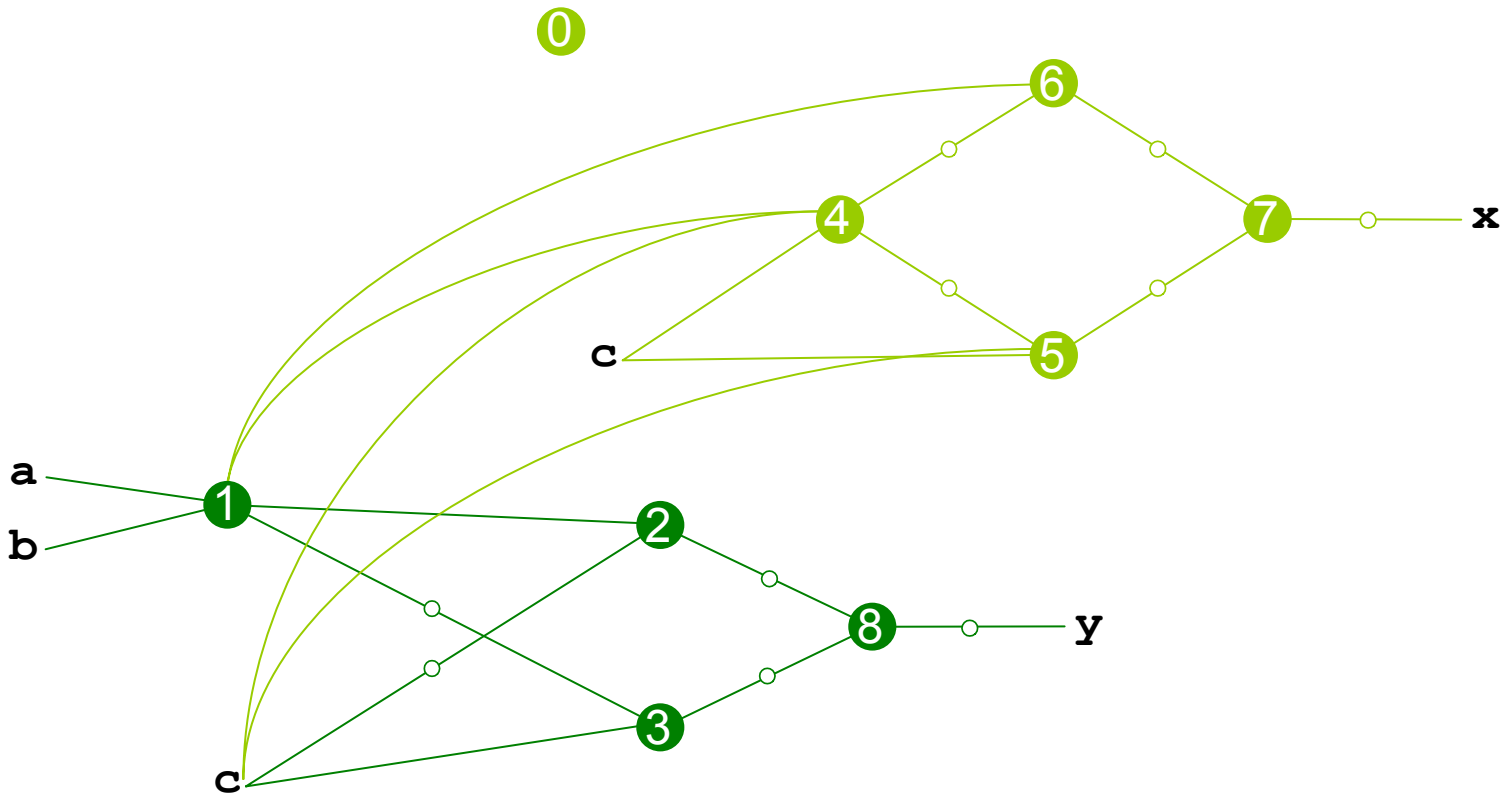

### AIG for Circuit #2



# And/Inverter Graph



# And/Inverter Graph



# Example: Outline

- Random Simulation -
  - Send random vectors through the two circuits, collecting pairs of candidate equivalent nodes
- And/Inverter Graph
  - Find more equivalent nodes by creating the AIG of the circuits
- SAT Sweeping
  - Use candidate equivalent nodes to guide SAT searches, merging AIG nodes which reduces the complexity of future SAT searches

# SAT Sweeping

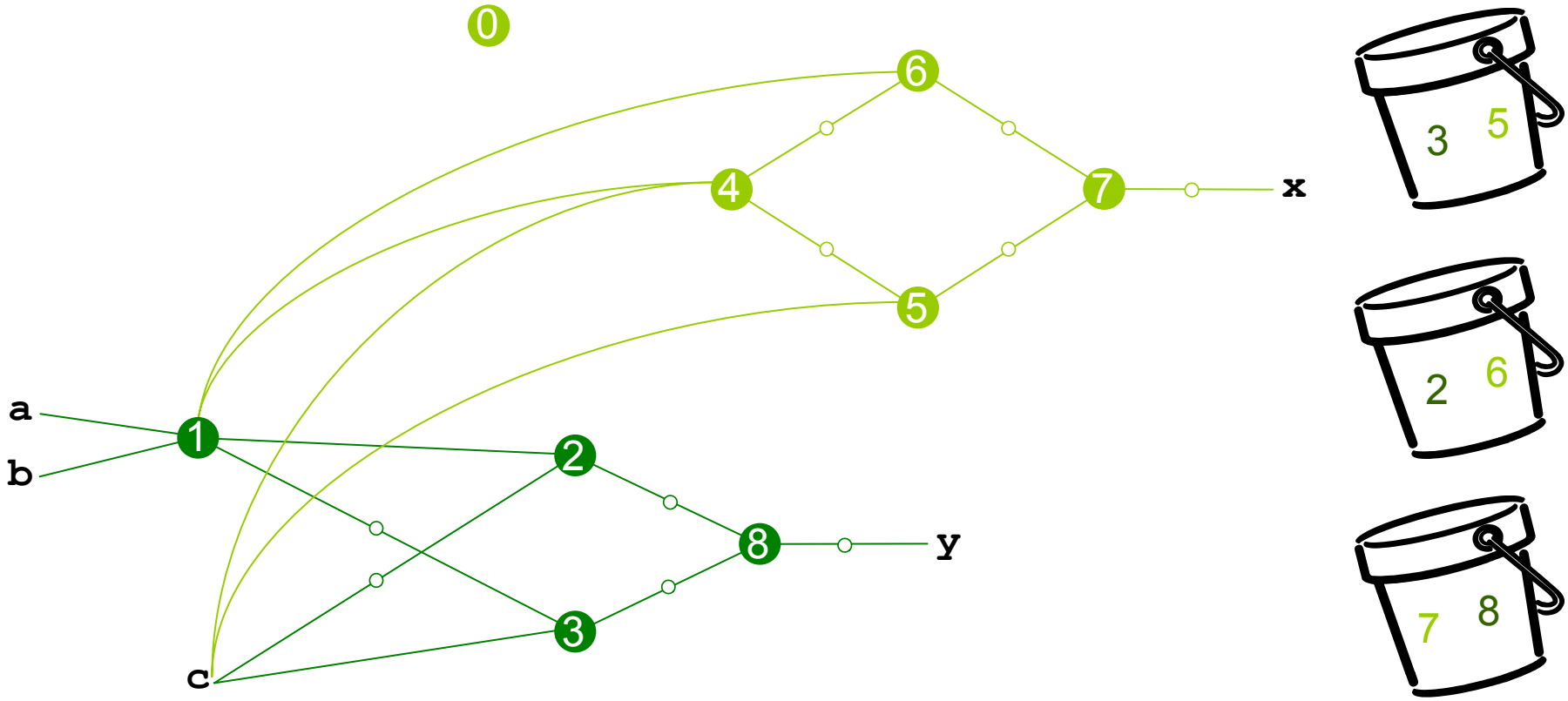
- Use SAT to prove whether or not the candidate equivalent nodes, from the random simulation phase, are equivalent
- Generate SAT problems that are solved from inputs to outputs using the candidate equivalent nodes as a guide [3]

