

Hardware Verification

2IMF20

Julien Schmaltz

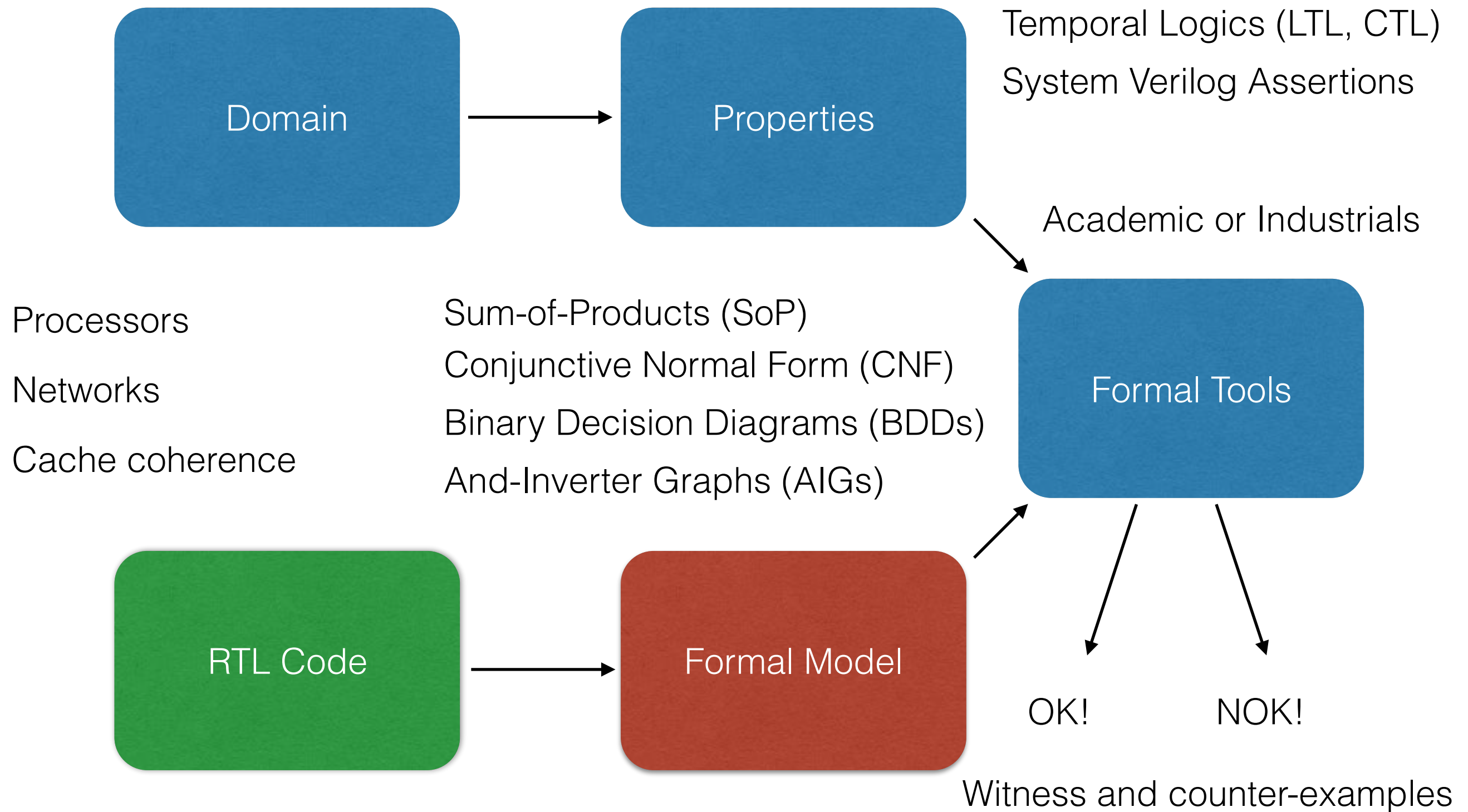
Lecture 02:
Boolean Functions, SAT, CEC



Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Course content - Formal tools



Goal: Reason about hardware

- » Given two circuits: do they compute the same function?
 - » Equivalence checking
 - » Combinatorial and Sequential (next lecture)
 - » Notion of a “Miter” (XOR between outputs)
- » Given a property and circuit: prove that the circuit satisfies the property
 - » Formal Property Verification (FPV) (some lectures from now)
- » In all cases, ***a mathematical representation of the circuit is needed.***

Important concepts

- » The need to represent Boolean functions efficiently
- » Different representations have different pros and cons
- » Get to know the main representations used in practice
 - » SoP
 - » DNF and CNF
 - » DAG
 - » AIG
 - » BDD
- » Have a feeling about how good/bad they are
- » Know how they are used in Combinatorial Equivalence Checking
- » Basic principle of SAT solvers

- » Note: representing Boolean functions is active research
 - » “Cyclic Boolean Circuits” by Riedel and Bruck, Discrete Applied Mathematics (2012)

Program for today

- » Boolean functions
- » Boolean satisfiability (very brief, more later)
- » Combinatorial Equivalence Checking

Hardware to Formal Representation

- » The first step before applying any formal analysis technique is to obtain a formal representation of the design.
 - » from 4 valued logic to Booleans
- » Symbolic Boolean expressions of the wires.
- » Different representations of these expressions
 - » Directed Acyclic Graphs (DAG)
 - » Sum-of-Products (SoP)
 - » Conjunctive Normal Form (CNF)
 - » Disjunctive Normal Form
 - » Binary Decision Diagrams (BDD's)
 - » And-Inverter Graphs (AIG's)

Boolean functions

- Matches most digital hardware, other hardware can be translated into Boolean functions.
- For example, Verilog HDL has 0, 1, and also:
X (unknown / error)
Z (not driven, open wire)
- To translate, simply use two 'bits', or Boolean values:
(False,False): 0
(False,True): X
(True,False): Z
(True,True): 1
(any other choice will also do)

Why Boolean functions

- For this lecture, we assume two valued logic without X or Z
- **Means assumptions of the input circuits !!**
 - some examples: no combinatorial cycles, no tristate buffers
- As Boolean value, we write:
0 for False, and 1 for True
- We talk about & for 'and', | for 'or', ! for 'not', etc.
We can assume we are talking about the logical functions:

$$1 \& 1 = 1, x \& 0 = 0, 0 \& x = 0$$

$$0 | 0 = 0, x | 1 = 1, 1 | x = 1$$

$$!0 = 1, !1 = 0$$

Expressing Boolean functions:

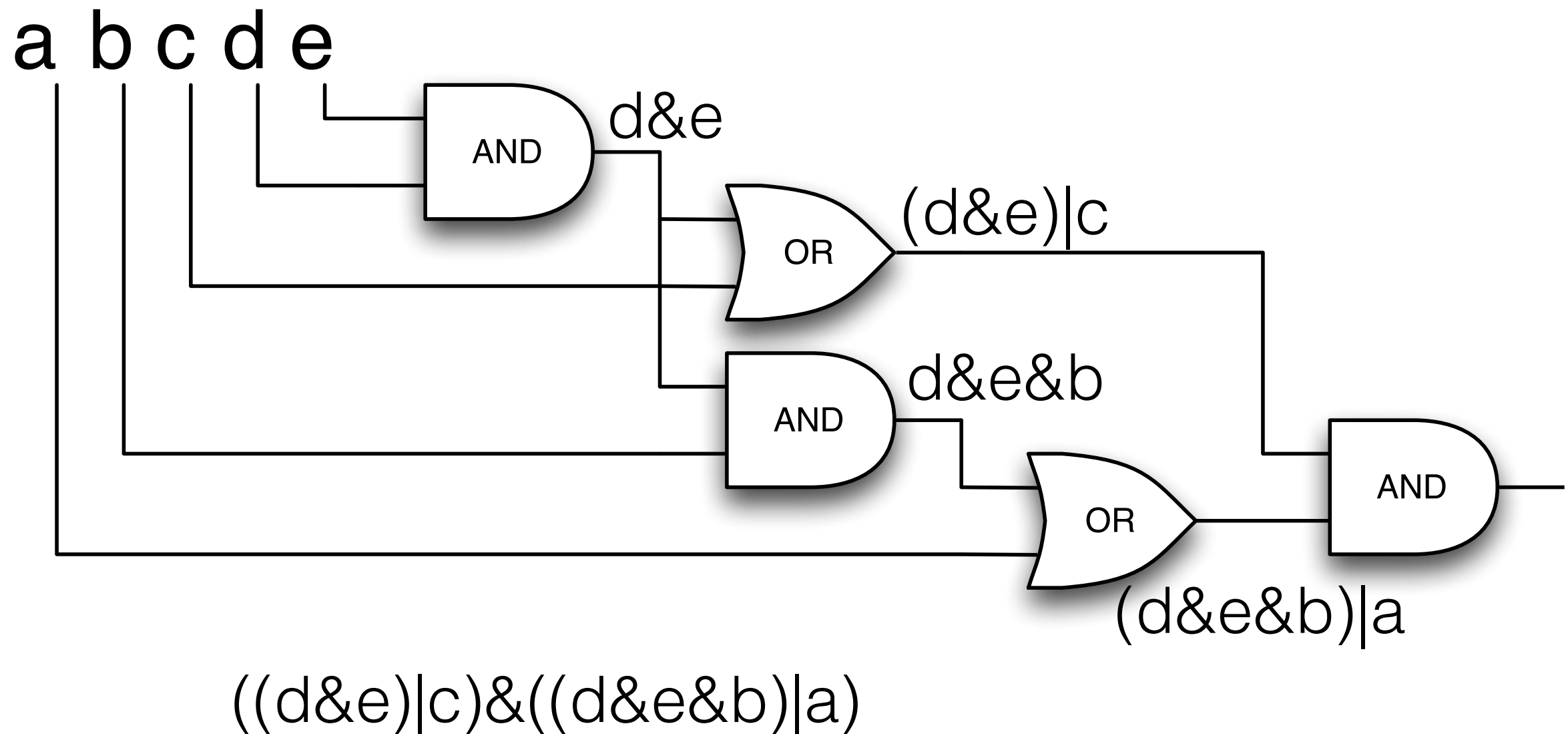
- Data structure should be:
 - Efficient to construct
 - Easy to reason about

How compact can we store a function?

- Truth table:
 - for a Boolean function on N inputs
 - 2^N possible input assignments
 - So, 2^N rows in the truth table
- As most Boolean functions are not “any Boolean function”: we can have smaller representations, most of the time.
- For instance, only represent rows for which an output is 1

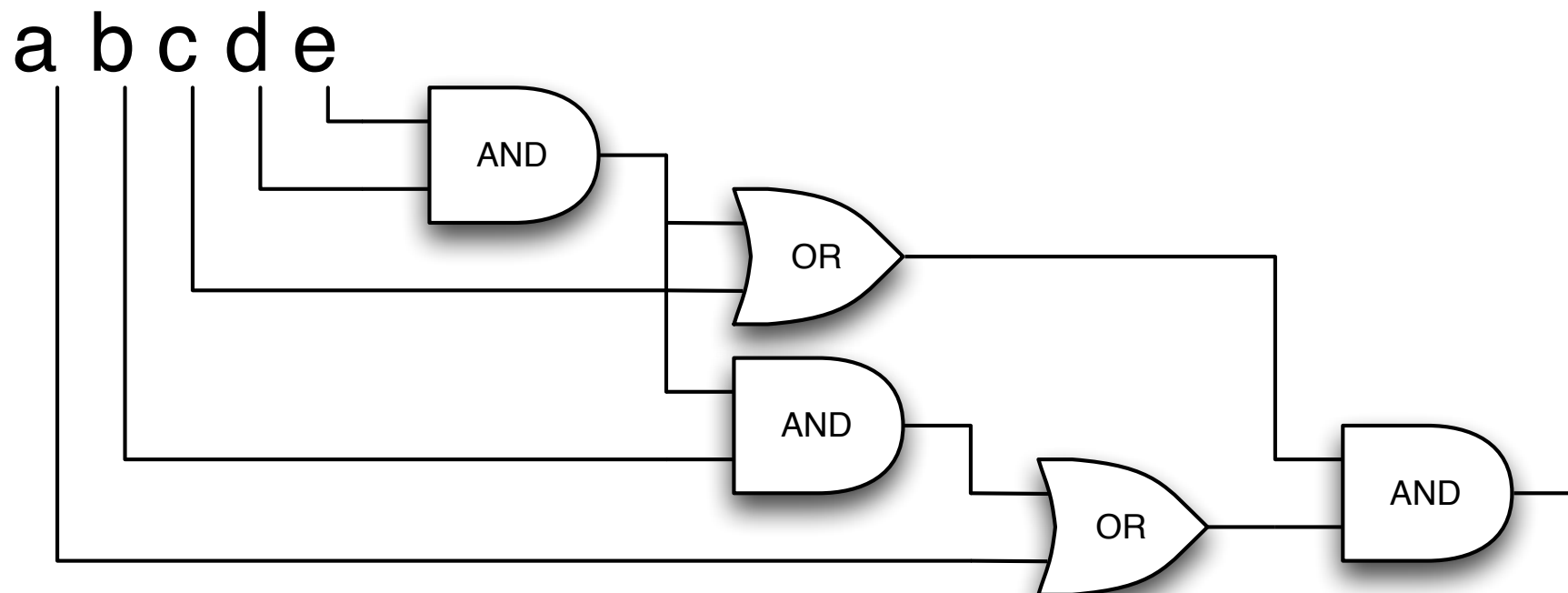
Directed acyclic graph (DAG)

- A directed acyclic graph is a way to represent a Boolean function, it is often used as a synonym for 'circuit'.
- Can be more compact than the function written out!



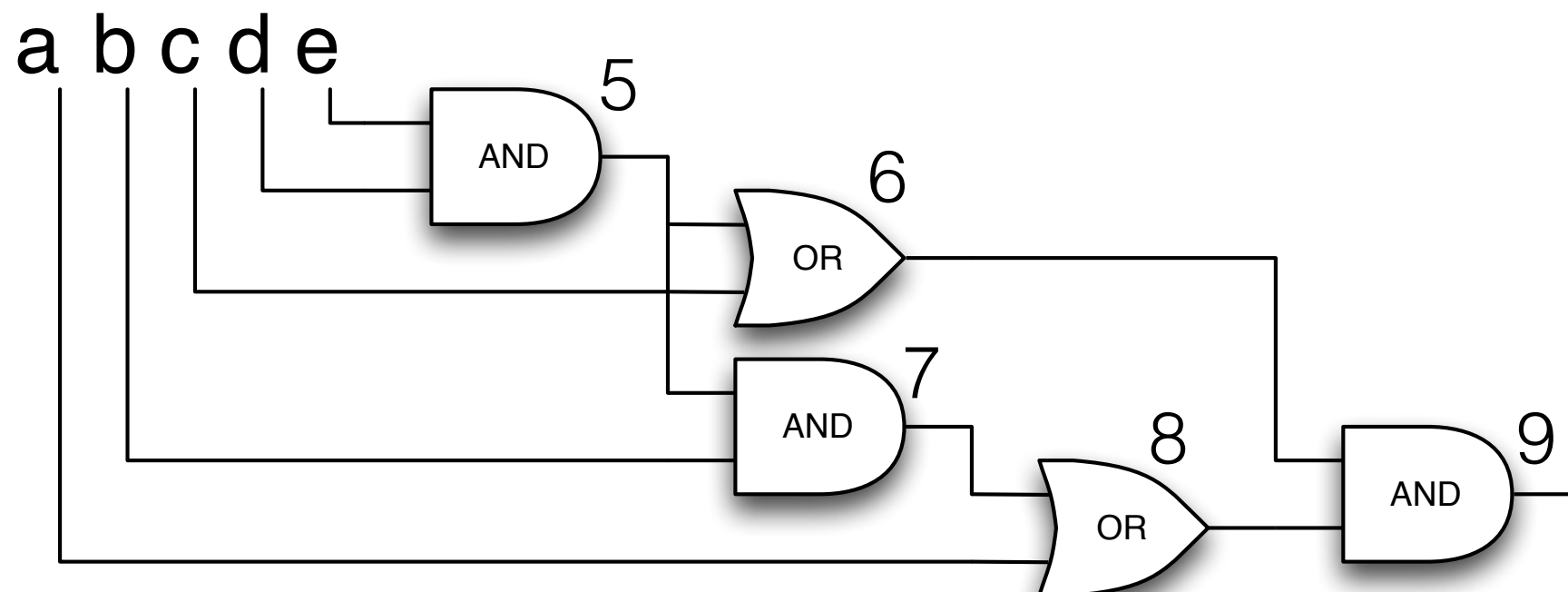
Directed acyclic graph (DAG)

- DAG is a list of 'gates'. A gate at position i has:
 - a Boolean function assigned to it
 - Each Boolean function gets a list of pointers to other gates, from which it gets its input values
 - Each of those numbers need to point backwards
 - Variables are represented as gates without inputs



Directed acyclic graph (DAG)

- 0: variable "a"
- 1: variable "b"
- 2: variable "c"
- 3: variable "d"
- 4: variable "e"
- 5: and [3,4]
- 6: or [5,2]
- 7: and [5,1]
- 8: or [7,0]
- 9: and [6,8]