3. A. Kuehlmann. Dynamic Transition Relation Simplification for Bounded Property Checking. In ICCAD, 2004

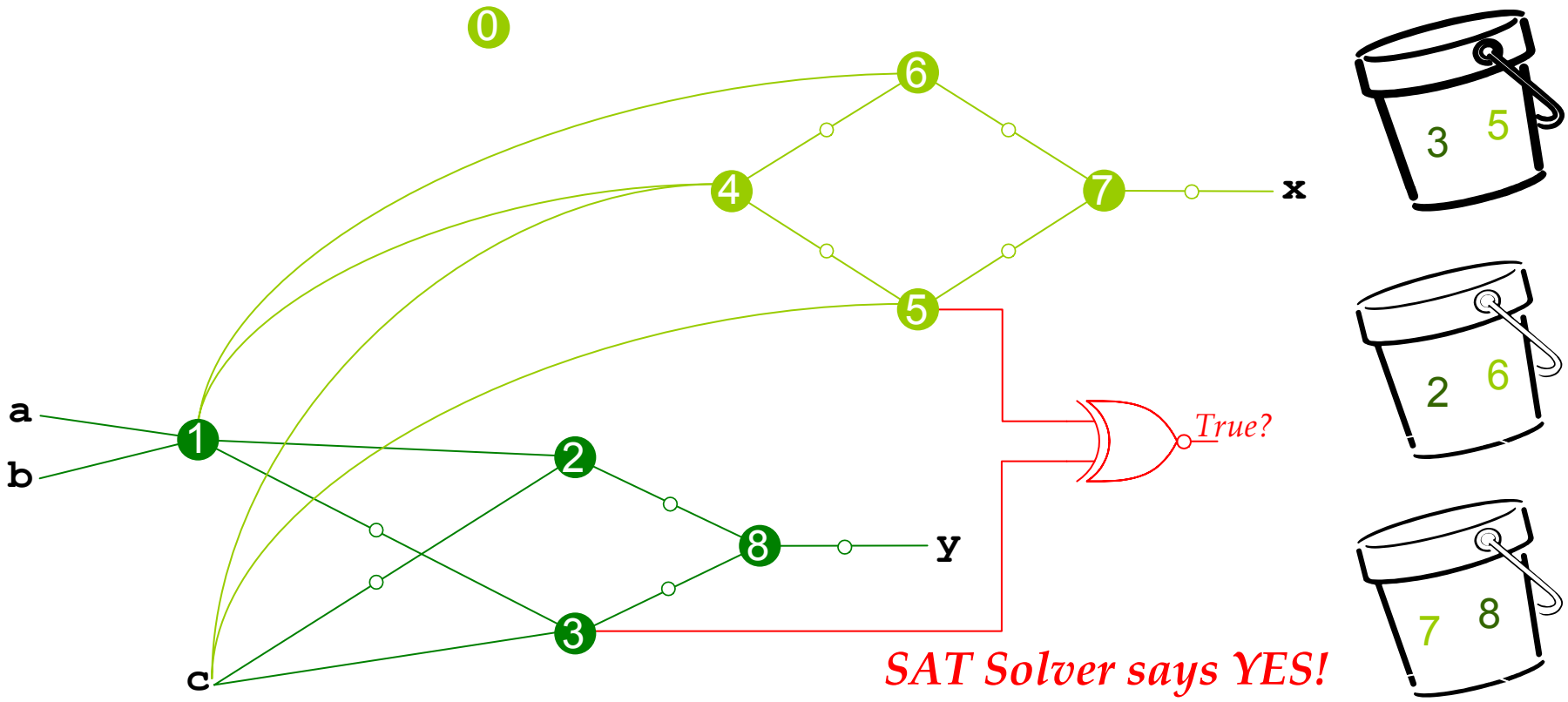
# SAT Instances

- Create one SAT instance for each pair of candidate equivalent nodes
  - A SAT instance encodes a miter circuit
- Each SAT search can result in many mergers of equivalent AIG nodes, reducing the complexity of the AIG

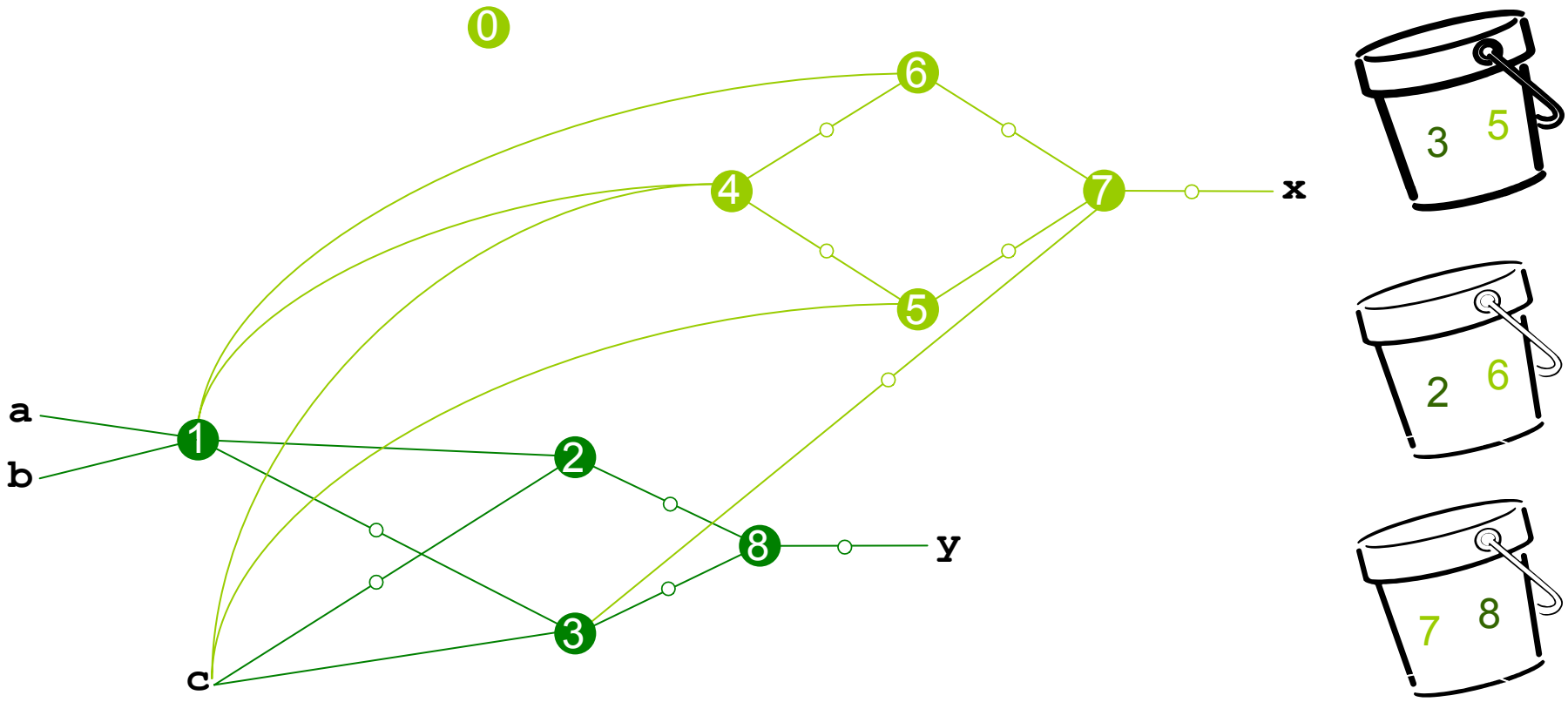
# SAT Sweeping



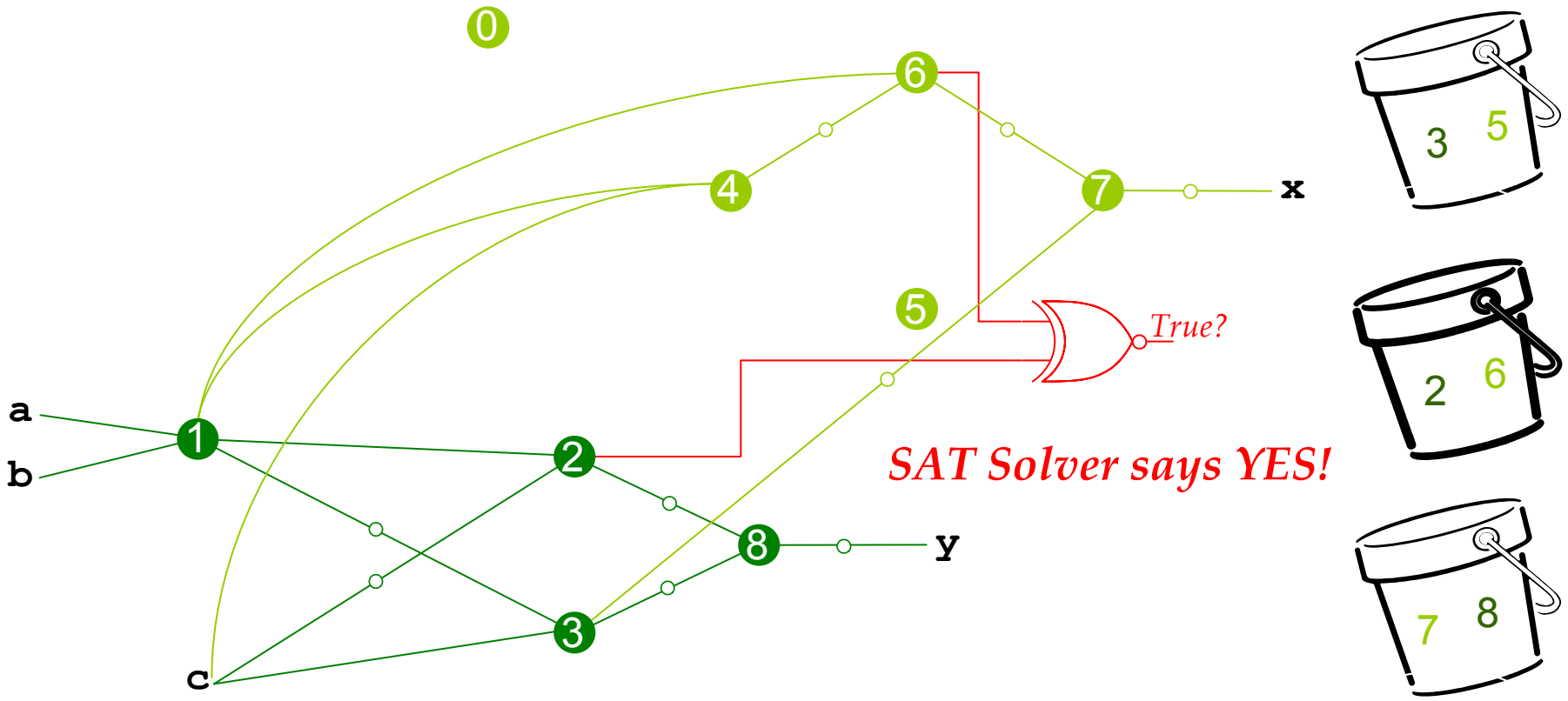
# Does 3 = 5?



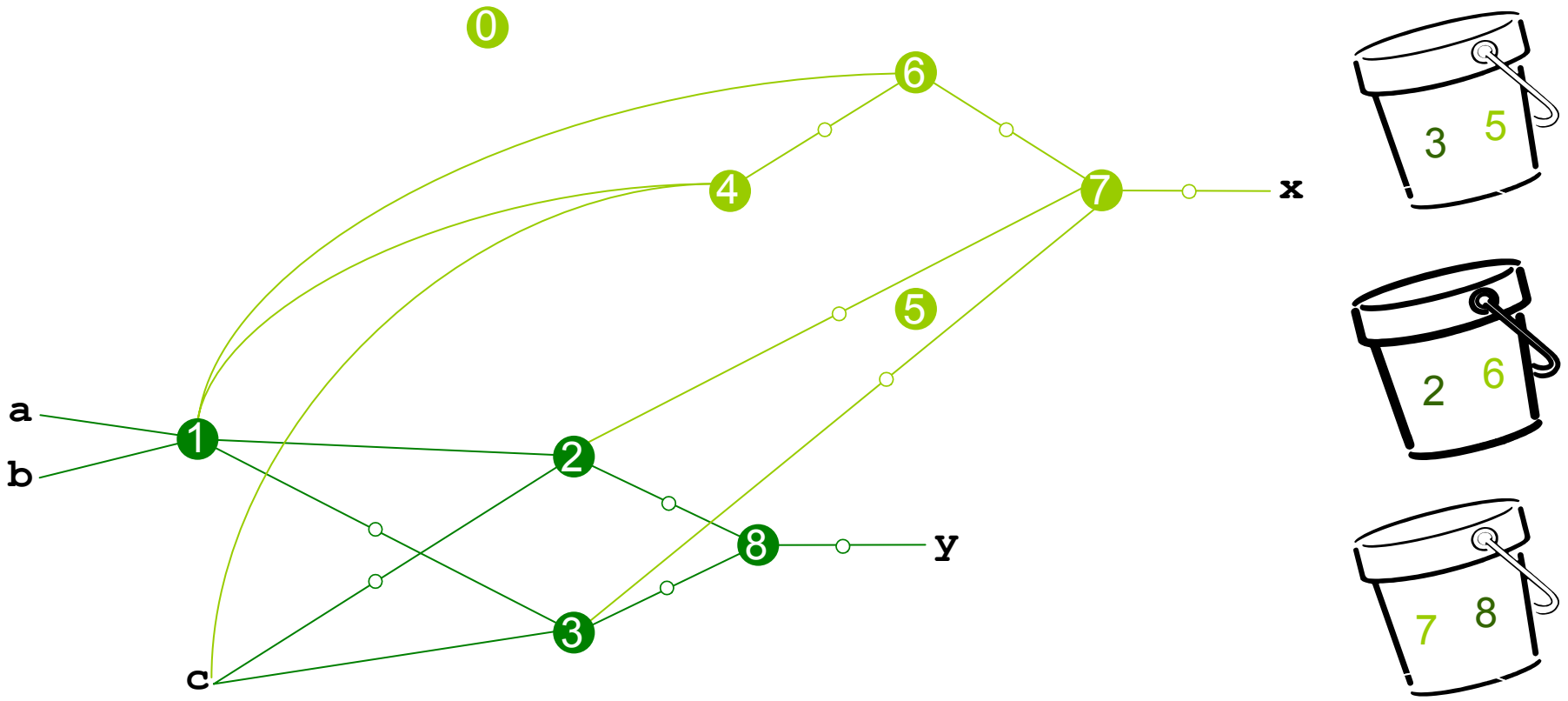
# Merge nodes 3 and 5



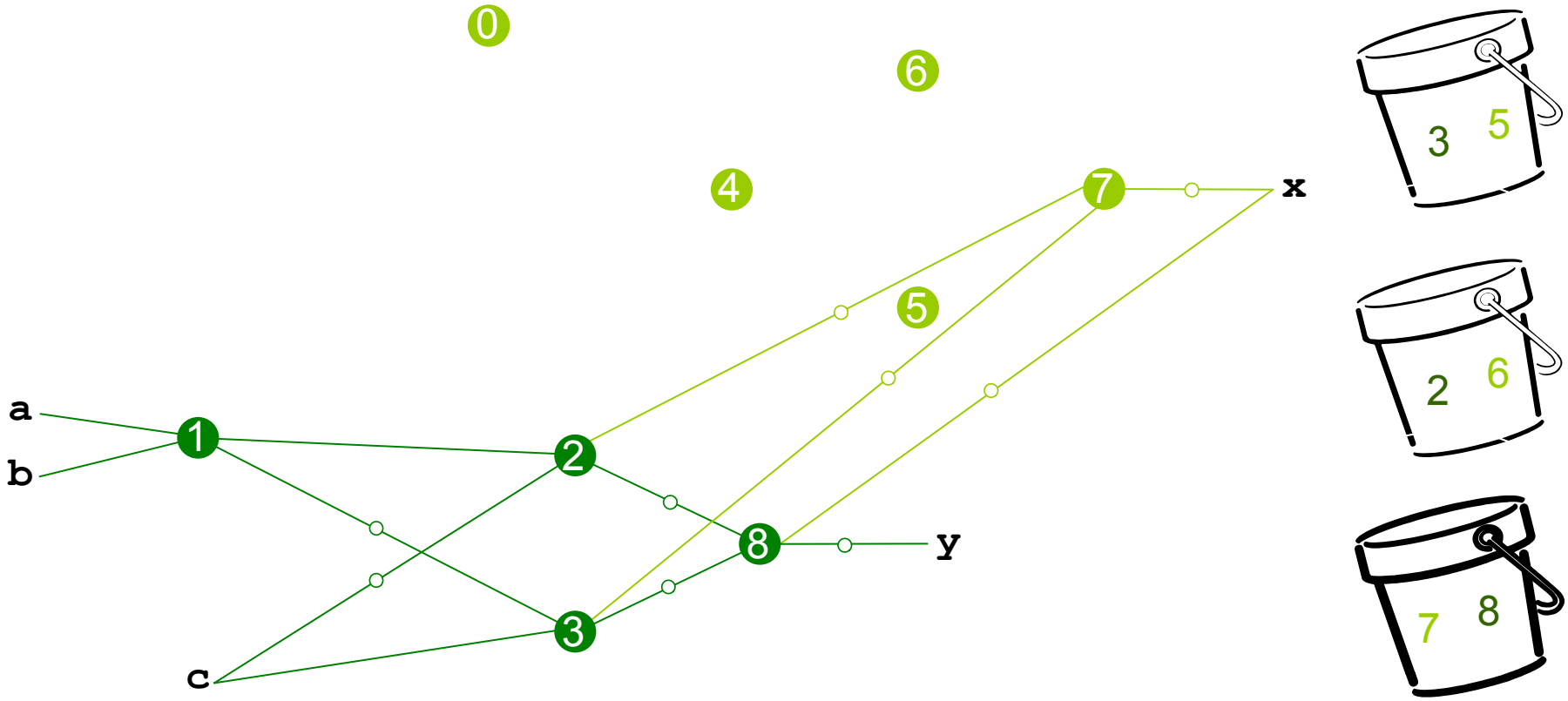
# Does 2 = 6?



# Merge nodes 2 and 6



# Structural hashing gives 7=8



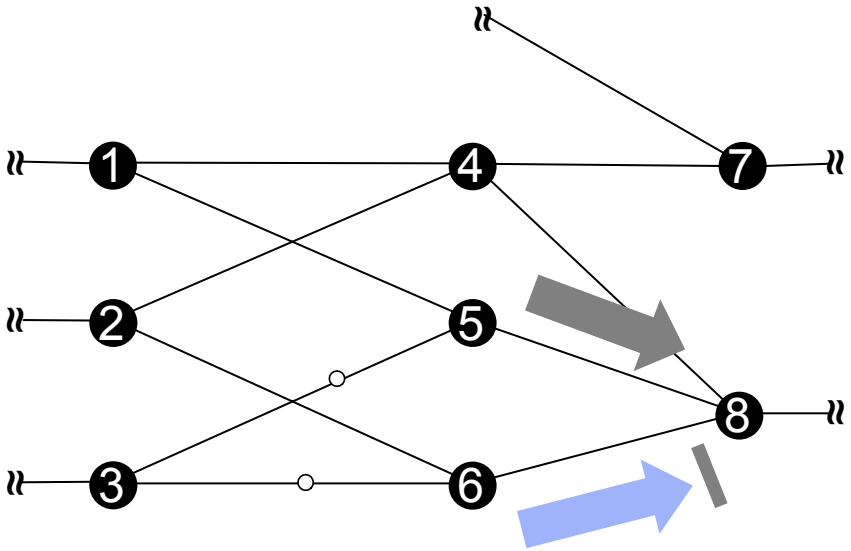
# Example: Outline

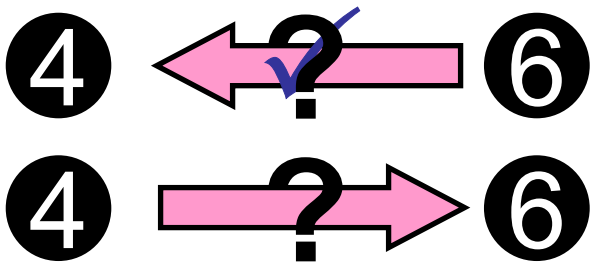
- Random Simulation -
  - Send random vectors through the two circuits, collecting pairs of candidate equivalent nodes
- And/Inverter Graph
  - Find more equivalent nodes by creating the AIG of the circuits
- SAT Sweeping
  - Use candidate equivalent nodes to guide SAT searches, merging AIG nodes which reduces the complexity of future SAT searches

# Advanced Techniques

- SAT sweeping is now about 5 years old
- Newer techniques are being incorporated into equivalence checkers
  - Cut Sweeping
  - Specialized Heuristics
  - Local Observability Don't-Cares
  - Higher level representations (more than AND/Inverter gates)
  - Integration with Theorem Provers / SMT solvers
  - …many more!