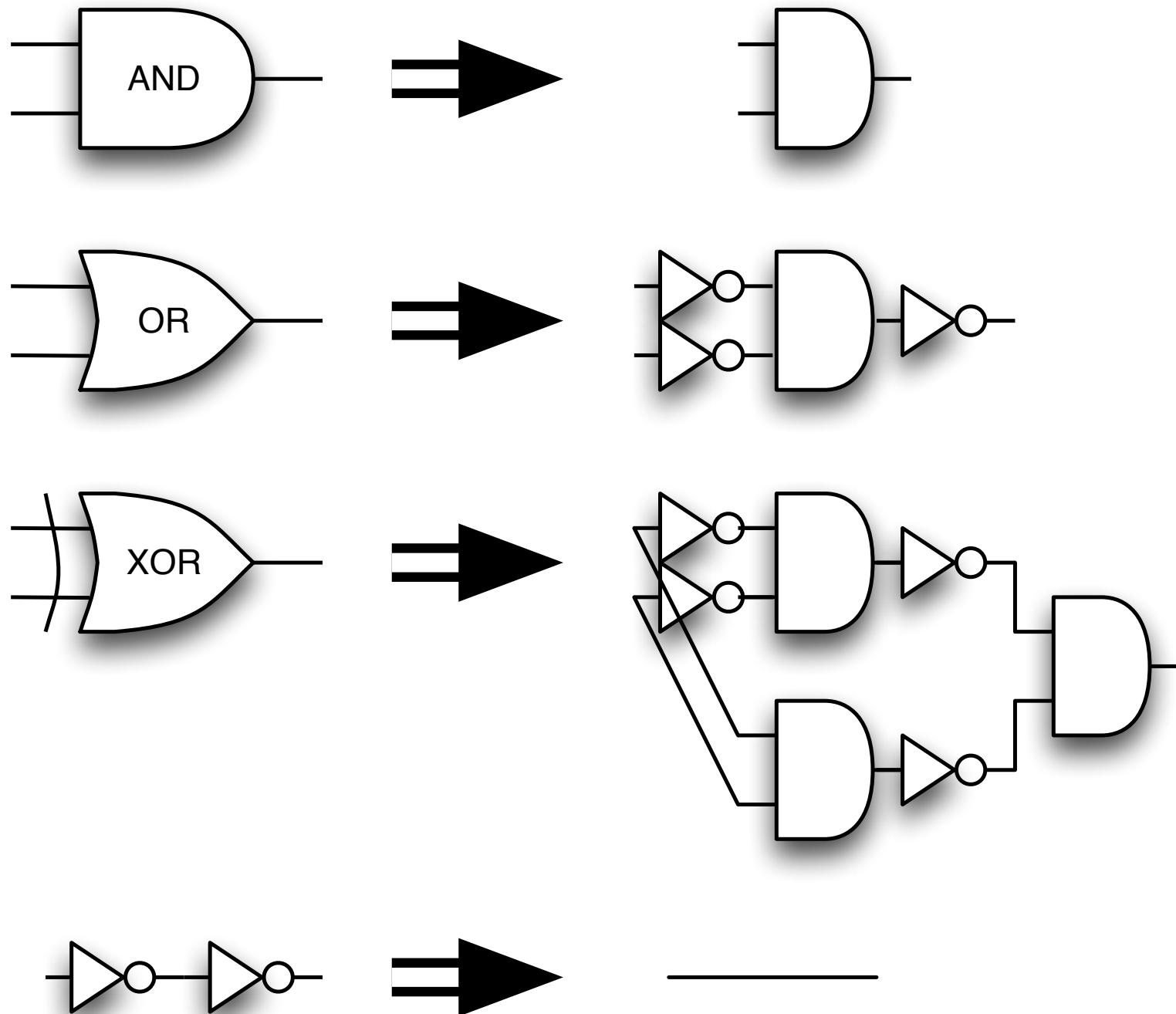


Directed acyclic graph (DAG)

- DAG is easy to construct (follows hardware directly)
- Large amount of different gates:
 - makes it hard to write and maintain programs that reason with DAGs
- Alternative: And-Inverter Graph (AIG)
 - only two gates: AND and NOT

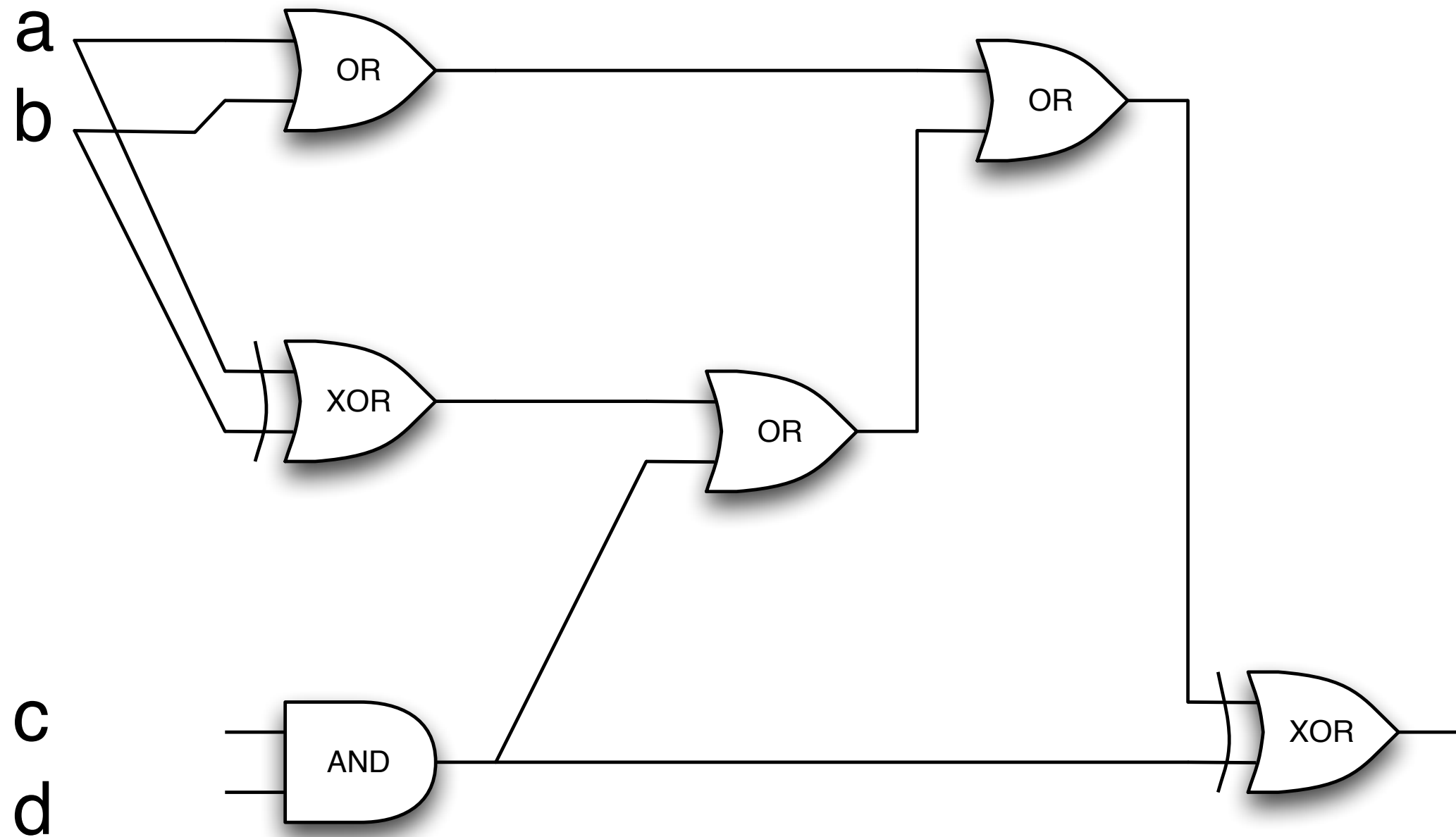
And-Inverter Graph (AIG)