# Local Observability Don't-Cares




4

| $x_1$ \ $x_2x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |

6

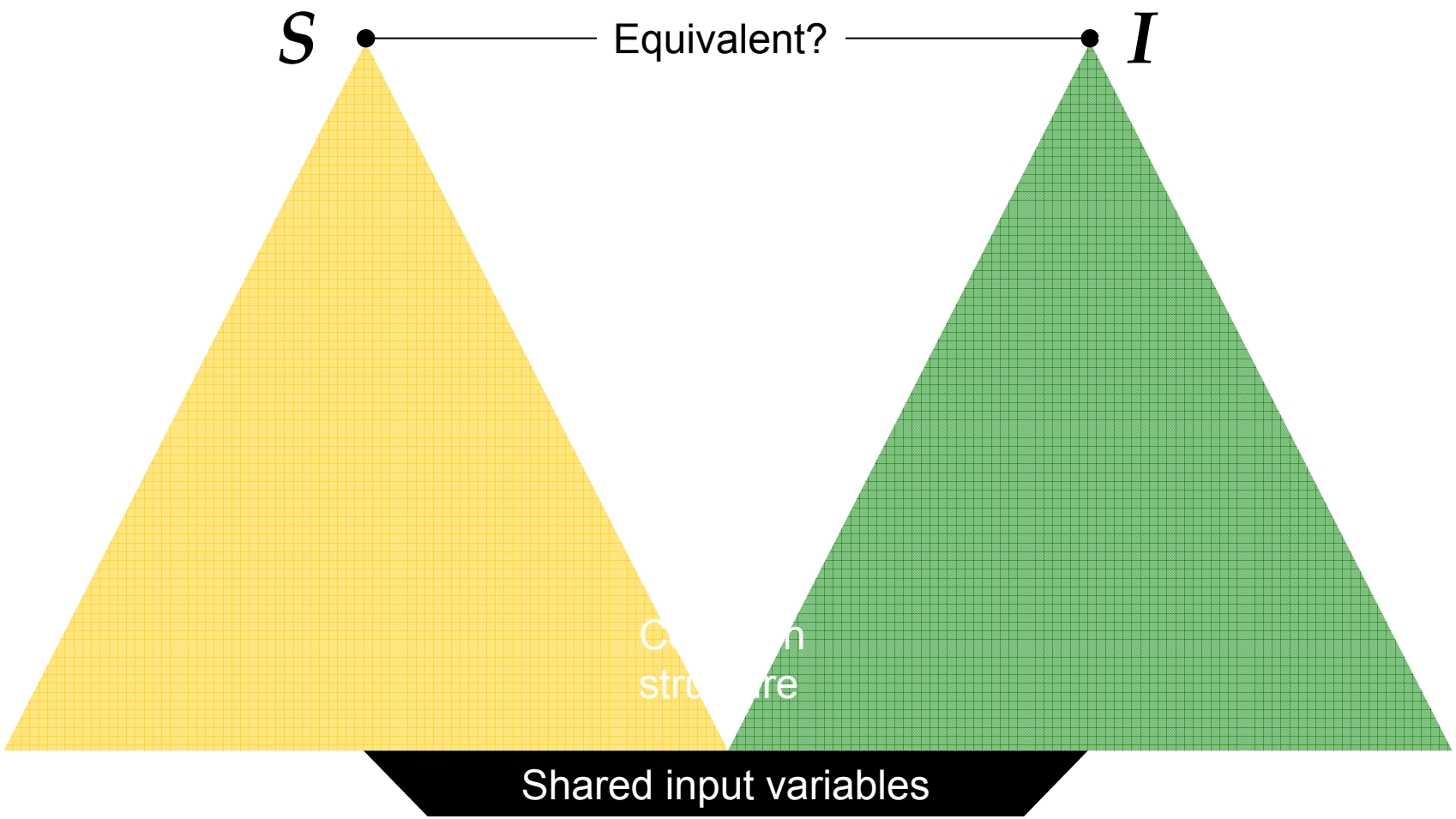
| $x_1$ \ $x_2x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 |

5

| $x_1$ \ $x_2x_3$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 |

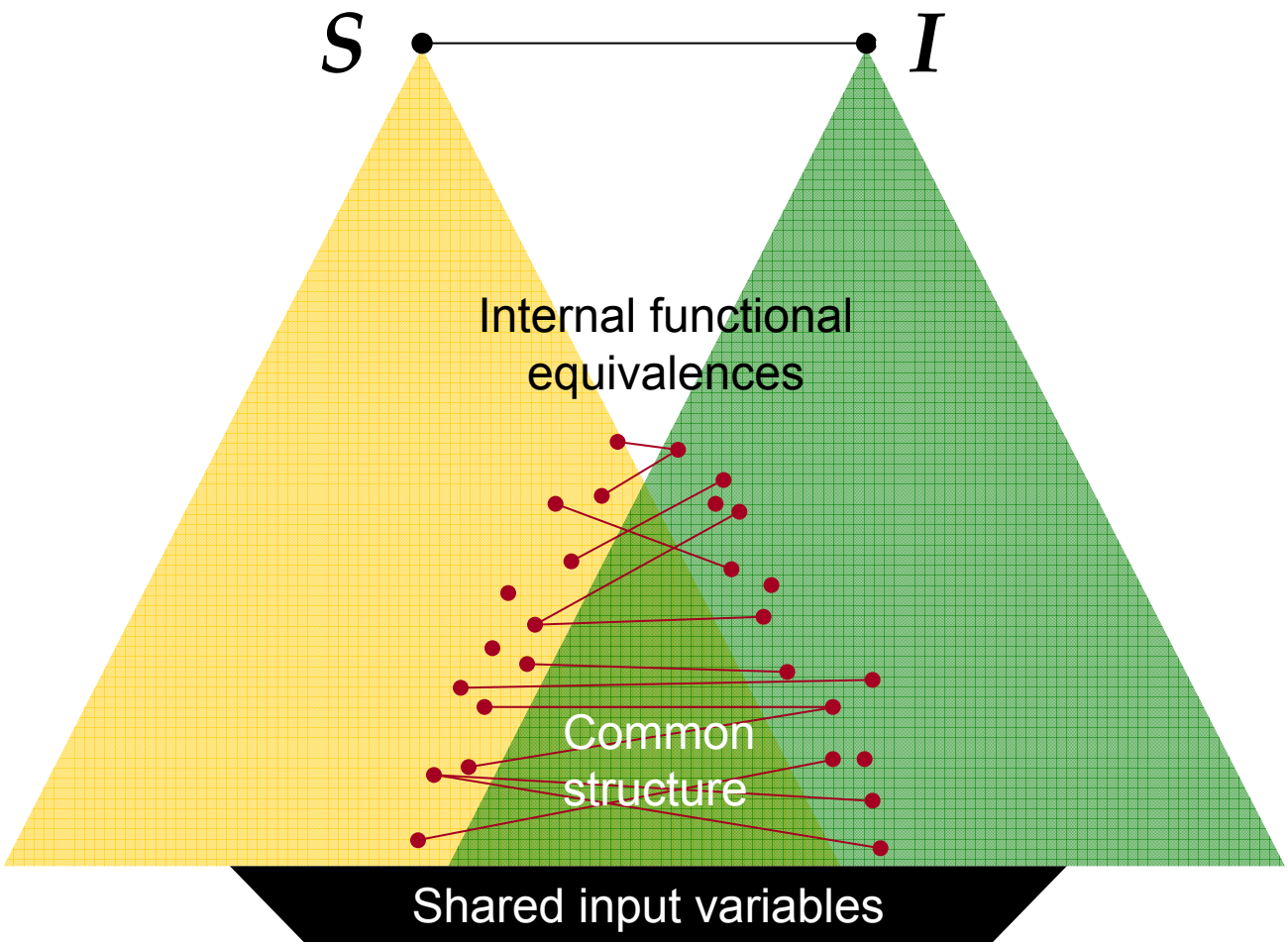
From N. Kitchen. "SAT Sweeping with Local Observability Don't-Cares."