- Every gate can be converted into a fixed amount of AIG gates



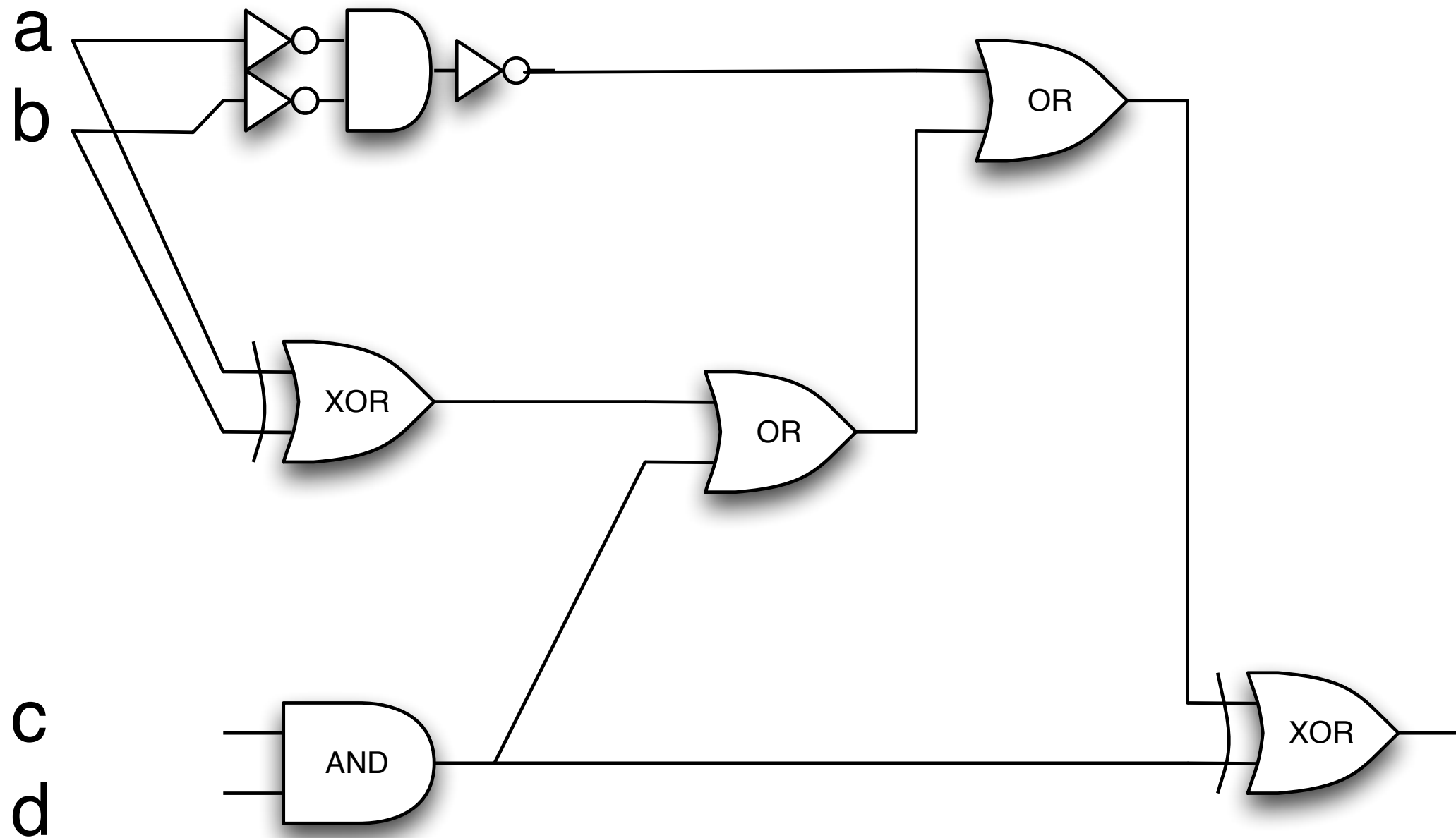
AIG

- Every gate can be converted into a fixed amount of AIG gates



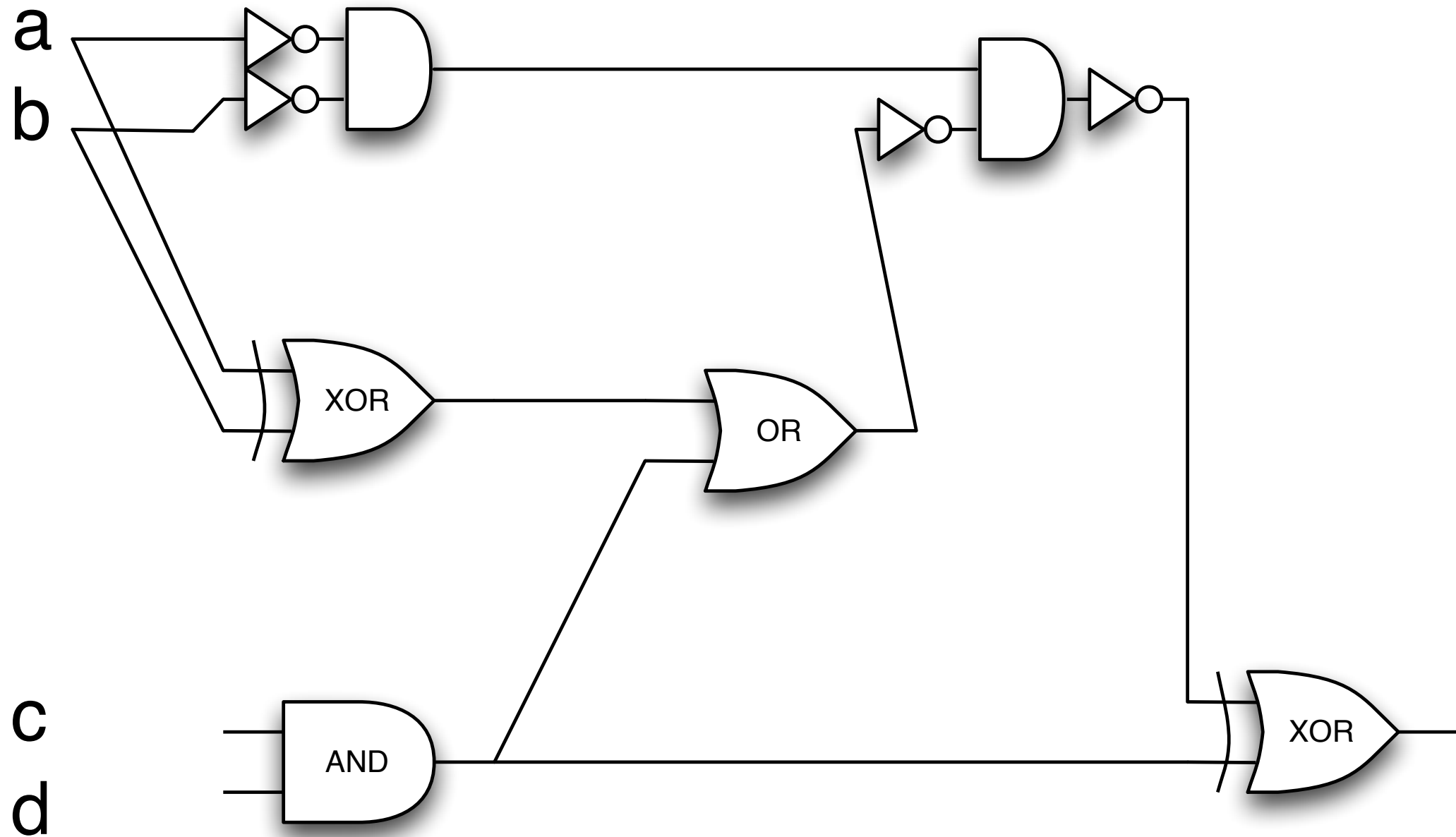
AIG

- Every gate can be converted into a fixed amount of AIG gates



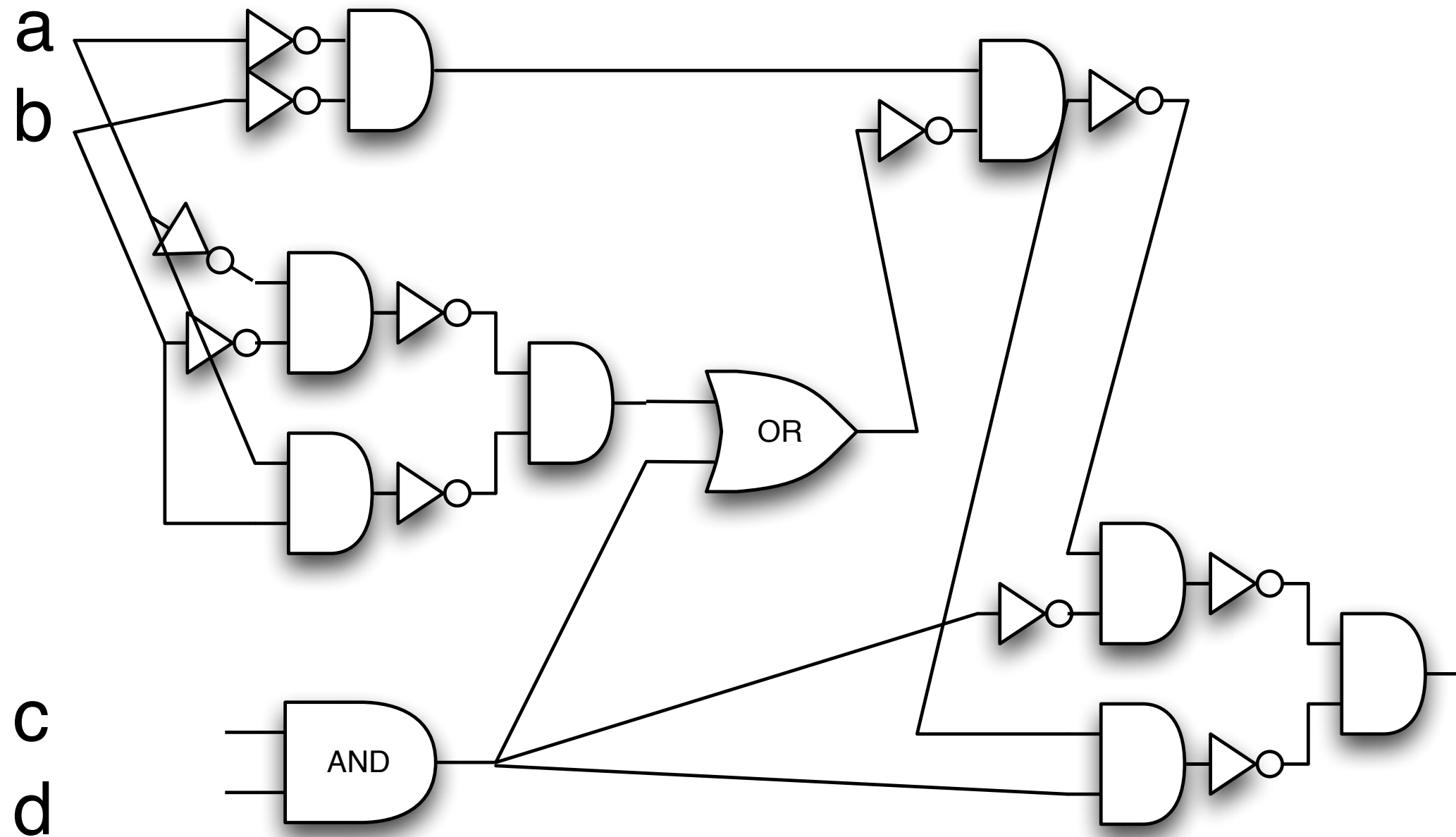
AIG

- Cannot put two inverters between two gates



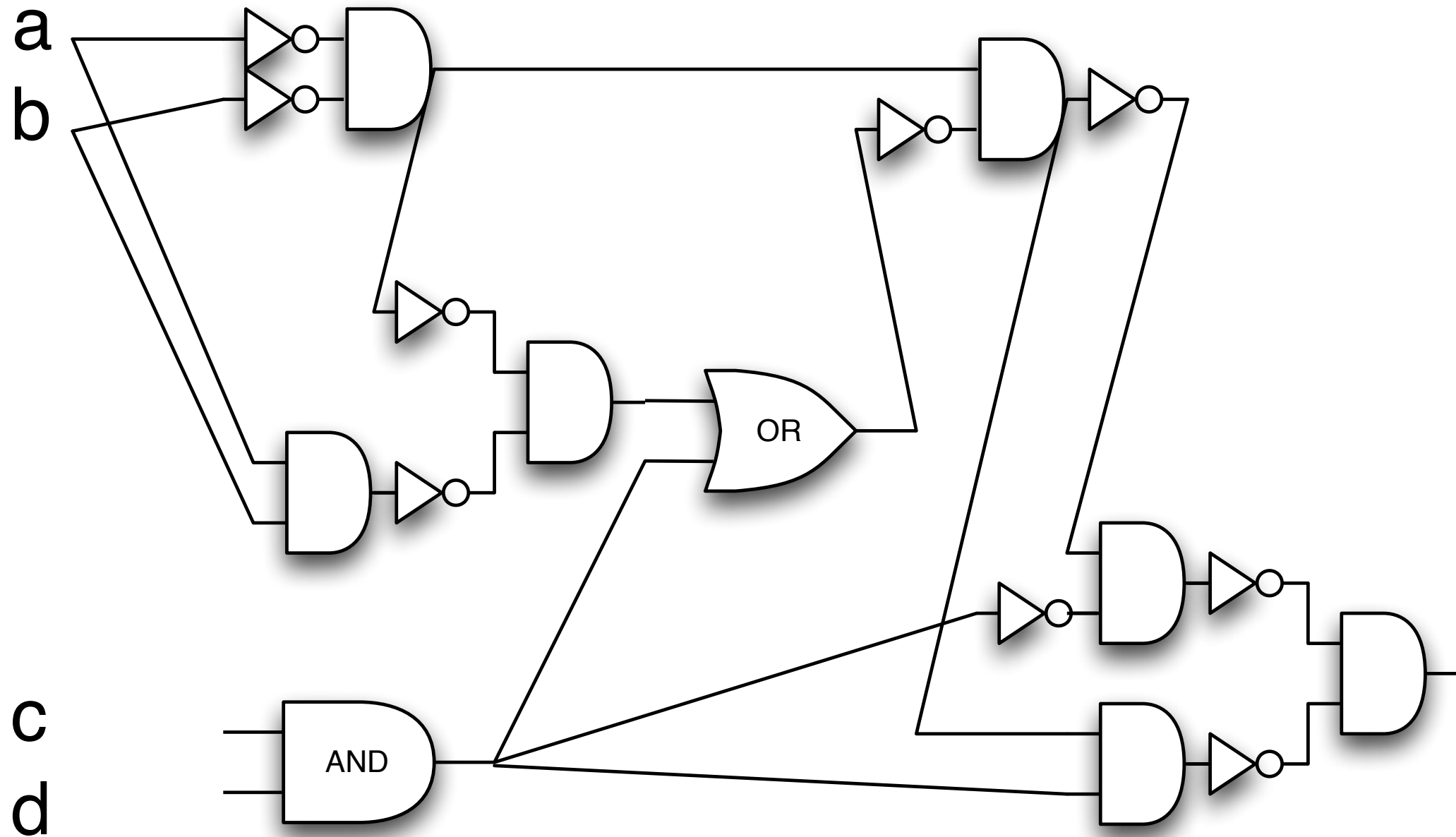
AIG

- Sometimes, AND gates are the same



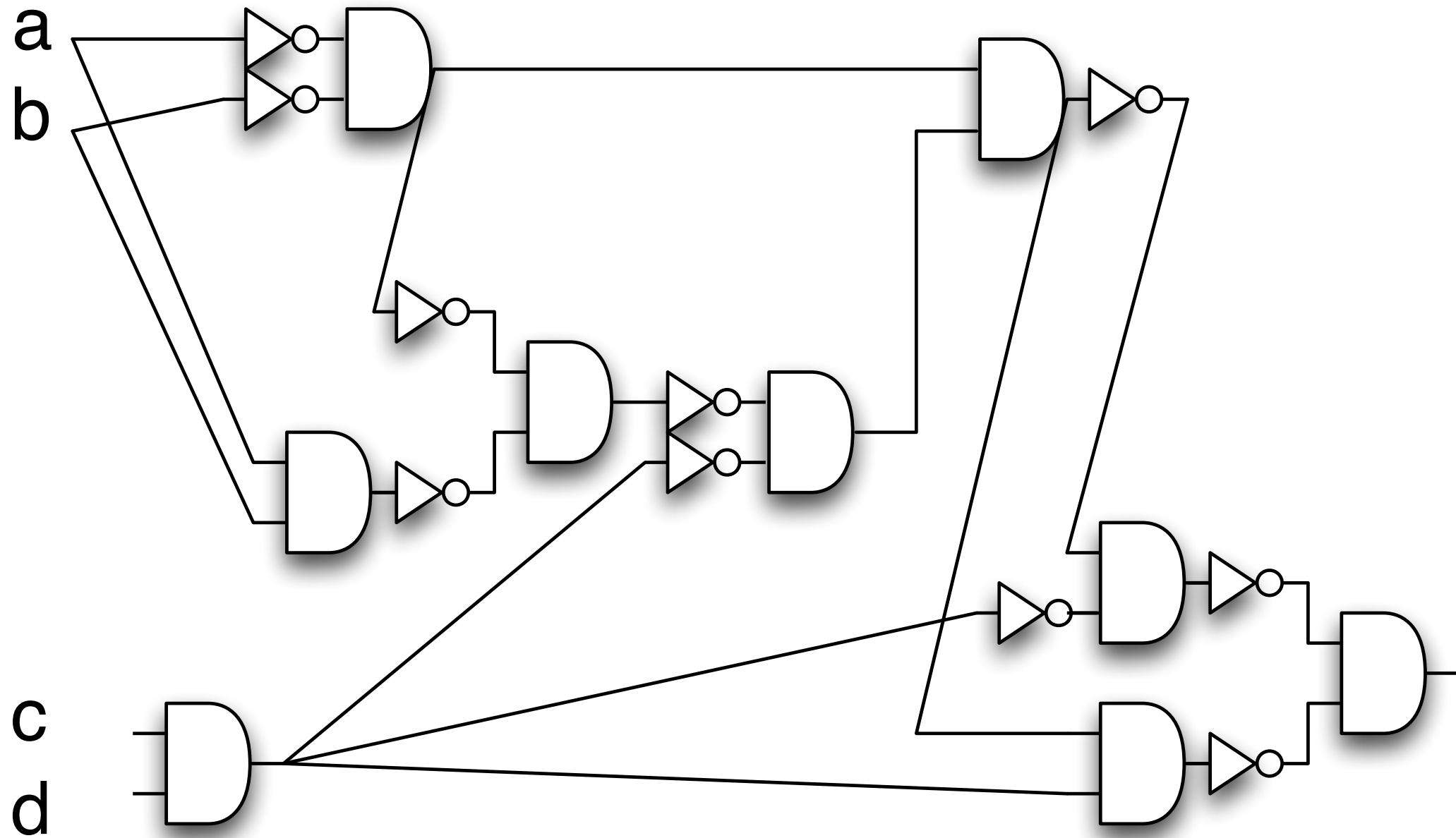
AIG

- We may share AND gates that have the same input



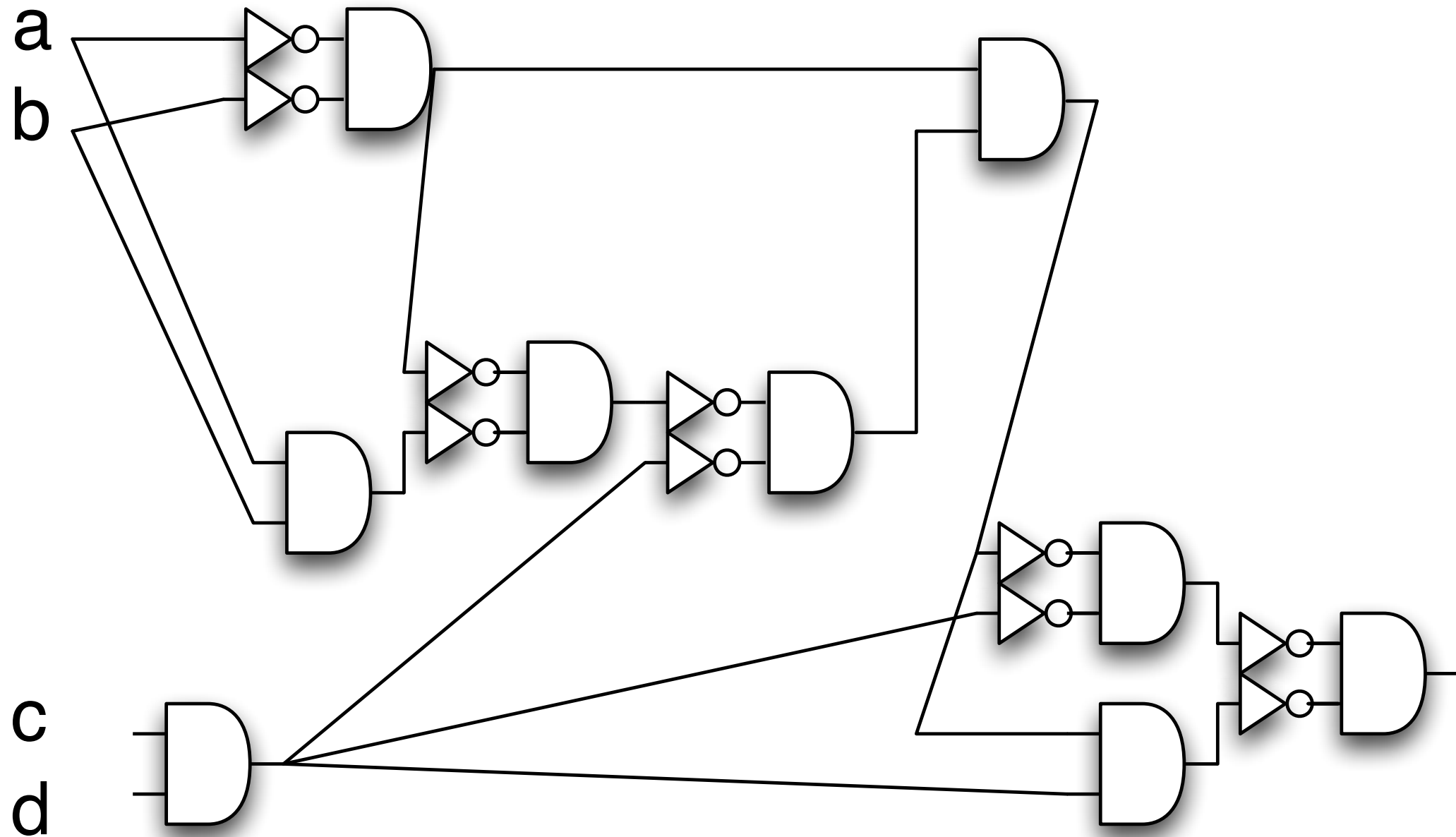
AIG

- Two inverters in a row are removed (not not is identity)



AIG

- Conceptually, inverters belong with the next gate

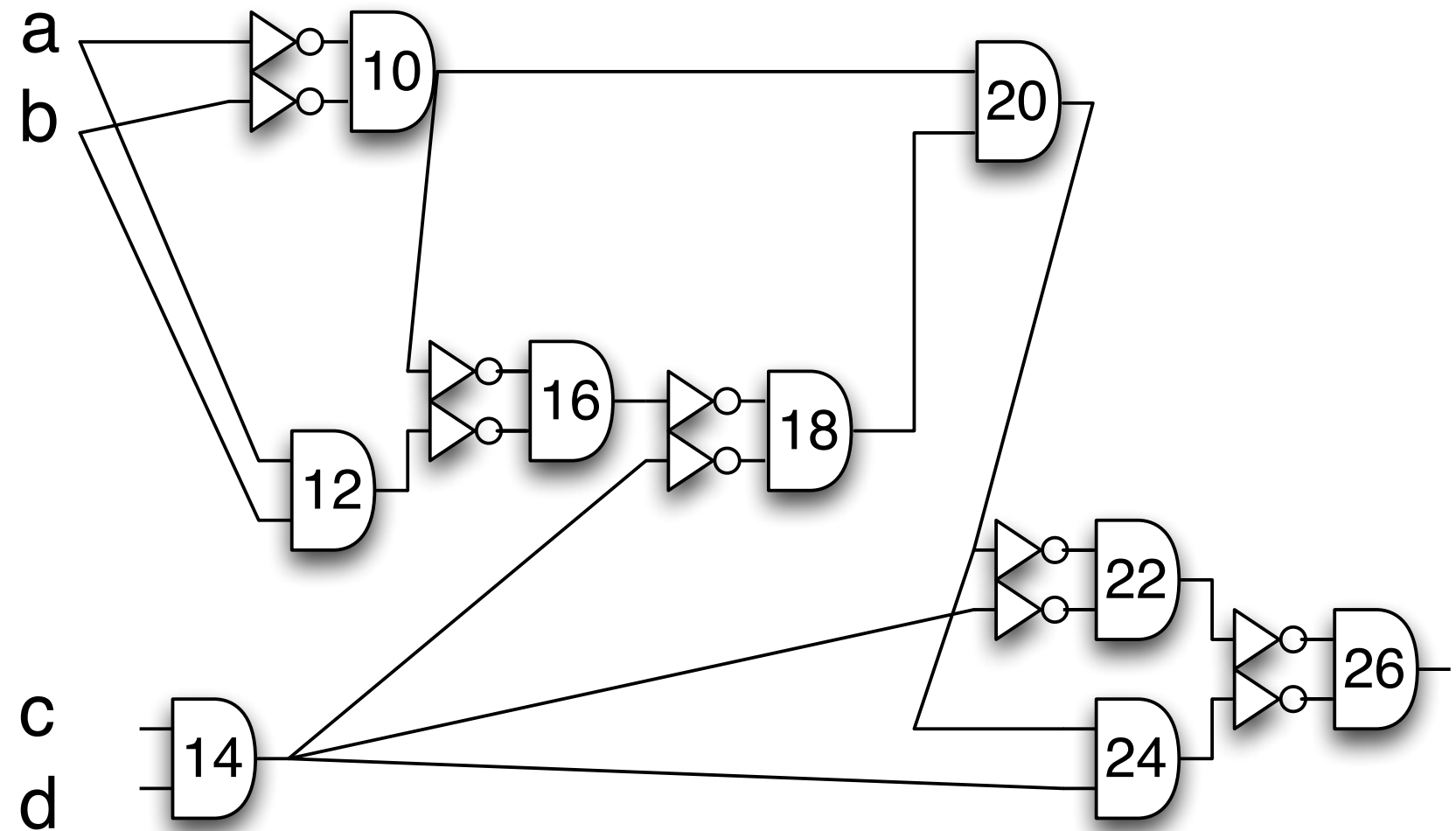


AIG

- AIG storage is really compact:
 - All nodes get even numbers
 - First node, 0, stands for False
 - To negate a node, use its number +1

AIG

- 2 *a*
- 4 *b*
- 6 *c*
- 8 *d*
- 26 *output*
- 10 3 5 *!a&!b*
- 12 2 4
- 14 6 8
- 16 11 13
- 18 17 15
- 20 10 18
- 22 21 15
- 24 20 14
- 26 23 25