# Equivalence Checking in the Large:



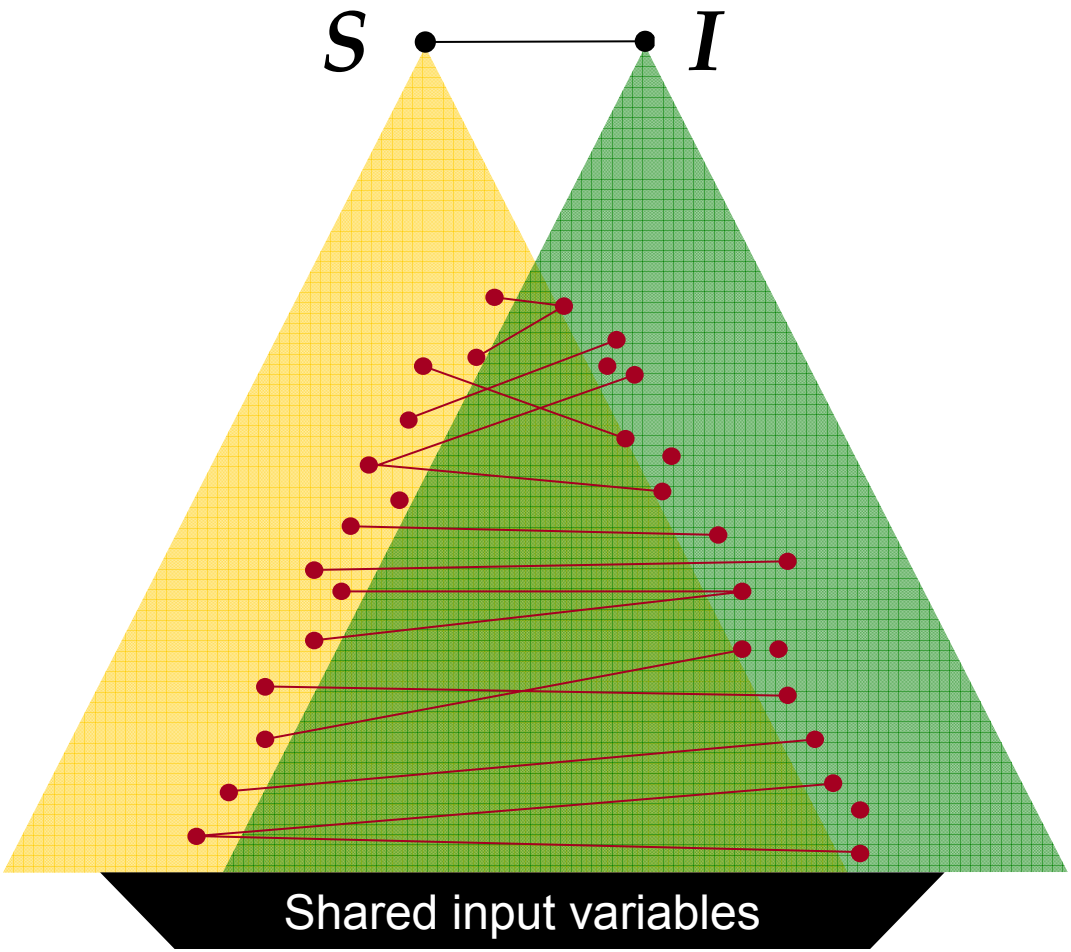

from A. Biere, “SAT in Formal Hardware Verification”

# Equivalence Checking in the Large:




from A. Biere, “SAT in Formal Hardware Verification”

# Equivalence Checking in the Large:



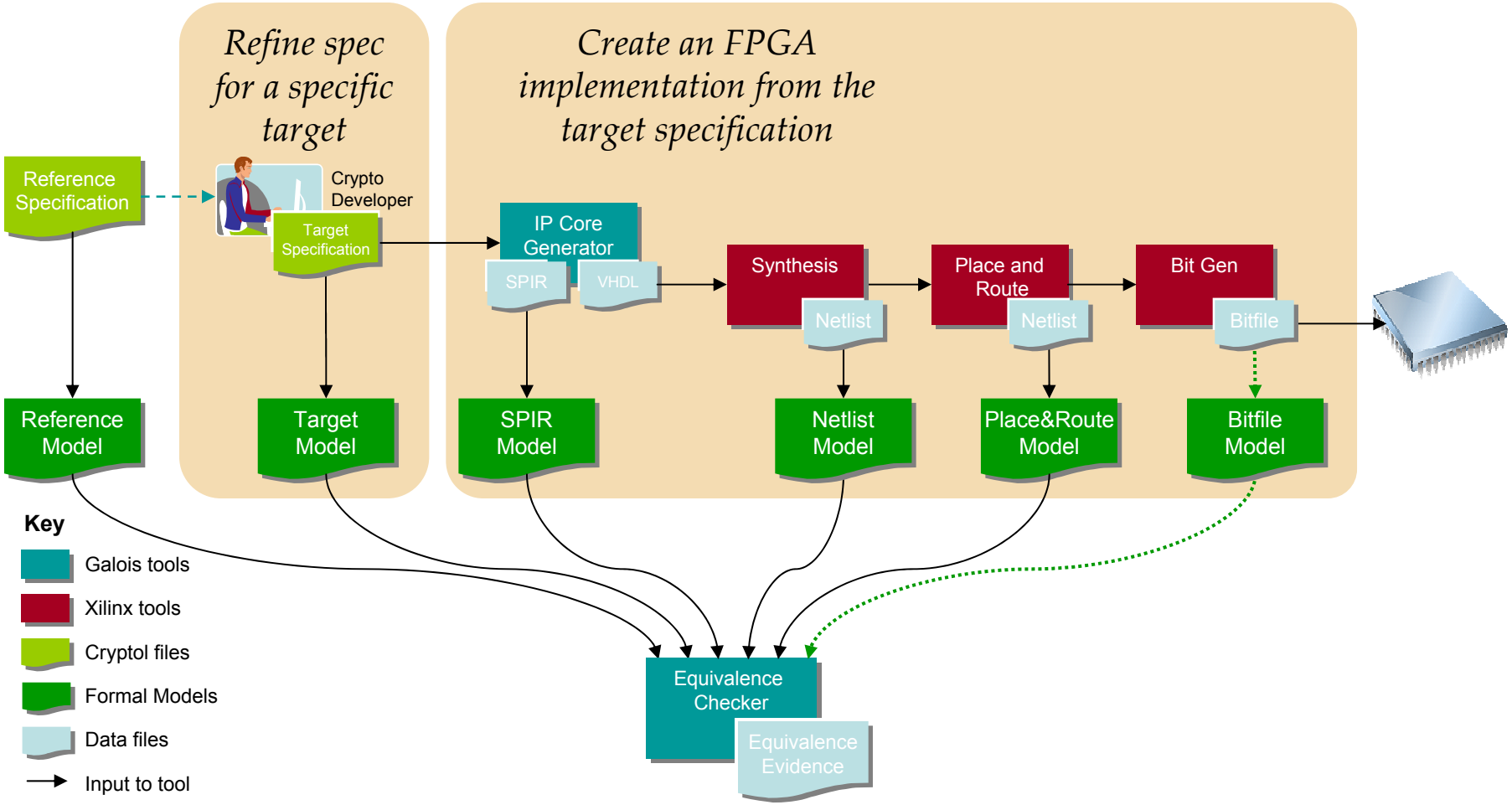

from A. Biere, “SAT in Formal Hardware Verification”

# Cryptol

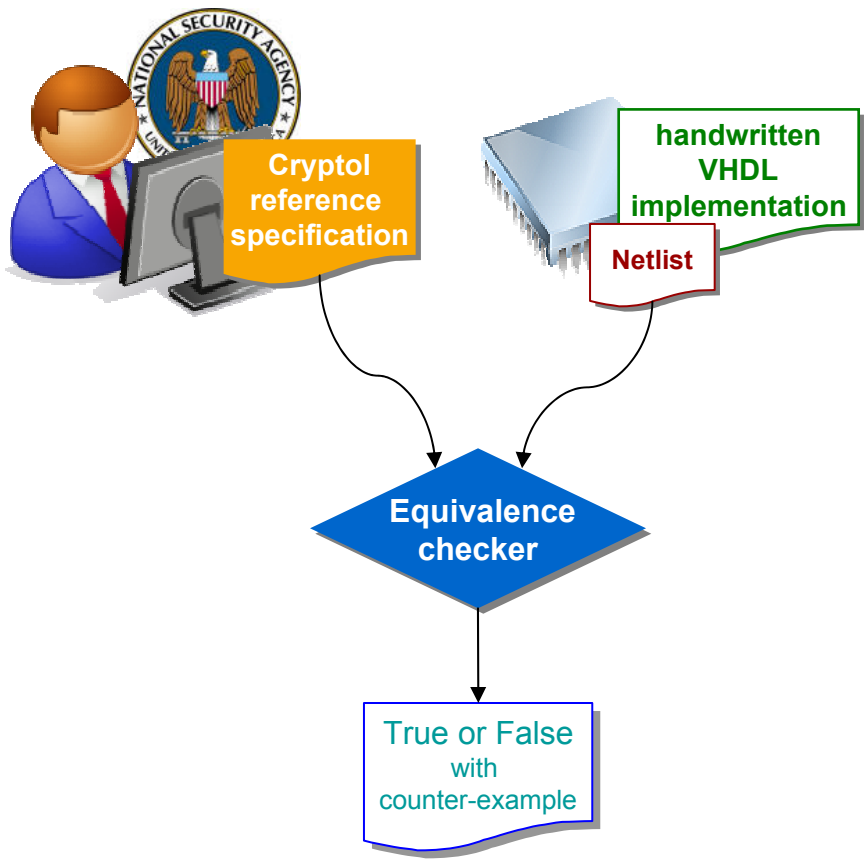
- Cryptol is a “Haskell-esk” functional cryptographic specification language
- Created by Galois Inc. with support from NSA cryptographers
- Cryptol specifications can be transformed into AIGs
  - Built in equivalence checker - jaig
  - also provides support for ABC (UC Berkeley)

# Tool Chain Verification Strategy

*Show function equivalence with the reference specification throughout the tool chain*



# Typical Cryptol use in an IAD evaluation




A Crypto device evaluator:

- Creates a reference specification and associated AIG
- Generates an AIG for the evaluation VHDL
- Uses an equivalence checker to asses the correctness of the VHDL

# Some Results

- NIST Hash Competition (Skein)
  - Men Long (Intel)
  - Stefan Tillich (TU Graz)
- NIST AES Competition
  - Reference C
  - Optimized C
- Van der Waerden Numbers

# NIST Hash Competition

- “NIST has opened a public competition to develop a new cryptographic hash algorithm, which converts a variable length message into a short “message digest” that can be used for digital signatures, message authentication and other applications.”
- 51 submissions
- Thanks to Frank Taylor and Galois, many of these have Cryptol specs.
- Galois has received requests to verify VHDL implementations for some
- We’ll look at their recent Skein verification

http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

# Verification Process

- Develop a specification
- Understand the implementation
- Coerce the type signature of the implementation and specification
- Use Cryptol to generate AIGs for both the implementation and specification
- Call the equivalence checker