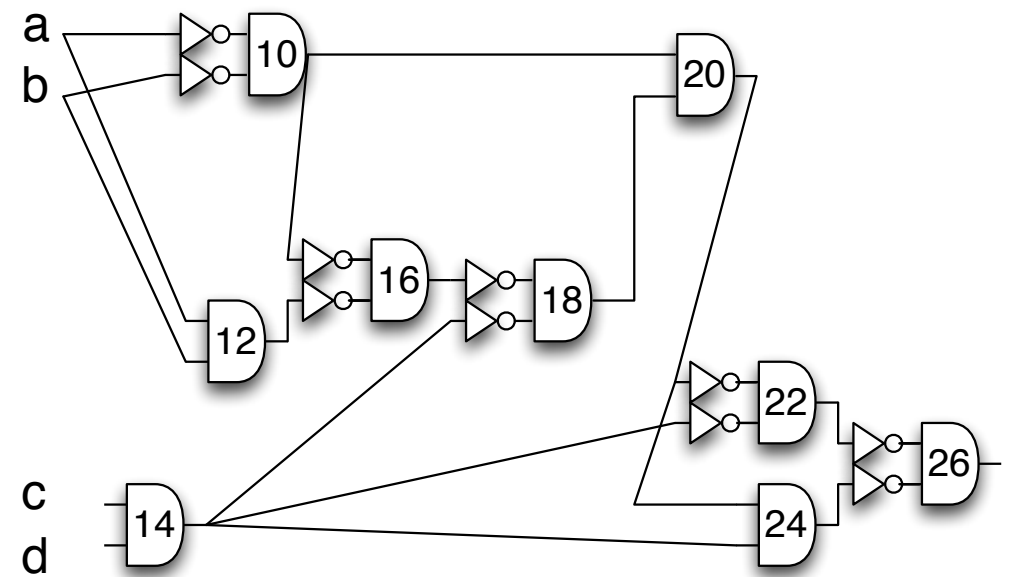
Advantages of AIG

- Linear size compared to DAG
- Useful as intermediate structure for synthesis:
 - NAND gate has 4 transistors in CMOS
 - NOT has 2
 - AIG Structure gives good area and clock-speed estimates
 - Example tool that uses AIGs:

ABC, A System for Sequential Synthesis and Verification
Berkeley Logic Synthesis and Verification Group

Data structures, so far:

- DAG
- AIG (Is roughly the same, but with a few conditions)
- Next:
conjunctive normal form



Conjunctive normal form (CNF)

- Simple grammar:
CNF = (Disjunction) & CNF
CNF = (Disjunction)
CNF = True
Disjunction = Term | Disjunction
Disjunction = Term
Disjunction = False
Term = Variable
Term = ! Variable
- Example:
 $(x \mid y) \& (!z \mid x \mid !y) \& (z \mid !x)$
- Rules:
 - All variables within a disjunction must be unique. Including x and $!x$: they do not occur in the same disjunction

Creating CNF

- Naive way:
Procedure for AND and NOT, translate from AIG
- AND is trivial:
AND of $[(a|b|c)\&(d|e|f)\&(i|j)] [(g|h)\&(i|j)]$ becomes:
 $[(a|b|c)\&(d|e|f)\&(g|h)\&(i|j)]$ or even
 $[(a|b|c)\&(d|e|f)\&(i|j)\&(g|h)\&(i|j)]$
- NOT is problematic: goes from CNF to DNF back to CNF
 $[(a|b|c)\&(d|e|f)\&(i|j)]$ becomes:
 $[(-a|-d|-i)\&(-a|-d|-j)\&(-a|-e|-i)\&(-a|-e|-j)\&(-a|-f|-i)\&(-a|-f|-j)$
 $\&(-b|-d|-i)\&(-b|-d|-j)\&(-b|-e|-i)\&(-b|-e|-j)\&(-b|-f|-i)\&(-b|-f|-j)$
 $\&(-c|-d|-i)\&(-c|-d|-j)\&(-c|-e|-i)\&(-c|-e|-j)\&(-c|-f|-i)\&(-c|-f|-j)]$
- **This is not going to scale!**

Another way to create CNF - AND

- » Consider $C = \text{AND}(A, B)$
- » Goal: create CNF formula f such that $f(a, b, c) == (C = A \& B)$
- » if A is false, then C is false
 - » “ $\neg A$ implies $\neg C$ ”, logically equivalent to the clause $A \mid \neg C$
 - » similarly for B : $B \mid \neg C$
- » If A and B are true, then C is true
 - » “ $A \& B$ implies C , logically we get $\neg A \mid \neg B \mid C$
- » Finally, the encoding for an AND-gate is:
 - » $(A \mid \neg C) \& (B \mid \neg C) \& (\neg A \mid \neg B \mid C)$
- » Linear expansion: 3 clauses for each AND-gate

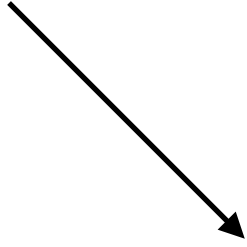
Another example: XOR

- » Consider $C = \text{XOR}(A, B)$
- » Goal: create CNF formula f such that $f(a, b, c) == (C = A \text{ XOR } B)$
- » if A and B are false, then C is false
 - » “ $\neg A$ and $\neg B$ implies $\neg C$ ”, logically equivalent to $A \mid B \mid \neg C$
- » If A is true and B is false, then C is true
 - » “ $A \ \& \ \neg B$ implies C ”, logically we get $\neg A \mid B \mid C$
 - » Symmetric case: $A \mid \neg B \mid C$
- » If A and B are true, then C is false
 - » “ $A \ \& \ B$ implies $\neg C$ ”, logically we get $\neg A \mid \neg B \mid \neg C$
- » Finally, the encoding for an XOR-gate is:
 - » $(A \mid B \mid \neg C) \ \& \ (\neg A \mid B \mid C) \ \& \ (A \mid \neg B \mid C) \ \& \ (\neg A \mid \neg B \mid \neg C)$

CNF

- Like AIG, CNF is linear in the size of the original DAG, but only if we add 'helper' variables.
- CNF is used as the internal structure of most SAT solvers, including MiniSAT
- CNF is the input format in the SAT competition, and in many of its variations
- Some optimisations are easier on AIGs, so tools built on SAT solvers sometimes translate Boolean primitives to AIG to CNF, for example: Boolector. Other tools translate Boolean primitives to CNF directly, such as Yices

CNF: typical optimisations

- Never have disjunctions with one variable:
 $(a \mid b \mid v10) \& (-a \mid -v10) \& (-b \mid -v10) \&$
 $(-a \mid -b \mid v12) \& (a \mid -v12) \& (b \mid -v12) \&$
 $(-c \mid -d \mid v14) \& (c \mid -v14) \& (d \mid -v14) \&$
 $(v10 \mid v12 \mid v16) \& (-v10 \mid -v16) \& (-v12 \mid -v16) \& (-v10)$ 
- $-v10$ is necessarily True, so $v10$ is False
 $(-a) \& (-b) \&$
 $(-a \mid -b \mid v12) \& (a \mid -v12) \& (b \mid -v12) \&$
 $(-c \mid -d \mid v14) \& (c \mid -v14) \& (d \mid -v14) \&$
 $(v12 \mid v16) \& (-v12 \mid -v16)$
- New single variable disjunctions: a and b are False
 $(-v12) \&$
 $(-c \mid -d \mid v14) \& (c \mid -v14) \& (d \mid -v14) \& (v12 \mid v16) \& (-v12 \mid -v16)$
- New single variable disjunctions: $v12$ is False
 $(-c \mid -d \mid v14) \& (c \mid -v14) \& (d \mid -v14) \& (v16)$
- New single variable disjunctions: $v16$ is True

CNF: typical optimisations (2)

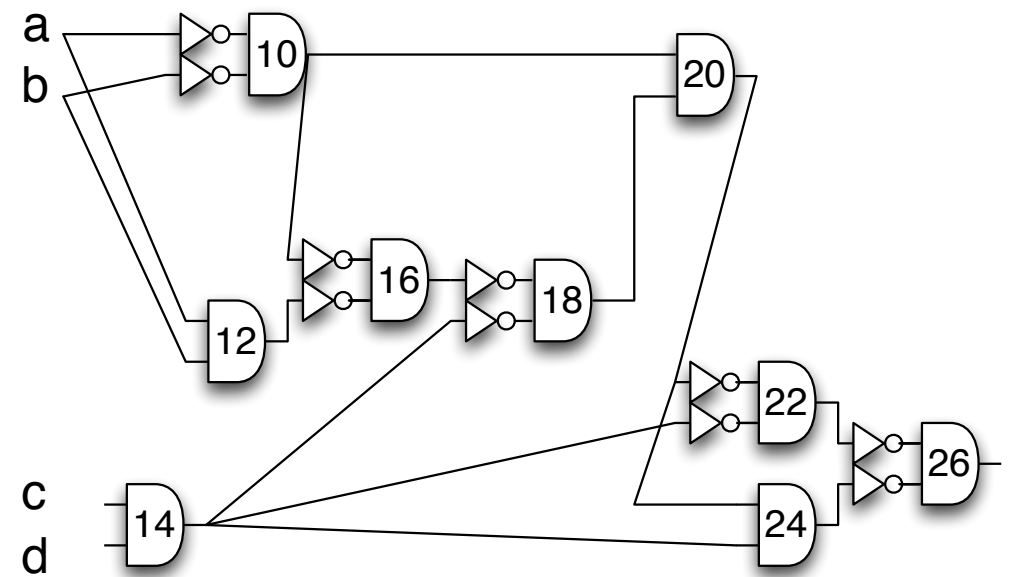
- Remove strictly larger disjunctions:
 $(a \mid b)$ implies $(a \mid b \mid c)$, so $(a \mid b \mid c)$ is redundant.
Replace $(a \mid b) \ \& \ (a \mid b \mid c)$ by $(a \mid b)$.
- If a variable only occurs positively/negatively, remove it:
 $(-c \mid -d \mid v14) \ \& \ (c \mid -v14)$
becomes: $(c \mid -v14)$ by assigning d to False,
which then becomes: True
by assigning c to True (or $v14$ to False)
- If a variable occurs twice, positively in one clause, and negatively in another clause, we can merge these clauses:
 $(-c \mid -d \mid v14) \ \& \ (a \mid b \mid c)$ becomes $(a \mid b \mid -d \mid v14)$
(if c does not occur elsewhere!)
- Recall the rule:
Never have $-x$ and x in one disjunction (it is always True)

CNF: summary

- Can be constructed in linear size if we allow for additional variables
- Easy to reason with
- Common file format for many purposes

Data structures, so far:

- DAG
- AIG
- CNF
- Next: binary decision diagram



Binary Decision Diagram (BDD)

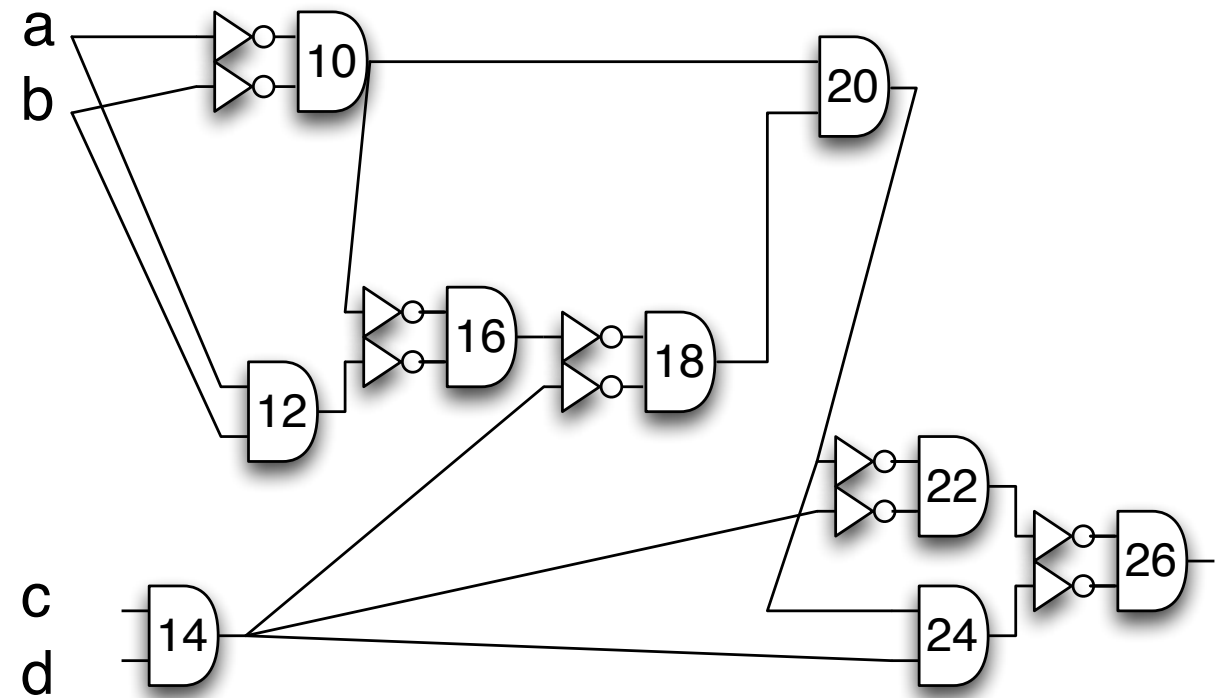
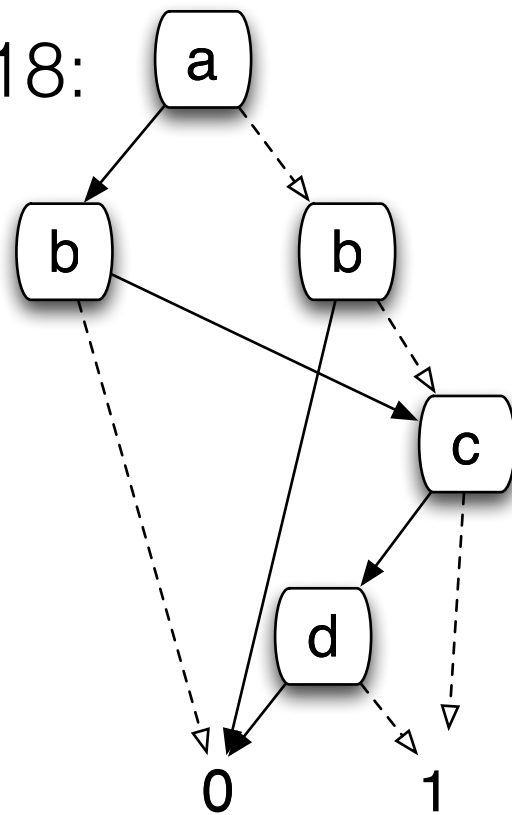
- Canonical form exists:
two structures are equivalent if they are equal.
- Drawback: usually very large structures

Binary Decision Diagram (BDD)

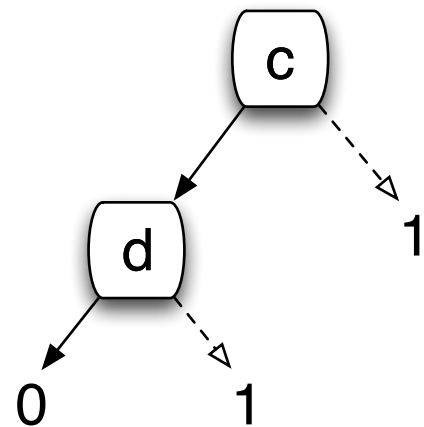
- DAG with the following nodes
 - Constant 0
 - Constant 1
 - If-then-else with a variable as condition
- Rules for Ordered-BDD:
 - Variables are ordered, gates must occur in that order:
if $a > b > c > d > e > f$, then the “if c then .. else ..” gate can contain gates with b and a, but not with d, e and f.
- Rules for Reduced-BDD:
 - All gates must be different (no two gates with the same variables and inputs, e.g. “if x then y else z”)
 - A gate cannot have the same “then” and “else” clause
- Theorem: if a BDD is Reduced AND Ordered, it is canonical.