# Develop a Specification

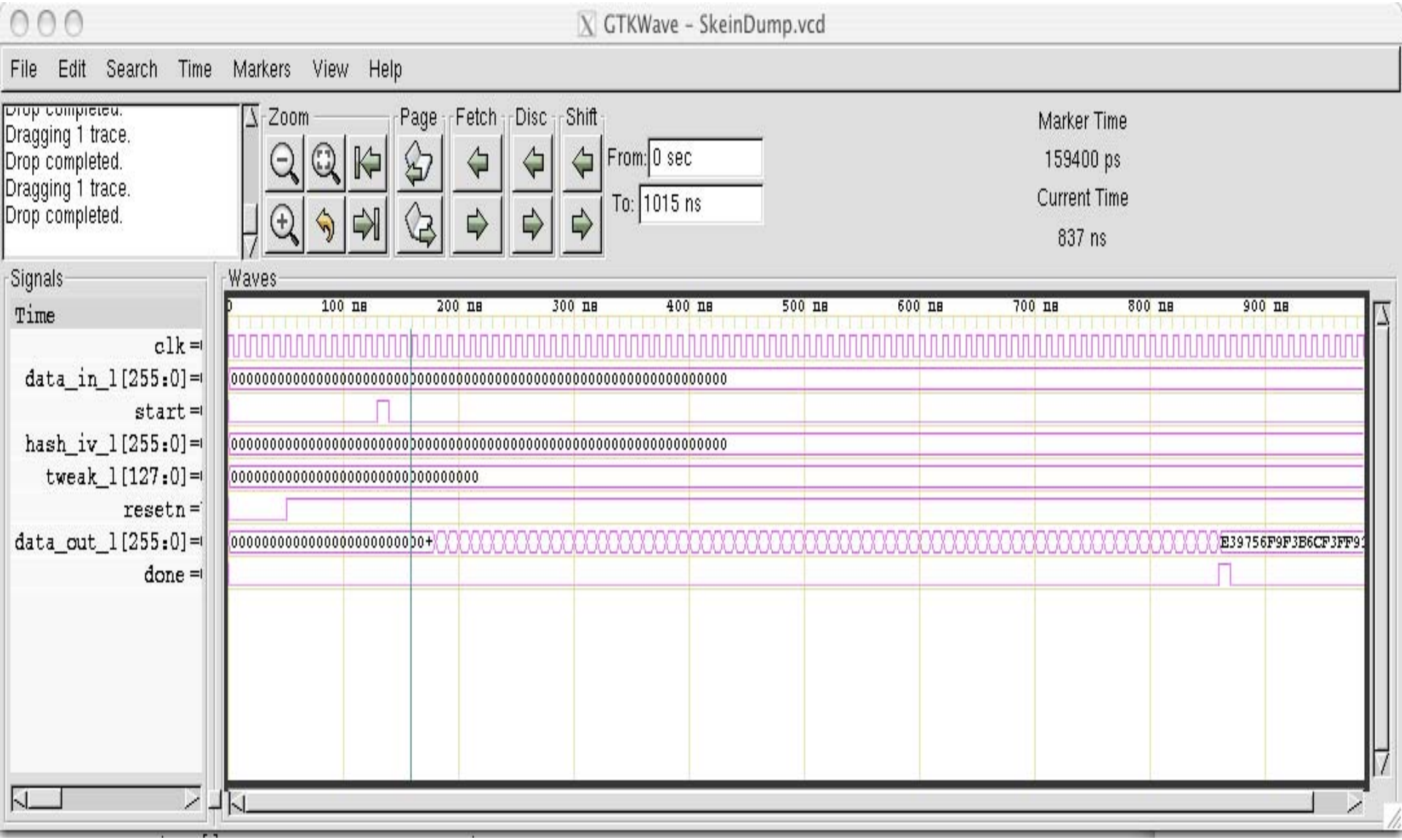
```
encrypt256 : ([32][8],[16][8],[32][8]) -> [4][64];
encrypt256 (key,tweak,pt) = vn + kn
  where {
   // Threefish-256 has 72 rounds:
   nr  = 72;
   nw  = 4;
   …
   key_words : [4][64];
   key_words = split(join key);
   tw_words : [2][64];
   tw_words = split(join tweak);
   pt_words : [nw][64];
   pt_words = split(join pt);
  };
```

http://www.galois.com/blog/category/cryptography/

# Verification Process

- Develop a specification
- Understand the implementation
- Coerce the type signature of the implementation and specification
- Use Cryptol to generate AIGs for both the implementation and specification
- Call the equivalence checker

# Understanding the Implementation



# Understanding the Implementation

```
extern vhdl("datatype.vhd”, "skein_mixcolumn.vhd", "skein_round.vhd”,
            "skein_shiftrow.vhd“, "skein_add_round_key.vhd”, "skein.vhd“,
             skein, clock=clk, reset=resetn, invertreset)
extern_menLong : [inf](start:Bit, data_in_L:[256], hash_iv_L:[256],
                       tweak_L:[128]) ->
                 [inf](done:Bit, data_out_L:[256]);

menLongVHDL : [256] -> [256];
menLongVHDL inp = res
  where { wait     = (False, inp, zero, zero);
          start    = (True, inp, zero, zero);
          rest     = [wait] # rest;
          (_, res) = extern_menLong([wait start] # rest) @ 74;
        };
```

# Verification Process

- Develop a specification
- Understand the implementation
- Coerce the type signature of the implementation and specification
- Use Cryptol to generate AIGs for both the implementation and specification
- Call the equivalence checker

# Coerce the Type Signatures

```
menLongRef : [256] -> [256];
menLongRef inp =
  alignOut(encrypt256(zero, zero, reverse (split inp))) ^ inp
  where { alignOut : [4][64] -> [256];
          alignOut xs = join (reverse (split (join xs) : [32][8]));
        };


menLongVHDL : [256] -> [256];
menLongVHDL inp = res
  where { wait     = (False, inp, zero, zero);
          start    = (True, inp, zero, zero);
          rest     = [wait] # rest;
          (_, res) = extern_menLong([wait start] # rest) @ 74;
        };
```

# Verification Process

- Develop a specification
- Understand the implementation
- Coerce the type signature of the implementation and specification
- Use Cryptol to generate AIGs for both the implementation and specification
- Call the equivalence checker

# Men Long Equivalence Check

- VHDL Implementation of the Skein UBI Block
- Skein UBI Block AIG Sizes
  - Cryptol Reference, 118156 nodes
  - Men Long, 653963 nodes
- Used ABC (UC Berkeley) Equivalence Checker
- Time: ~1h
- VHDL code is equivalent to Cryptol spec.

# Stefan Tillich Equivalence Check

- Full Skein VHDL Implementation
- Skein AIG Sizes (256 bits input/output)
  - Cryptol Reference, 301342 nodes
  - Stefan Tillich, 900496 nodes
- Used ABC (UC Berkeley) Equivalence Checker
- Time: ~17.5h
- VHDL code is equivalent to Cryptol spec.

http://www.iaik.tugraz.at/content/research/

# NIST AES Competition

- “AES was announced by National Institute of Standards and Technology (NIST) as U.S. FIPS PUB 197 (FIPS 197) on November 26, 2001 after a 5-year standardization process in which fifteen competing designs were presented and evaluated before Rijndael was selected as the most suitable”
- “AES is the first publicly accessible and open cipher approved by the NSA for top secret information”

http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

# AES Verification

- Two C programs are provided by NIST that are the official AES specifications.
  - One is for reference, and one is optimized for speed.
- We have a Cryptol specification
- We also computer generated a C version from the Cryptol spec.

# AES Results

| Source | AIG Nodes | Runtime (minutes) |
|---|---|---|
| Generated C | 939087 | 3 |
| NIST Optimized C | 1482247 | 7.5 |
| NIST Reference C | 2397319 | 29 |

- Results of experiments verifying various AES implementations against a Cryptol AES specification (934831 nodes)
- Cryptol's jaig equivalence checker was used

# Van der Waerden Numbers

- Van der Waerden numbers W(k,r)=n
  - Smallest n such that every partition of {1,…,n} into k blocks results in an arithmetic progression of length r in some block
  - New assertions by Michal Kouril
    - W(2,6) = 1132
    - W(3,4) = 293
  - This is quite a feat because now only 7 numbers are known and no new ones had been found since 1979

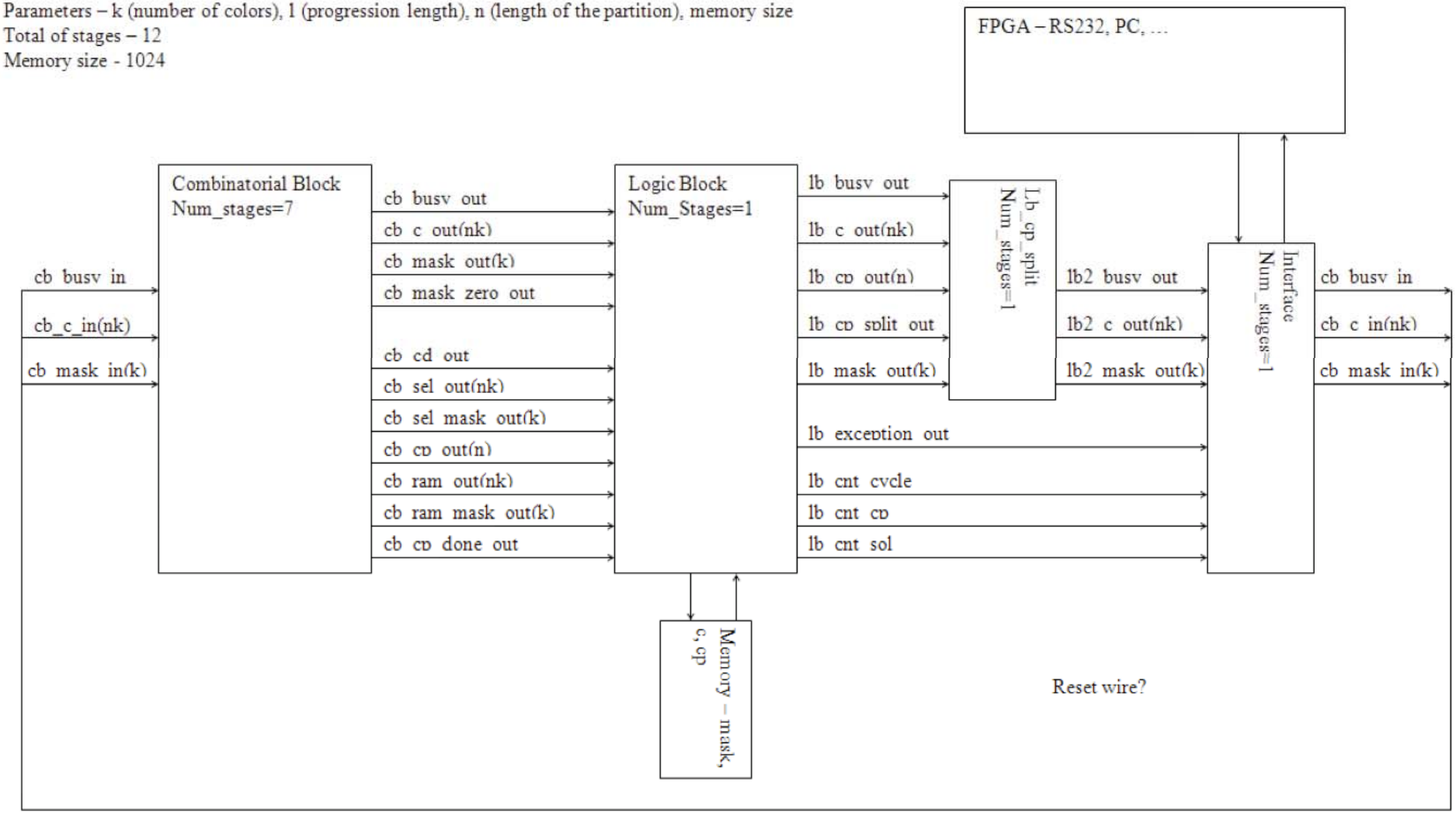
# High Performance Computing

- Michal's solver is written in VHDL, runs on a cluster of FPGAs at UC
- The solver performs backtracking search (specialized SAT solver) and exhausts the search space
- Both results took months of runtime
- How do we gain confidence that the solver is correct?
- Use equivalence checking!

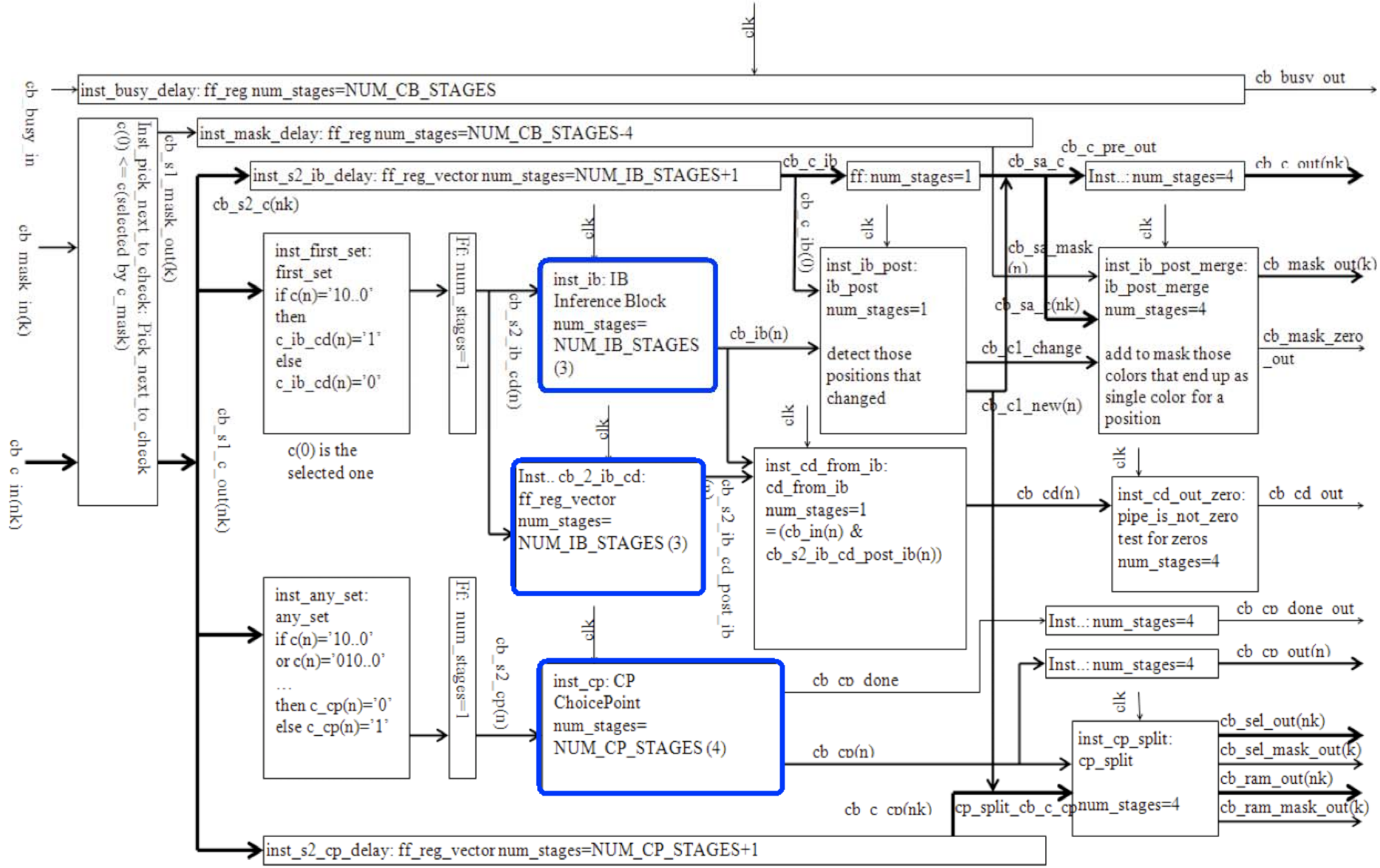
# High Level Structure

**FPGA based SAT solver for Van der Waerden numbers**

Parameters – k (number of colors), l (progression length), n (length of the partition), memory size
Total of stages – 12
Memory size - 1024



# Combinatorial Block



# Results VdW

- Wrote Cryptol specifications for the three main VHDL entities
- Generated AIGs for the Cryptol spec. and VHDL entities.
- Used the Cryptol's equivalence checker 'jaig' to verify the VHDL entities
  - Each entity has $2^{240}$ possible inputs
  - Total time for all three checks < 30 minutes