Binary Decision Diagram (BDD)

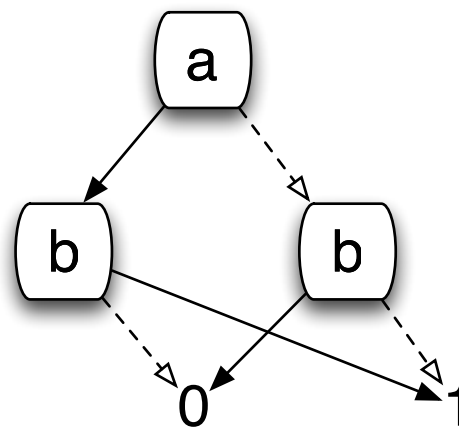
- AIG node 18:



- AIG node 15:



- AIG node 17:

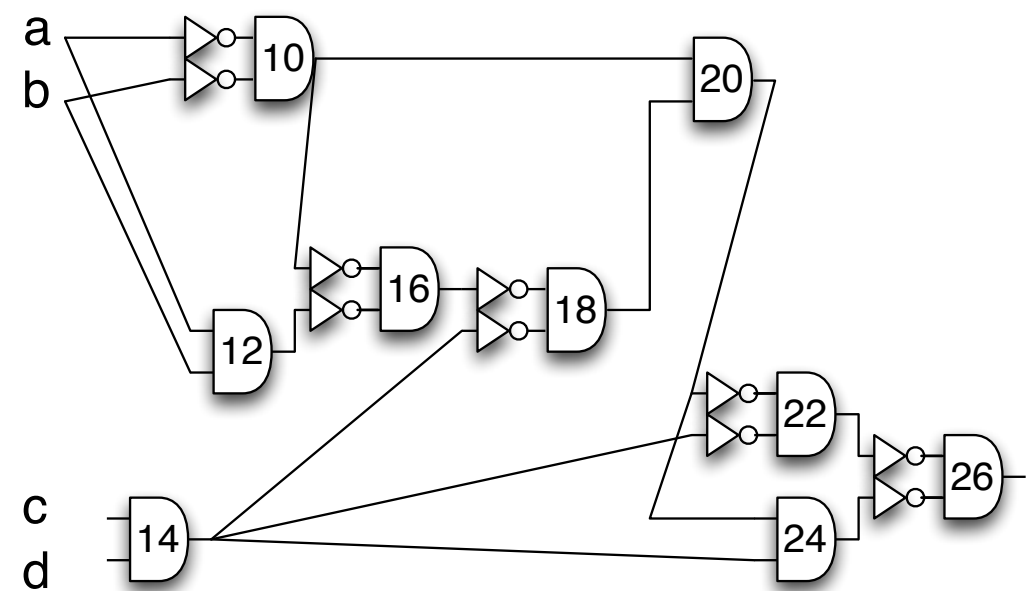


Binary Decision Diagram (BDD)

- Canonical form: ROBDD
- AND of two BDDs introduces blowup
- Used in model checkers
 - Usually SAT-based (CNF/AIG) model checking is faster
 - Not always
- We will come back with more details about BDD's later when we will talk about Symbolic Model Checking

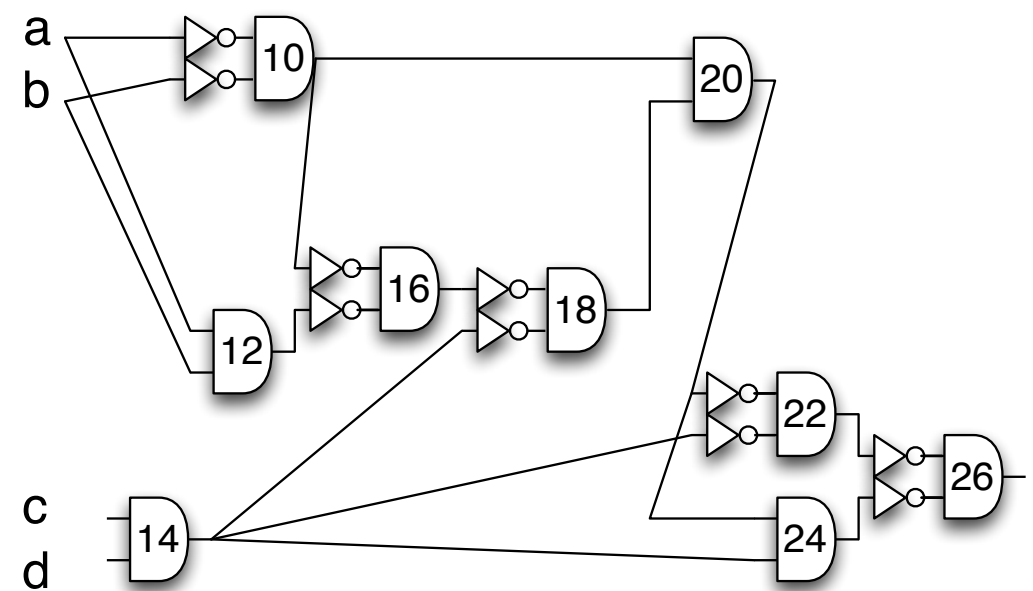
Data structures, so far:

- DAG
- AIG
- CNF
- BDD
- Up next: Sum of products



Sum of products

- A lot like CNF, but operations are chosen such that a SOP is canonical.
- Most common choice of operations: AND (product) + XOR (sum)
- AND is innermost, XOR is outermost operation



Sum of products

- not a: True XOR a

AND of {}

AND of {a}

- (True XOR a) & (True XOR b)

=

True & True

XOR

True & b

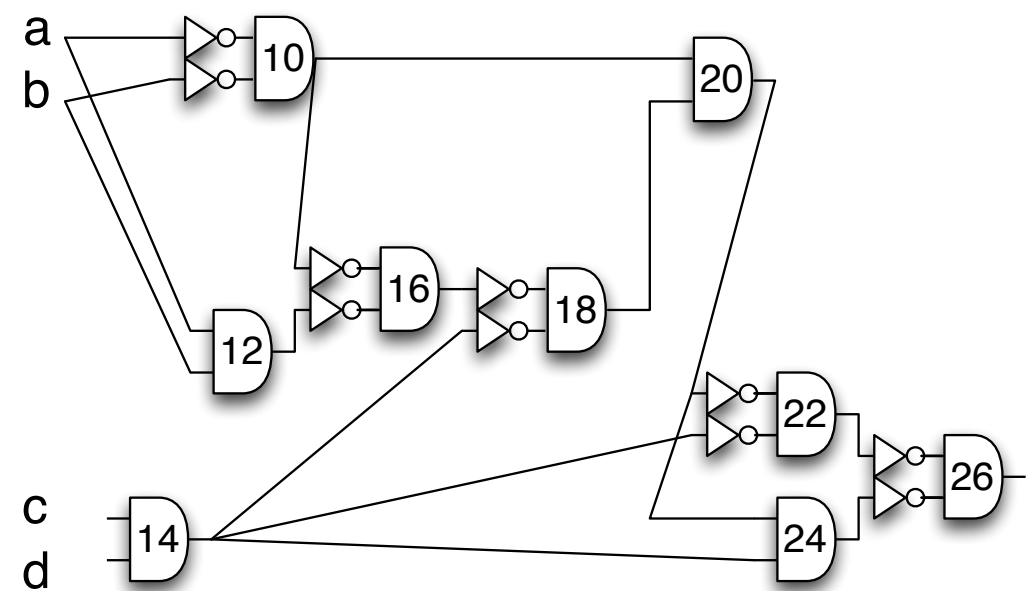
XOR

a & True

XOR

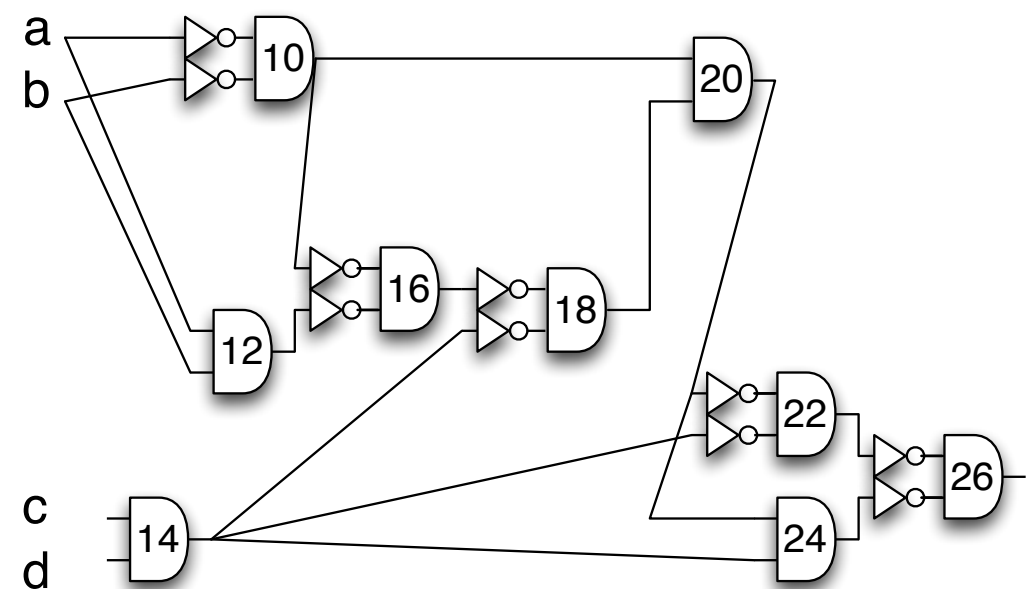
a & b

= True XOR a XOR b XOR a&b



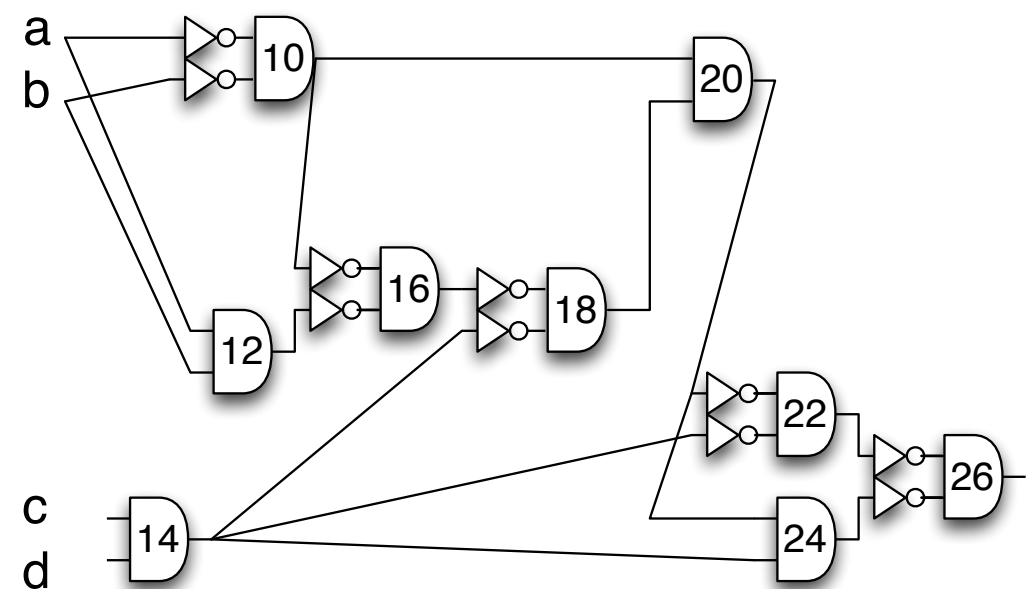
Sum of products (SOP)

- Unlike CNF, do not introduce helper variables
- Negation of x is simply “ x XOR 1”
- SOP is canonical, if AND- and XOR- clauses are considered as sets:
 - Sort variables within AND clause, no duplicates
 - Sort variable-sets within XOR clause, no duplicates
- AND of two SOPs introduces blowup



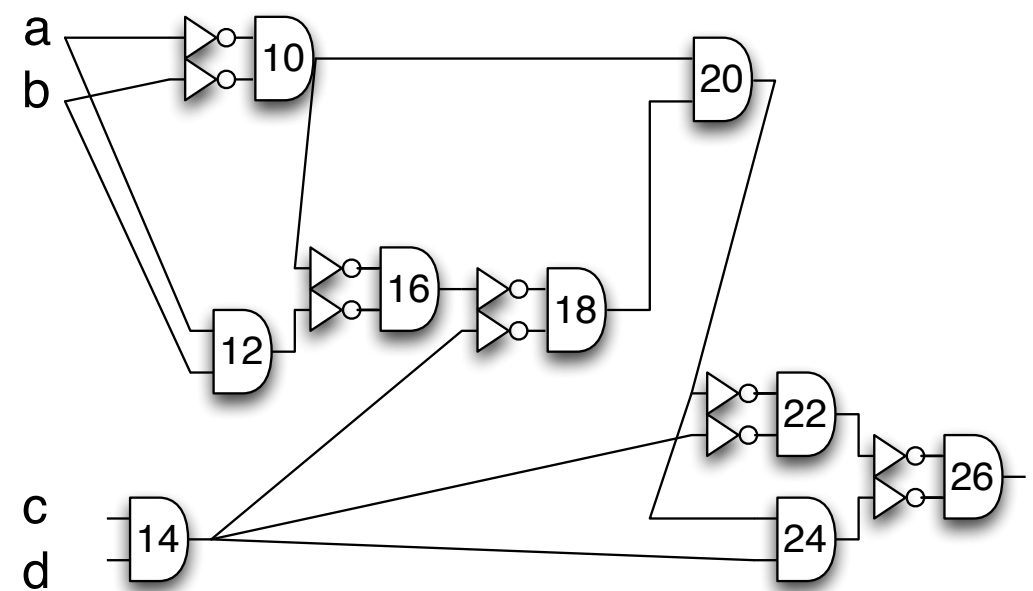
Sum of products (SOP)

- 10: True XOR a XOR b XOR a&b
12: a&b
11: a XOR b XOR a&b
13: True XOR a&b
16: True&a XOR True&b XOR True&a&b
XOR a&b&a XOR a&b&b XOR a&b&a&b
= a XOR b XOR a&b XOR a&b XOR a&b XOR a&b
= a XOR b
17: True XOR a XOR b
14: c&d
15: True XOR c&d
18: True XOR a XOR b XOR
c&d XOR c&d&a XOR c&d&b

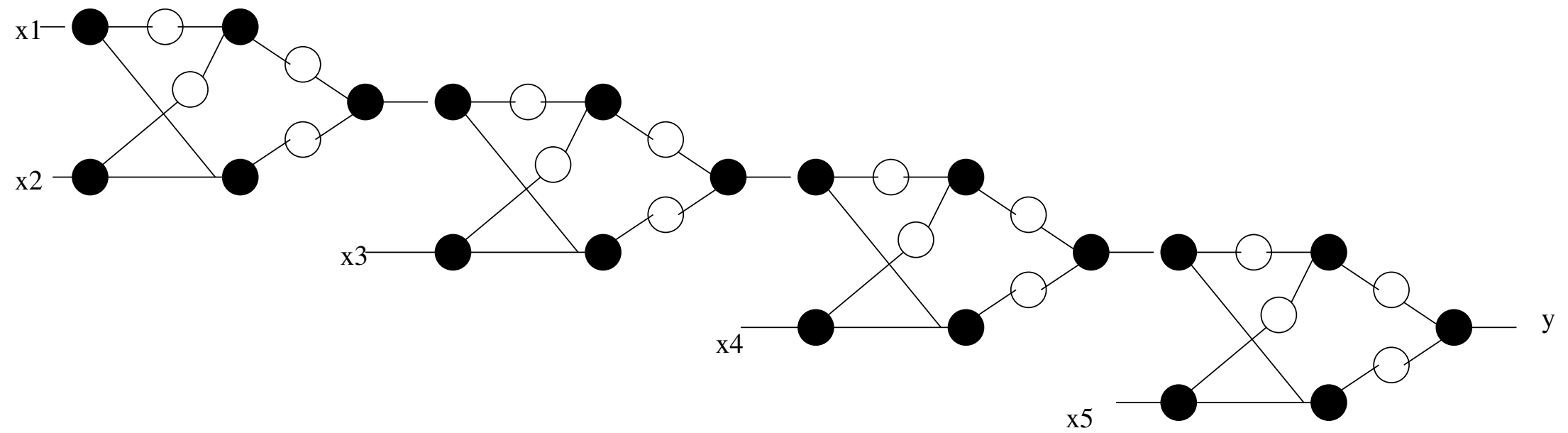
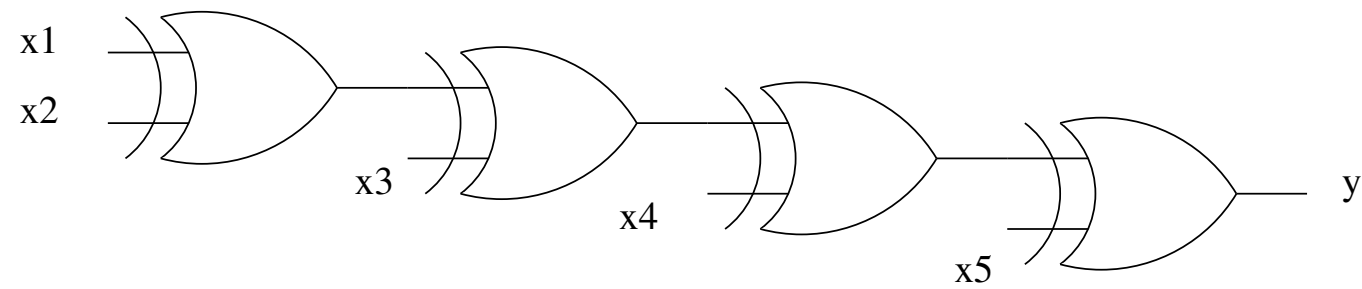


Datastructures, so far:

- DAG
- AIG
- CNF
- BDD
- SOP
- Up next: AIG (again!)



AIGs vs. SoP



AIGs vs. BDDs

- » AIGs always size proportional to input
- » BDDs always exponential size for some cases
 - (e.g. multiplier circuits)

SAT solving

- » Once we have Boolean functions, we can do SAT solving.
 - » This is an NP-Hard problem, but efficient in practice
 - » At the basis of almost all modern FV methods.
-
- » At the next lecture, we will go through the basic algorithm for SAT solving.

CEC with SoP

- » SoP is a normal form
- » CEC obtained by normalising expressions to SoP
- » Then check for syntactic equality

CEC with BDDs

- » ROBDDs is a normal form.
- » Compute the two ROBDDs.
- » Check for syntactic equality.