# CD Block Cryptol Code

```
n=240;
L = 6;

clause : ([n*2+1][1], [width n], [width n]) -> Bit;
clause(x, start, step) =
  if ((n-start-1)/step>=(L-1))
  then [|x@(start+(p-1)*step)||p<-[1..L]|]==[|1||p<-[1..L]|]
  else False;

cd : ([n][1]) -> [1];
cd(x) =
  if[|if clause(x#[|0||p<-[1..n+1]|], start-1, step) then 1 else 0
     ||start<-[1..n-1], step<-[1..n-1]|]==[|0||p<-[1..((n-1)*(n-1))]|]
  then 0
  else 1;
```

```
--
-- cd_block_L
--
-- detect whether there is a progression length L with step=1 in the given size n
-- array
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.vdw_vhdl.all;

Entity cd_block_L is
generic (n : natural := vdw_n;
         L : natural := vdw_L);
port (i : in  std_logic_vector(n-1 downto 0);
      o : out std_logic_vector(n-1 downto 0));
end cd_block_L;

architecture behavioral of cd_block_L is
type t_cd_out_tmp_type is array (0 to L-1) of std_logic_vector(n-1-(L-1) downto 0);
signal t_cd_out_tmp : t_cd_out_tmp_type;
begin
       t_cd_out_tmp(0) <= i(n-1-0 downto L-1-0);
my_body1:  for pos in 1 to L-1 generate
             t_cd_out_tmp(pos) <= t_cd_out_tmp(pos-1) and i(n-1-pos downto L-1-pos);
       end generate my_body1;

     o(n-1-(L-1) downto 0) <= t_cd_out_tmp(L-1);

pad: if n-1 >= n-1-(L-1)+1 generate
       o(n-1 downto n-1-(L-1)+1) <= (n-1 downto n-1-(L-1)+1=>'0');
    end generate;
end behavioral;

--
-- cd_block
-- reformat the array length n for the step=1 checker (cd_block_L) given step
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.vdw_vhdl.all;

entity cd_block is
generic (n     : natural := vdw_n;
         L     : natural := vdw_L;
         step  : natural := 1);
port (i  : in  std_logic_vector(n-1 downto 0);
      o  : out std_logic_vector(n-1 downto 0));
end cd_block;

architecture behavioral of cd_block is

component cd_block_L
   generic (n: natural := vdw_n;
            L: natural := vdw_L);
   port (i : in  std_logic_vector(n-1 downto 0);
         o : out std_logic_vector(n-1 downto 0));
end component;

type start_array_type is array (0 to step) of natural;

function get_start(offset: natural)
return natural is
variable pos : natural := 0;
begin
 pos := 0;
 if (offset>0) then
   for pos_start in 0 to offset-1 loop
     for pos_index in 0 to n-1 loop
       if (pos_index*step+pos_start < n) then
         pos:=pos+1;
       end if;
     end loop;
   end loop;
 end if;
 return pos;
end get_start;

function get_start_array(max_step: natural)
return start_array_type is
variable s : start_array_type;
begin
  for pos_start in 0 to max_step loop
    s(pos_start) := get_start(pos_start);
  end loop;
  return s;
end get_start_array;

constant start_array : start_array_type := get_start_array(step);

signal  t_in_block  : std_logic_vector(n-1 downto 0);
signal  t_cd_block  : std_logic_vector(n-1 downto 0);

begin

--         +-- i+1 --+-- i+1 --+-- i+1 --+-- i --+-- i --+
--         for each of these block instantiate separate cd_block_L

s: for start in 0 to step-1 generate
  -- for each index
  g: for index in 0 to n-1 generate
    -- if it is in
    i_in:  if start+index*step < n generate
           t_in_block(start_array(start)+index) <= i(start+index*step);
                  o(start+index*step) <= t_cd_block(start_array(start)+index);
     end generate;
    end generate g;

    -- create a contradiction detect block for each start
n_lt_L: if start_array(start+1)-start_array(start) < L generate
          t_cd_block(start_array(start+1)-1 downto start_array(start)) <=
                    (start_array(start+1)-1 downto start_array(start) => '0');
        end generate;

n_ge_L: if start_array(start+1)-start_array(start) >= L generate
mystep: cd_block_L generic map (n=> start_array(start+1)-start_array(start), L=>L)
                  port map (
                     i=>t_in_block(start_array(start+1)-1 downto start_array(start)),
                     o=>t_cd_block(start_array(start+1)-1 downto start_array(start)));
    end generate;
  end generate s;
end;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.vdw_vhdl.all;

entity cd_pipe_stage is
generic (n: natural := vdw_n;
         k: natural := vdw_k;
         L: natural := vdw_L;
         cd_min_step : natural;
         cd_max_step : natural);
port (c0  : in  std_logic_vector(n-1 downto 0);
      cd_out : out std_logic_vector(n-1 downto 0));
end cd_pipe_stage;

architecture behavioral of cd_pipe_stage is
component cd_block
   generic (n    : natural;
            L    : natural;
            step : natural);
   port (i : in  std_logic_vector(n-1 downto 0);
         o : out std_logic_vector(n-1 downto 0));
end component;

type c_unset_out_type is array (cd_max_step downto cd_min_step) of std_logic_vector(n-1 downto 0);
signal  c_unset_out_x : c_unset_out_type ;

type c_unset_out_xx_type is array (n-1 downto 0) of std_logic_vector(cd_max_step downto cd_min_step);
signal  c_unset_out_xx : c_unset_out_xx_type ;

begin

max_step_gt_0: if (n-1)/(L-1) > 0 generate

g_out: for i in cd_max_step downto cd_min_step generate
       mystep: cd_block generic map (n=>n,step=>i, L=>L) port map (i=>c0, o=>c_unset_out_x(i));
end generate g_out;

itestfor: for i in n-1 downto 0 generate
  jtestfor:  for j in cd_max_step downto cd_min_step generate
                  c_unset_out_xx(i)(j) <= c_unset_out_x(j)(i);
             end generate jtestfor;

             cd_out(i) <= '0' when c_unset_out_xx(i)=(c_unset_out_xx(i)'range=>'0') else '1';
           end generate itestfor;

end generate max_step_gt_0;

end;

-- ----------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.vdw_vhdl.all;

entity cd is
generic (n: natural := vdw_n;
         k: natural := vdw_k;
         L: natural := vdw_L;
         constant num_stages : natural := NUM_LOGIC_STAGES-NUM_CD_CHECK_ZERO_STAGES;
         constant debug_string:string:="";
         constant debug: integer:=0);
port (i  : in std_logic_vector(n-1 downto 0);
      clk : in std_logic;
      o : out std_logic_vector(n-1 downto 0));
end cd;

architecture behavioral of cd is

constant max_step : natural := (n-1)/(L-1);

function cd_point_max(stage_num: natural)
         -- and given n, cp_stage_min and num_stages
return natural is
begin
  -- return stage_num; -- FIXME -- just for testing
  -- return stage_num*max_step/num_stages;
  return stage_num*max_step/num_stages;

  -- change to 0, max_step/(2*num_stages), max_step/num_stages, .., max_step/num_stages, (max_step*2)/num_stages
--    if stage_num=0 then
--       return 0;
--    elsif stage_num=num_stages then
--       return max_step;
--    else
--       return max_step/(2*num_stages)+(stage_num-1)*max_step/num_stages;
--    end if;

end cd_point_max;

component cd_pipe_stage
generic (n: natural := vdw_n;
         k: natural := vdw_k;
         L: natural := vdw_L;
         cd_min_step : natural;
         cd_max_step : natural);
port (c0  : in  std_logic_vector(n-1 downto 0);
      cd_out : out std_logic_vector(n-1 downto 0));
end component;

component print_vec
generic (n: natural := 1;
         constant debug_string:string:="";
         constant debug:integer:=0);
port (i : in std_logic_vector(n-1 downto 0);
      clk : in std_logic);
end component;

type t_stages is array(1 to num_stages) of std_logic_vector(n-1 downto 0);
signal c0_stages : t_stages;
signal results : t_stages;
signal stage_results : t_stages;
begin

  stages: for stage in 1 to num_stages generate
     cd_stage: cd_pipe_stage generic map(n=>n, k=>k, L=>L,
                                         cd_min_step=>cd_point_max(stage-1)+1, -- (i-1)*max_step/num_stages+1,
                                         cd_max_step=>cd_point_max(stage)) -- i*max_step/num_stages
                           port map (c0=>c0_stages(stage), cd_out=>stage_results(stage));
  end generate;

num_stages_eq_1: if (num_stages=1) generate
                   o <= stage_results(num_stages);
                 end generate;

num_stages_gt_1: if (num_stages>1) generate
                   o <= results(num_stages-1) or stage_results(num_stages);
                 end generate;

 process(clk)
 begin
   if rising_edge(clk) then
      c0_stages(1) <= i;
      if num_stages>1 then
         results(1) <= stage_results(1);
         for i in 2 to num_stages loop
            results(i) <= results(i-1) or stage_results(i);
            c0_stages(i) <= c0_stages(i-1);
         end loop;
      end if;
   end if;
 end process;
end behavioral;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.vdw_vhdl.all;

entity cd_verify is
generic (n: natural := vdw_n;
         k: natural := vdw_k;
         L: natural := vdw_L);
port (
       clk : in std_logic;
       i  : in std_logic_vector(n-1 downto 0);
       o : out std_logic);
end cd_verify;

architecture behavioral of cd_verify is

component cd is
generic (n: natural := vdw_n;
         k: natural := vdw_k;
         L: natural := vdw_L;
         constant num_stages : natural := NUM_LOGIC_STAGES-NUM_CD_CHECK_ZERO_STAGES;
         constant debug_string:string:="";
         constant debug: integer:=0);
port (i  : in std_logic_vector(n-1 downto 0);
      clk : in std_logic;
      o : out std_logic_vector(n-1 downto 0));
end component;

component pipe_is_not_zero
generic (n: natural;
         constant num_stages: natural;
         constant debug_string:string:="";
         constant debug: integer:=0);
port (i : in std_logic_vector(n-1 downto 0);
      clk : in std_logic;
      o : out std_logic);
end component;

signal cd_detect_out : std_logic_vector(n-1 downto 0);

begin

  vdw_cd_detect:    cd generic map (n=>n, k=>k, L=>L, num_stages=>NUM_LOGIC_STAGES-NUM_CD_CHECK_ZERO_STAGES)
                                   port map (i=>i, clk=>clk, o=>cd_detect_out);

  vdw_cd_out_zero:  pipe_is_not_zero generic map (n=>n, num_stages=>NUM_CD_CHECK_ZERO_STAGES)
                                     port map (i=>cd_detect_out, clk=>clk, o=>o);

end behavioral;
```

# Current Research Challenges

- End-to-end Xilinx tool flow (R.E. the BITFILE)
- FPGA vendor specific primitives (e.g SIMPRIMS)
- Asynchronous circuits
- ECC and PKI algorithms
- Large bit vector sizes, deep semantic optimizations
- Equivalence checking for modes
- Unbounded equivalence checking in general
- Over allocated input/output buffers (IOBs)
- Symbolically terminating recursive functions
- … many more!

# Equivalence Checking References


A. Biere. Invited talk - SAT in Formal Hardware Verification. 8th Intl. Conf; on Theory and Applications of Satisfiability Testing (SAT'05), St. Andrews, Scotland, (2005).

D. Brand. Verification of Large Synthesized Designs. Proc. Intl Conf. Computer-Aided Design pp. 534-537 (1993).

E. Goldberg, Y. Novikov. How good can a resolution based SAT-solver be? SAT-2003, LNCS 2919, pp. 35-52 (2003).

F. Krohm, A. Kuehlmann, and A. Mets. The Use of Random Simulation in Formal Verification. Proc. of Int'l Conf. on Computer Design, Oct (1996).

A. Kuehlmann, F. Krohm. Equivalence Checking Using Cuts and Heaps. In Design Automation Conference (1997).

A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust Boolean Reasoning for Equivalence Checking and Function Property Verification. IEEE Trans. CAD, Vol. 21, No. 12, pp. 1377-1394 (2002).

A. Kuehlmann. Dynamic Transition Relation Simplification for Bounded Property Checking. In ICCAD (2004).

J. Lewis. Cryptol, A Domain Specific Language for Cryptography. http://www.cryptol.net/docs/CryptolPaper.pdf (2002).