CEC with SAT and CNF

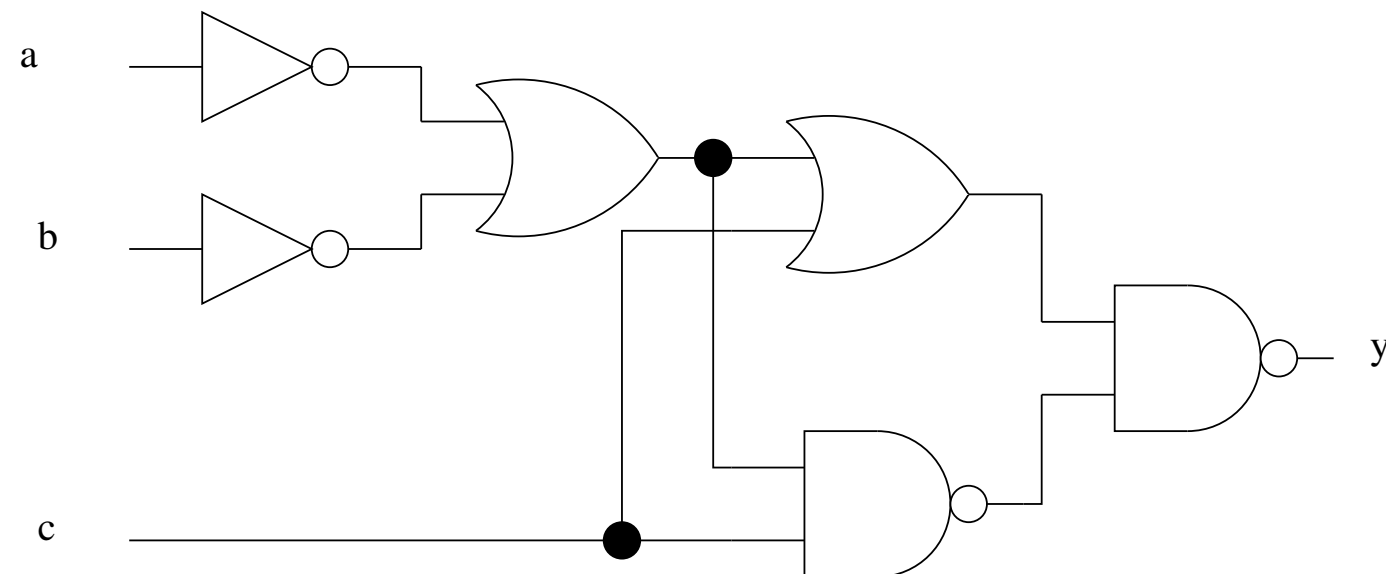
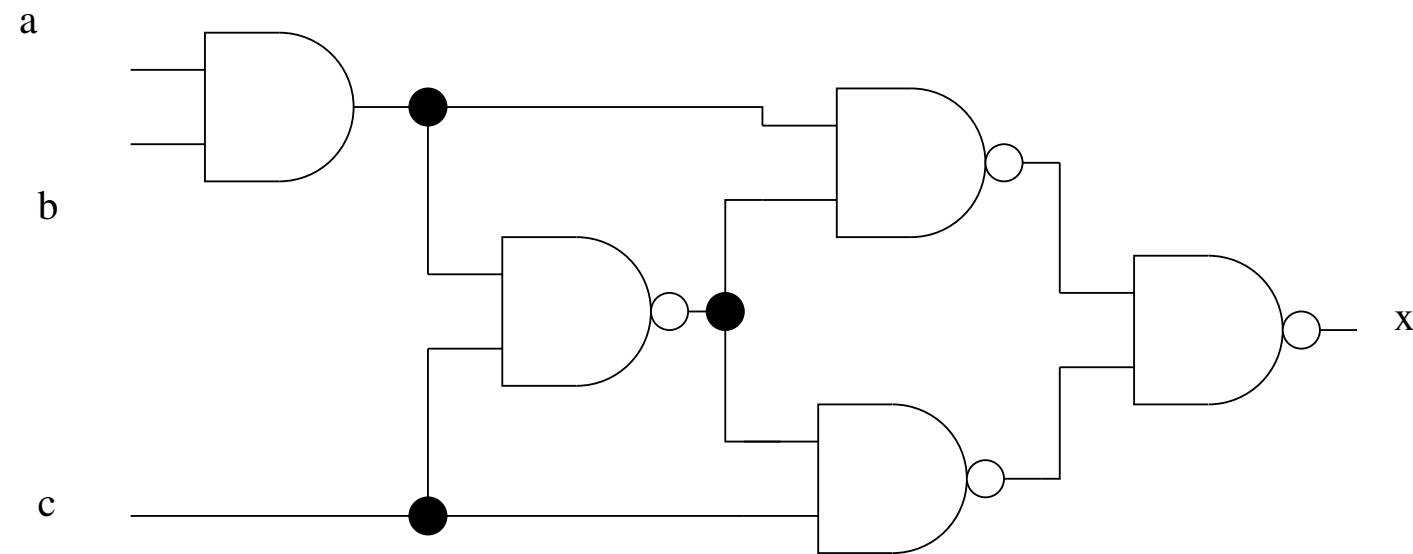
- » Take two circuits
 - » Create a CNF representation of each one of them
 - » XOR all outputs pairwise
 - » Assert one XOR output is 1
-
- » Look at code skeleton for assignment 1

CEC with AIGs

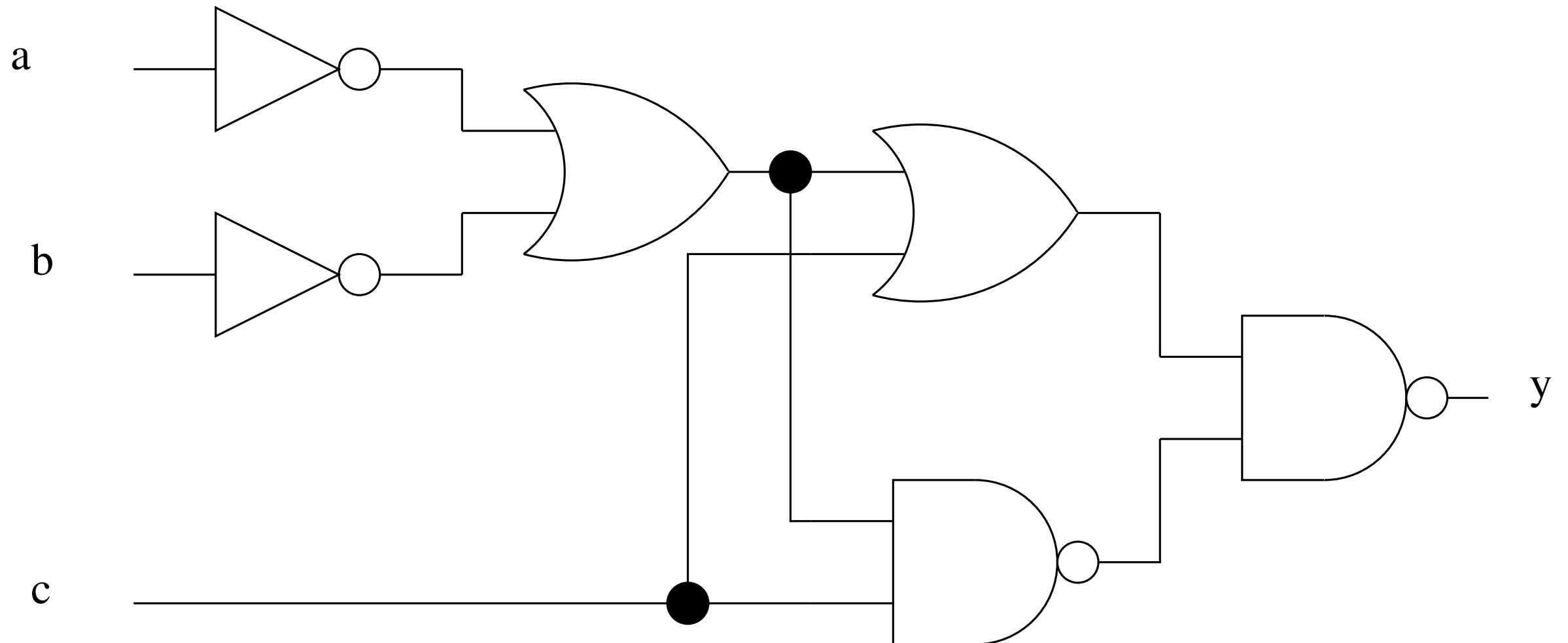
- » Step 1: random simulation
- » Step 2: build AIG
- » Step 3: SAT sweeping

(slides taken from Sean Weaver, see course webpage)

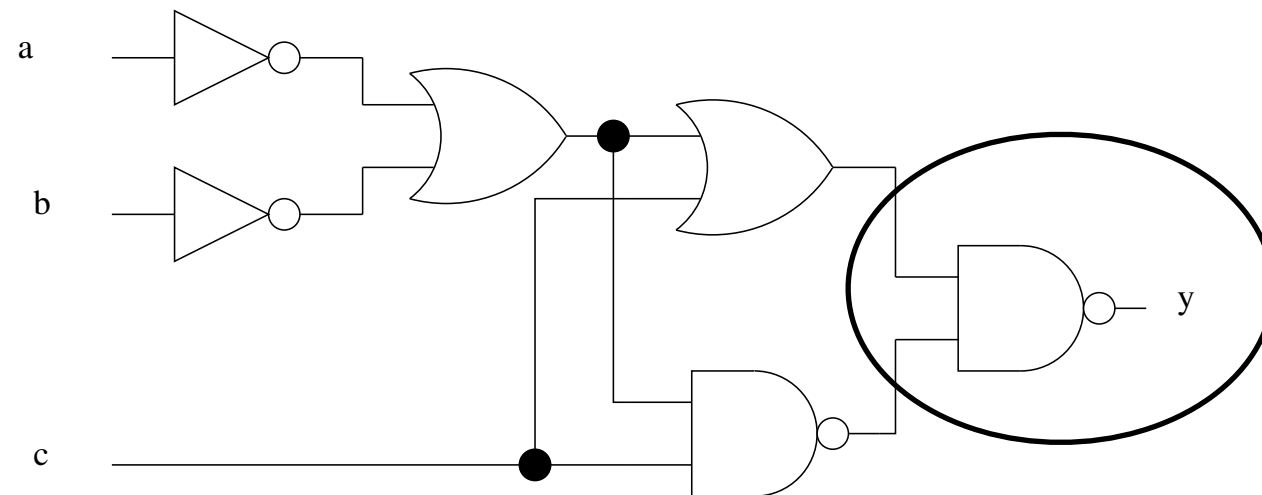
Equivalence Checking



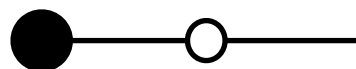
Another example (1)



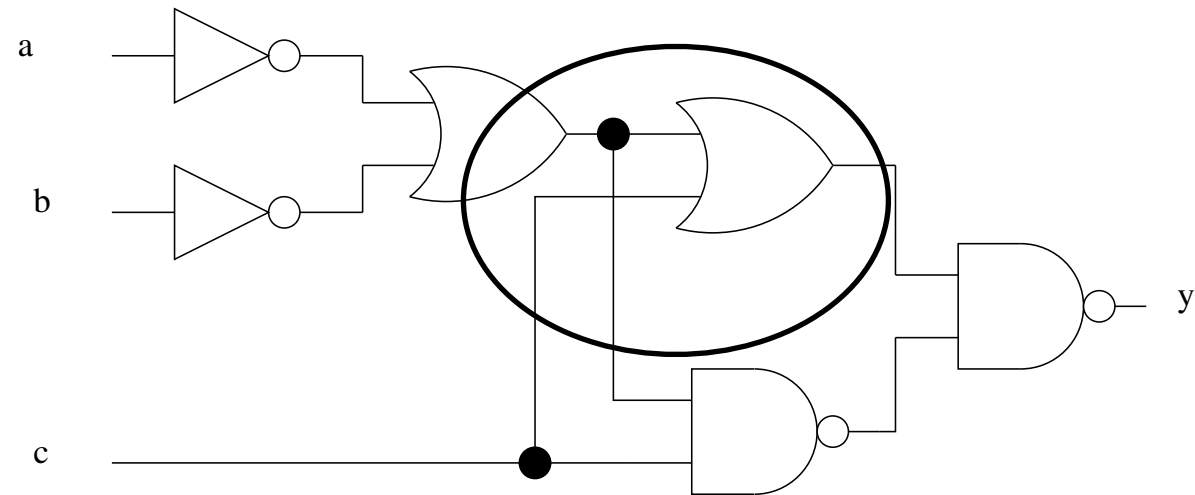
Another example (2)



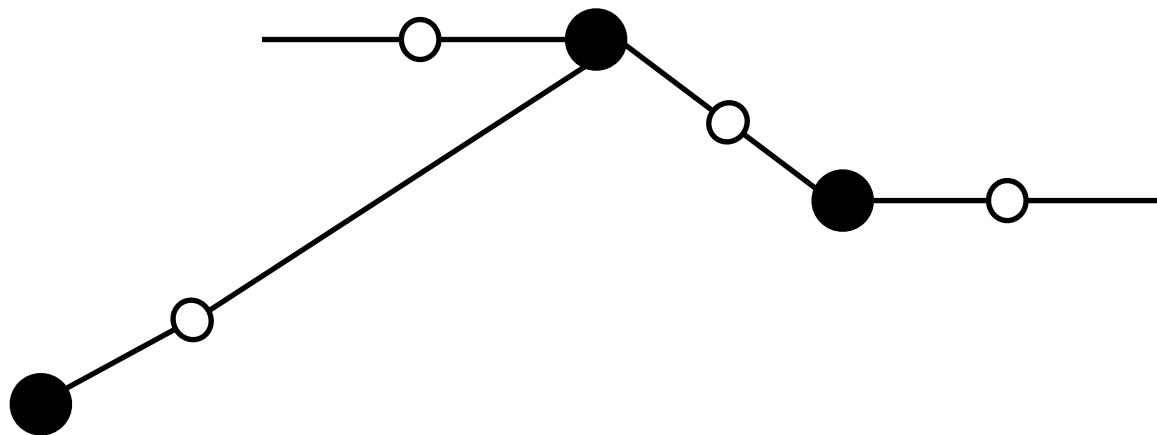
1 nand-gate to start



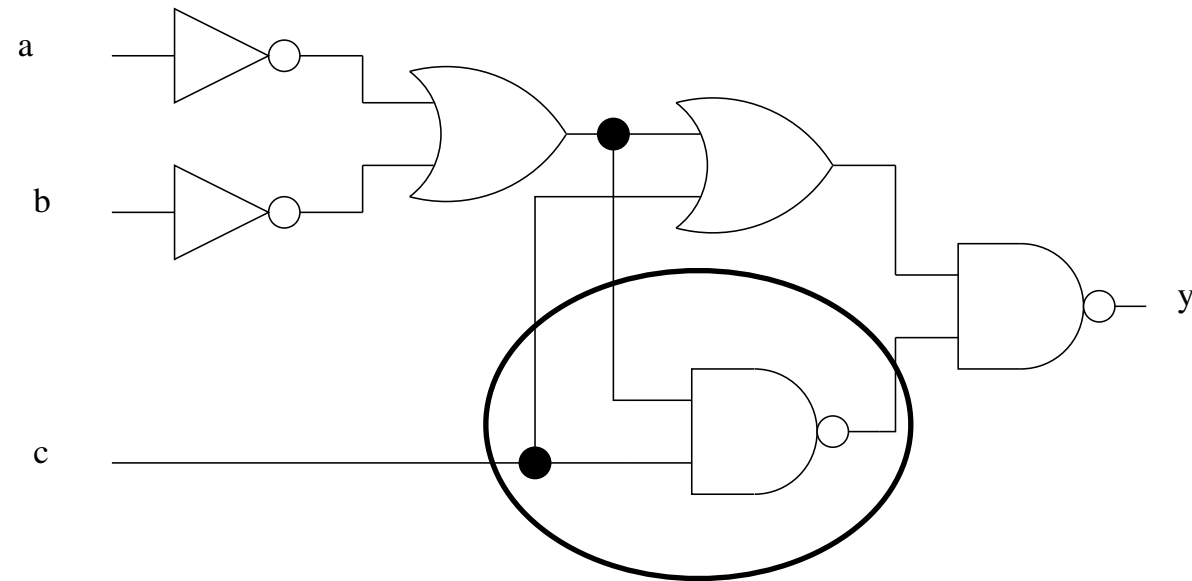
Another example (3)



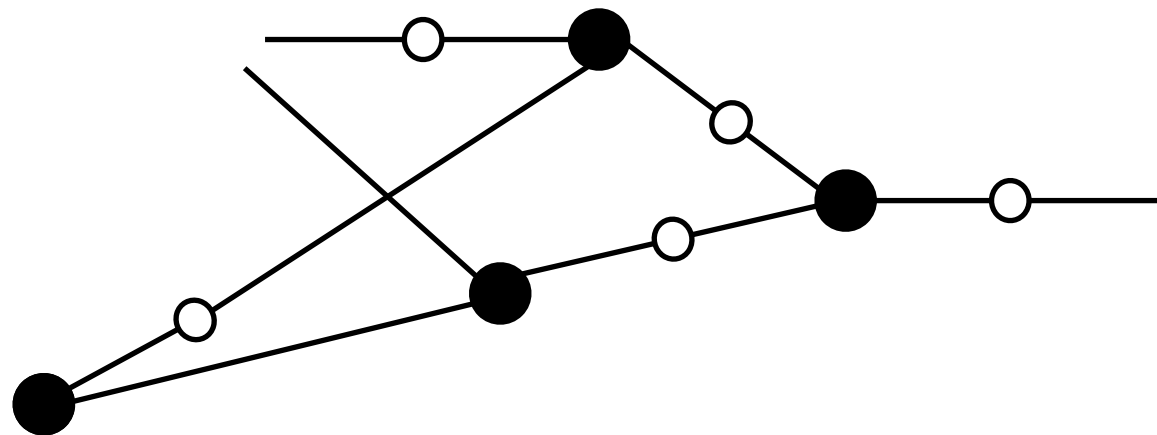
1 or gate



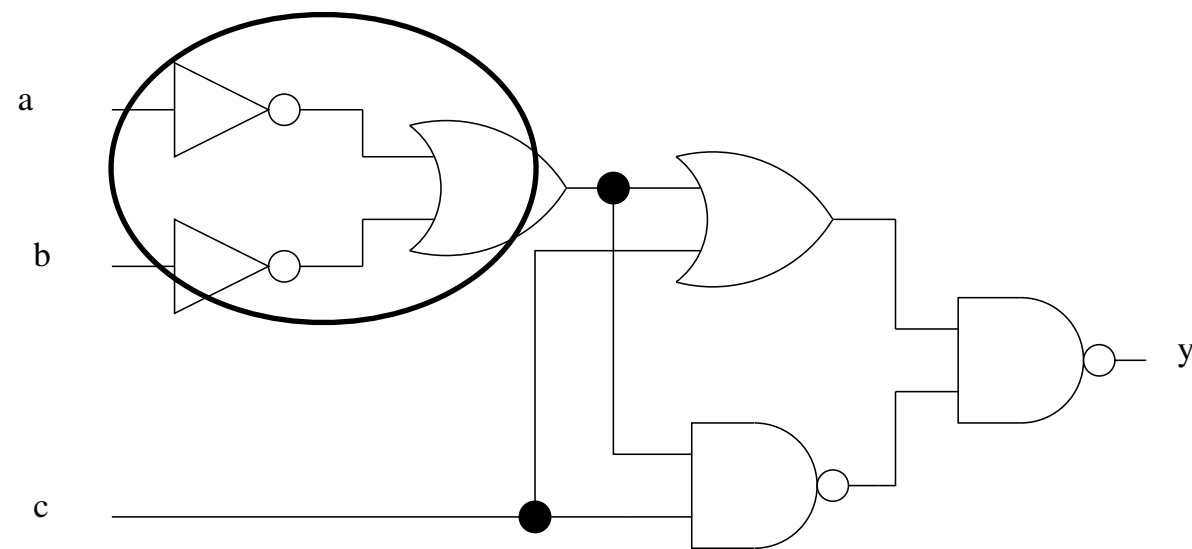
Another example (3)



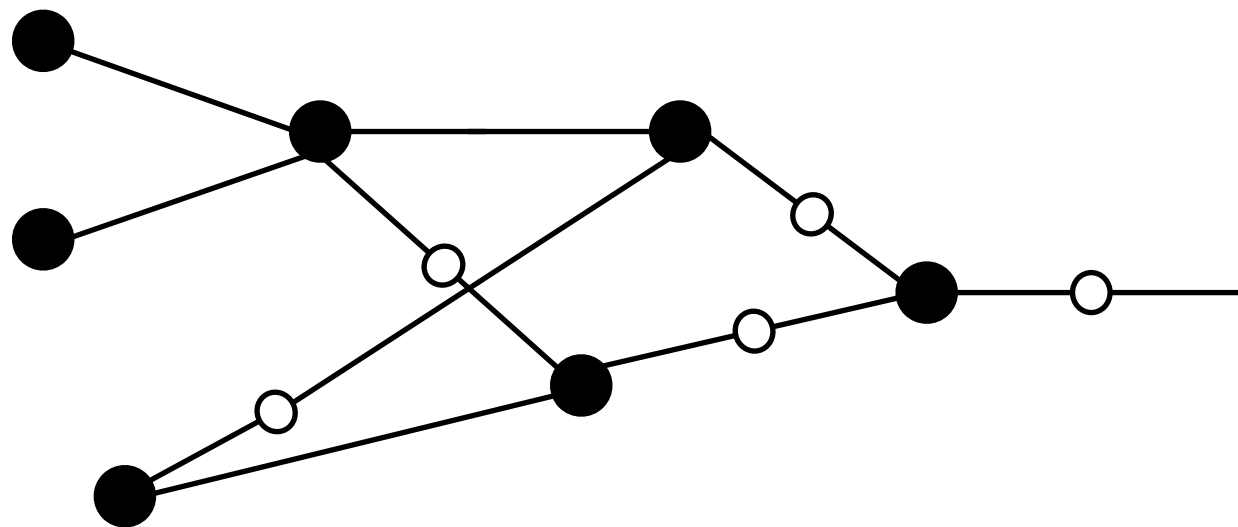
1 nand gate



Another example



finally 1 or gate with negated inputs



AIGs

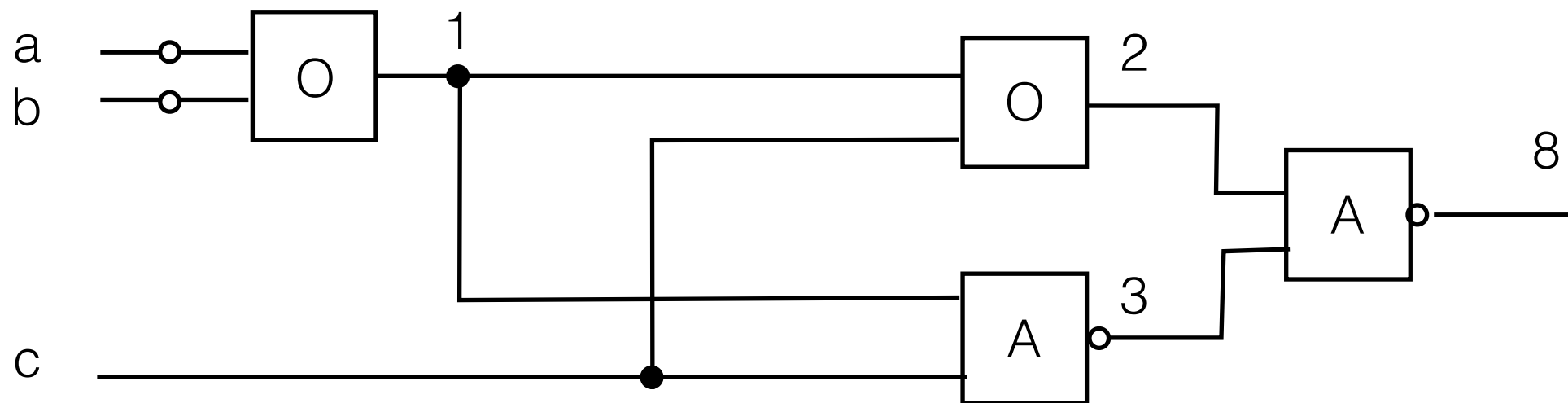
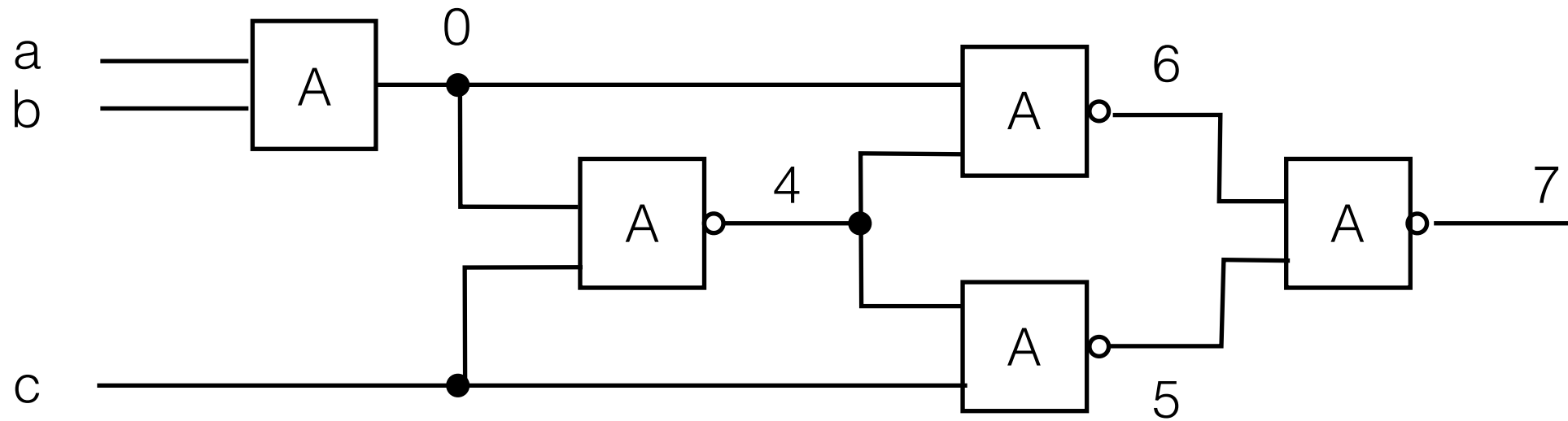
» Pros

- simple to build and manipulate
- unifying among synthesis, verification, technology mapping
- compact representation

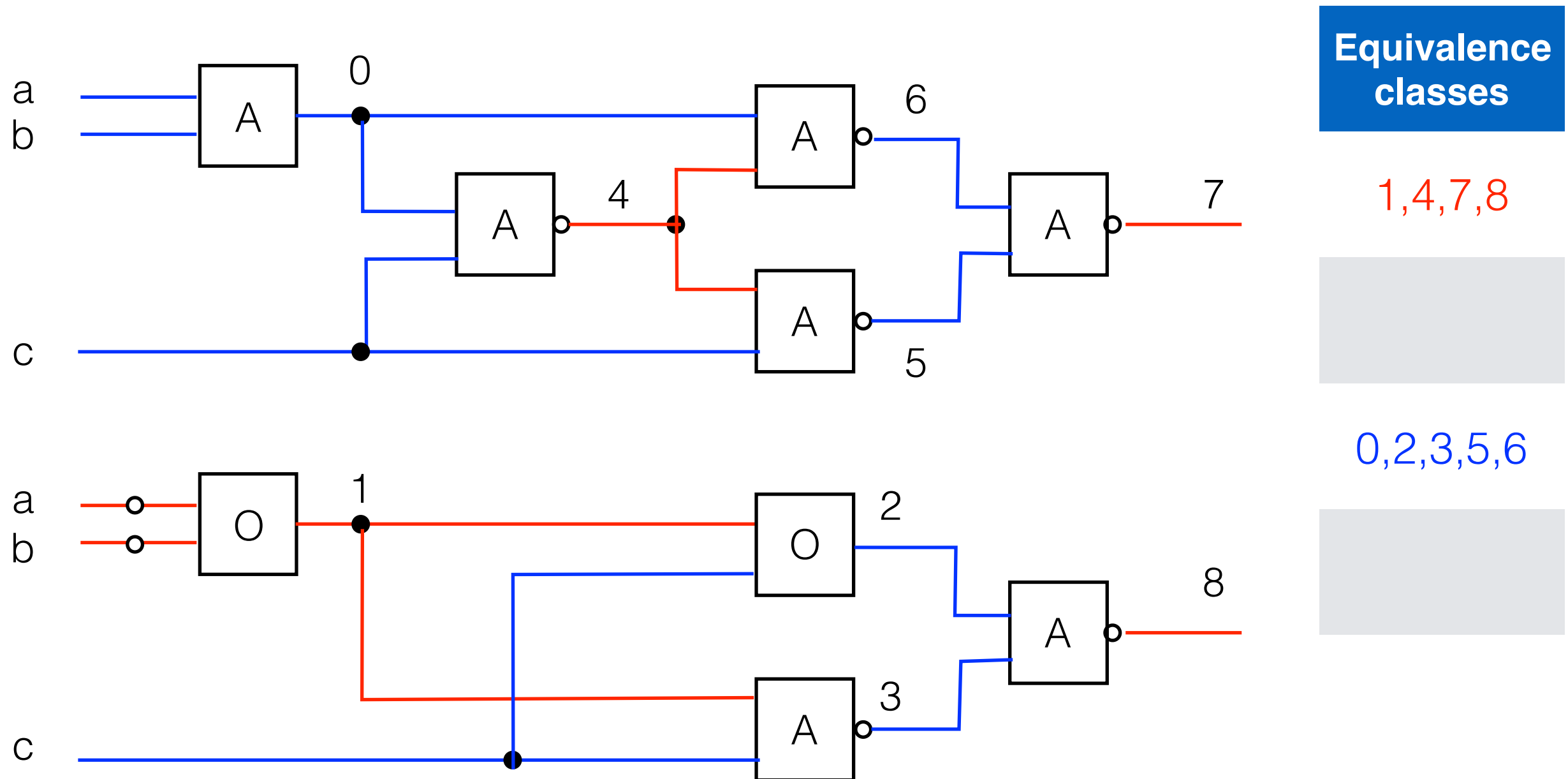
» Cons

- structurally not efficient (see FRAIG)
- non canonical

Equivalence Checking

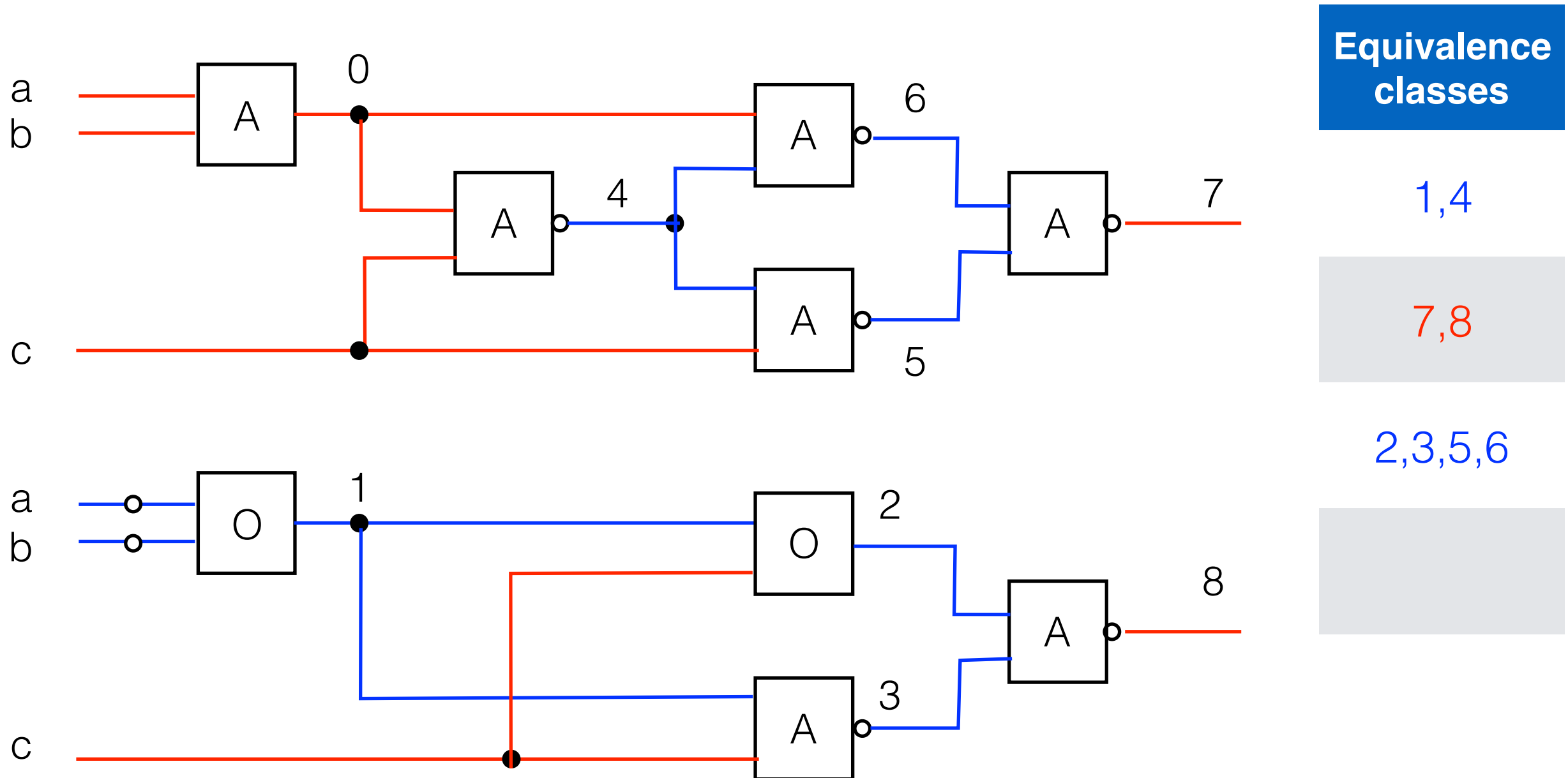


Random Simulation (1)



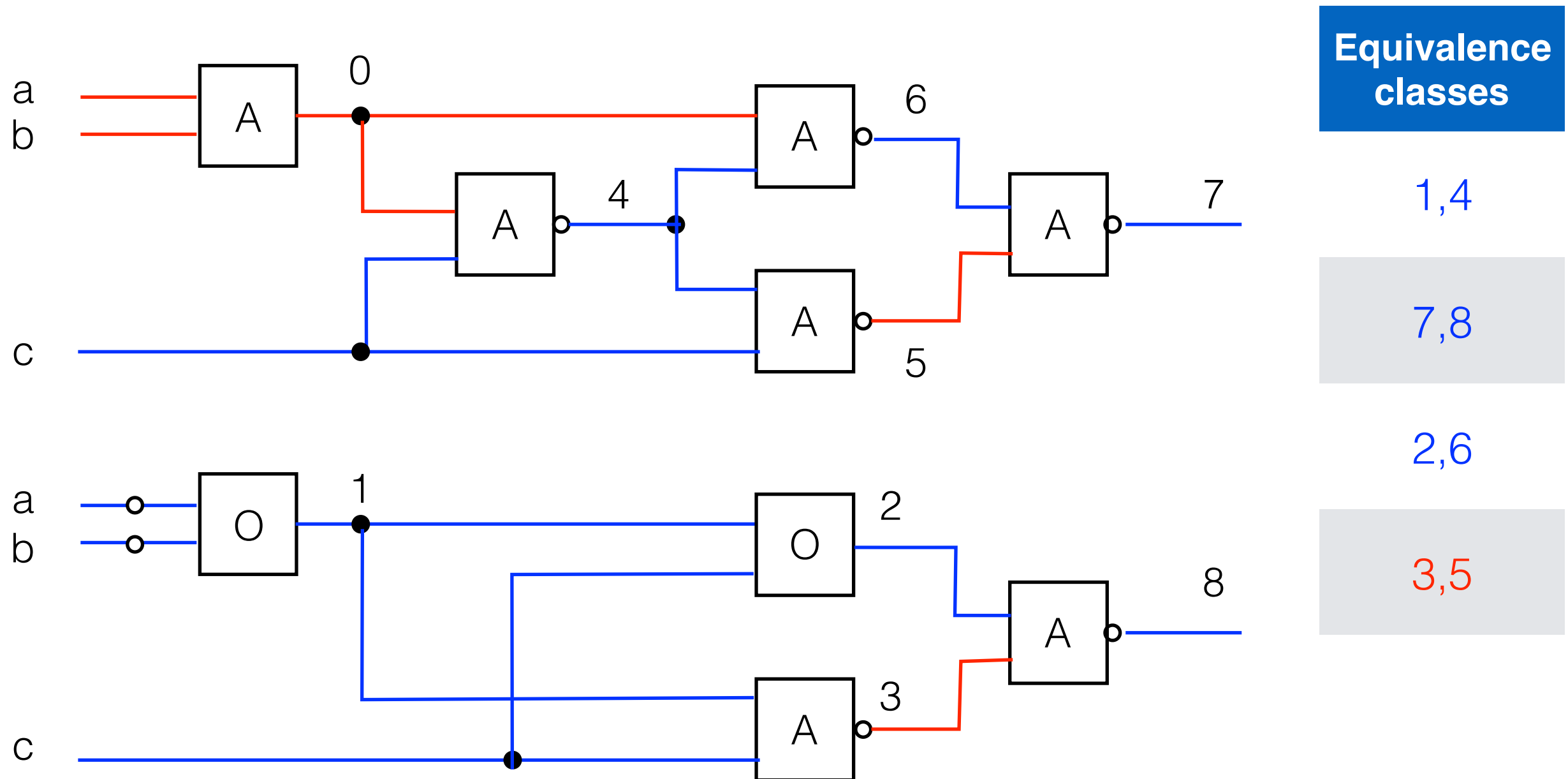
Random Vector: assign T to all inputs. $a = b = c = T$

Random Simulation (2)



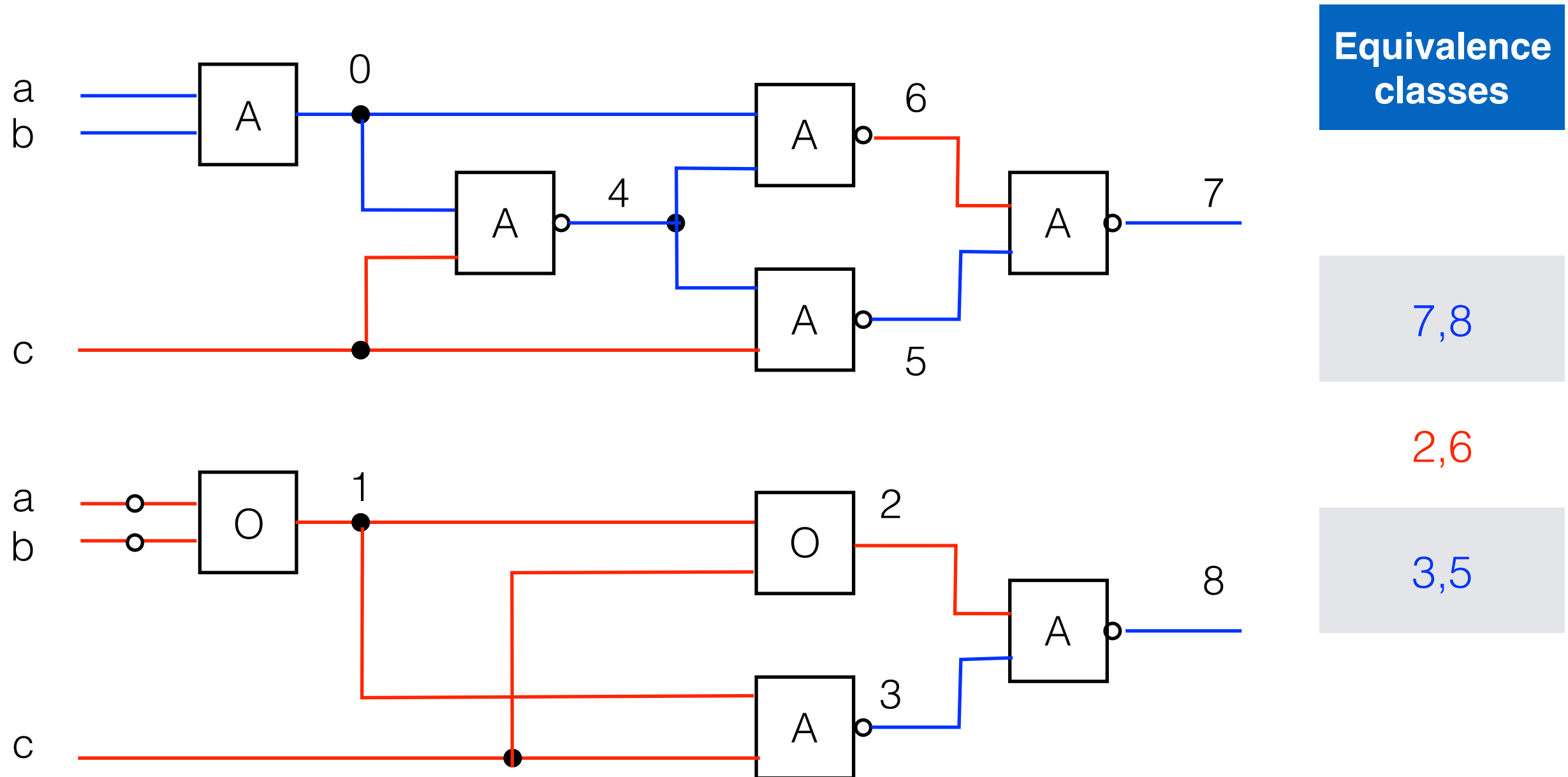
Random Vector: assign F to all inputs. $a = b = c = F$

Random Simulation (3)



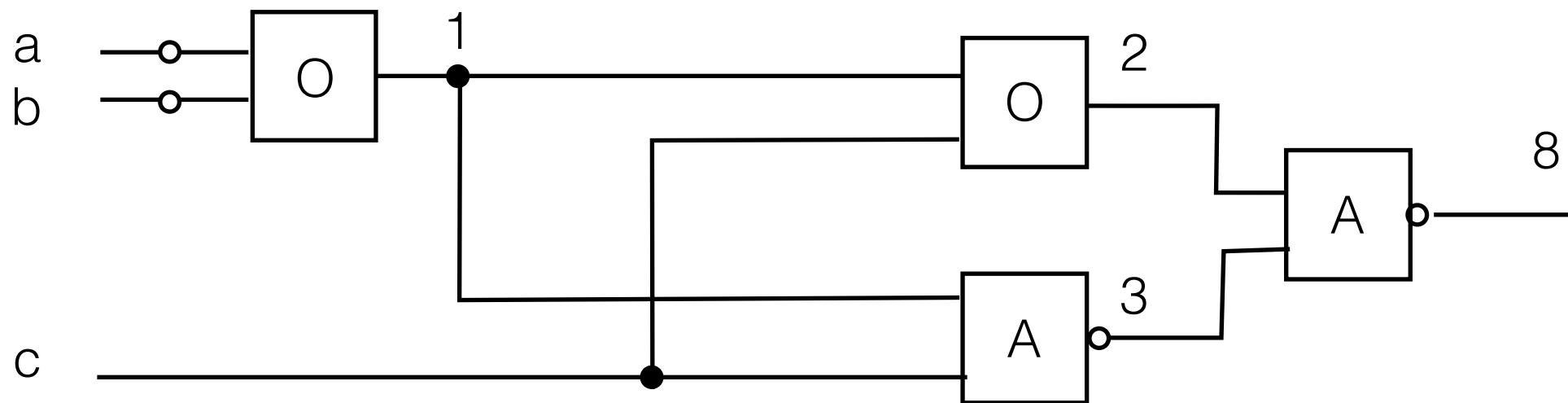
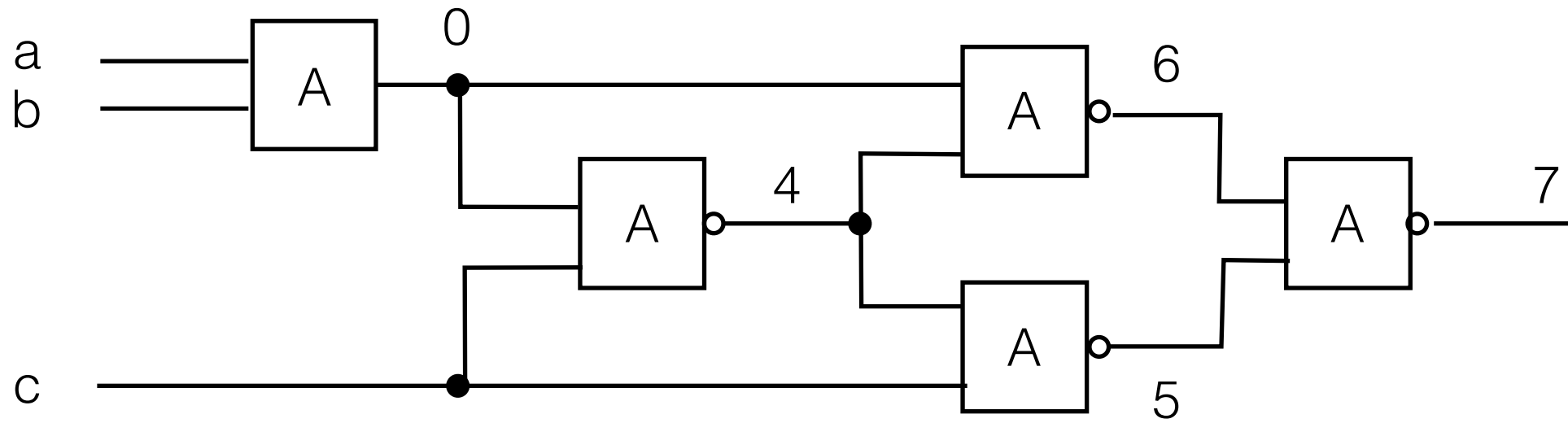
Random Vector: $a = b = \text{F}$ and $c = \text{T}$

Random Simulation (4)

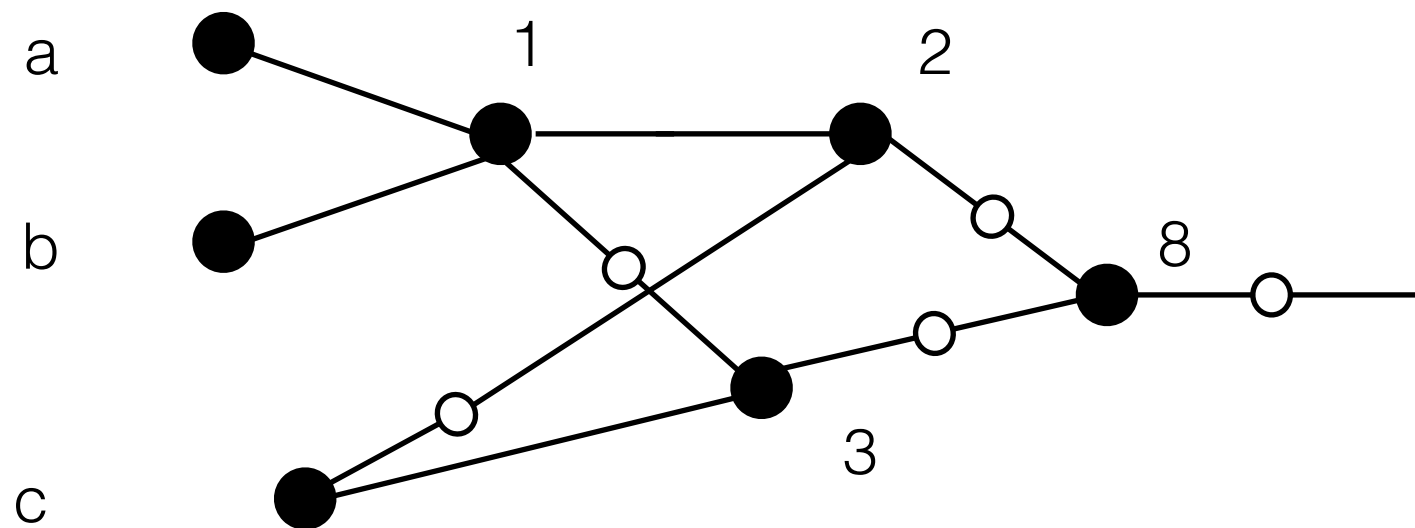
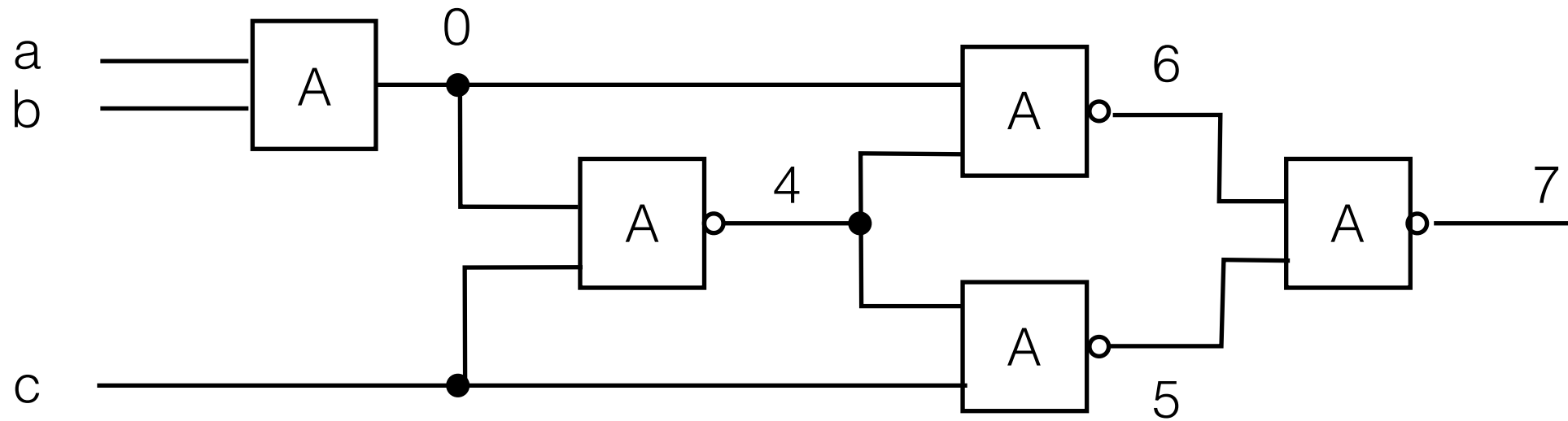


Random Vector: $a = b = \text{blue T}$ and $c = \text{red F}$

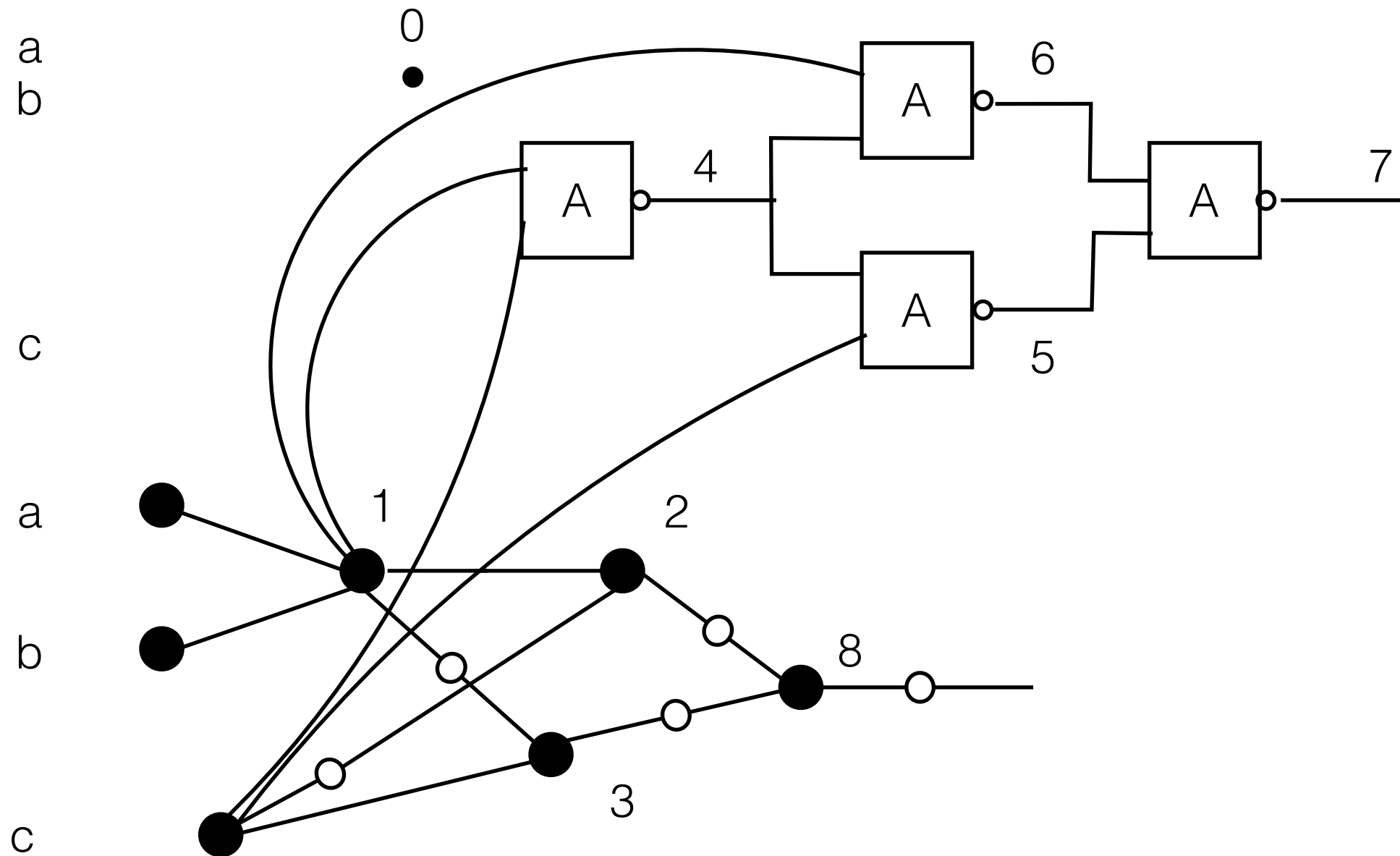
AIG (1)



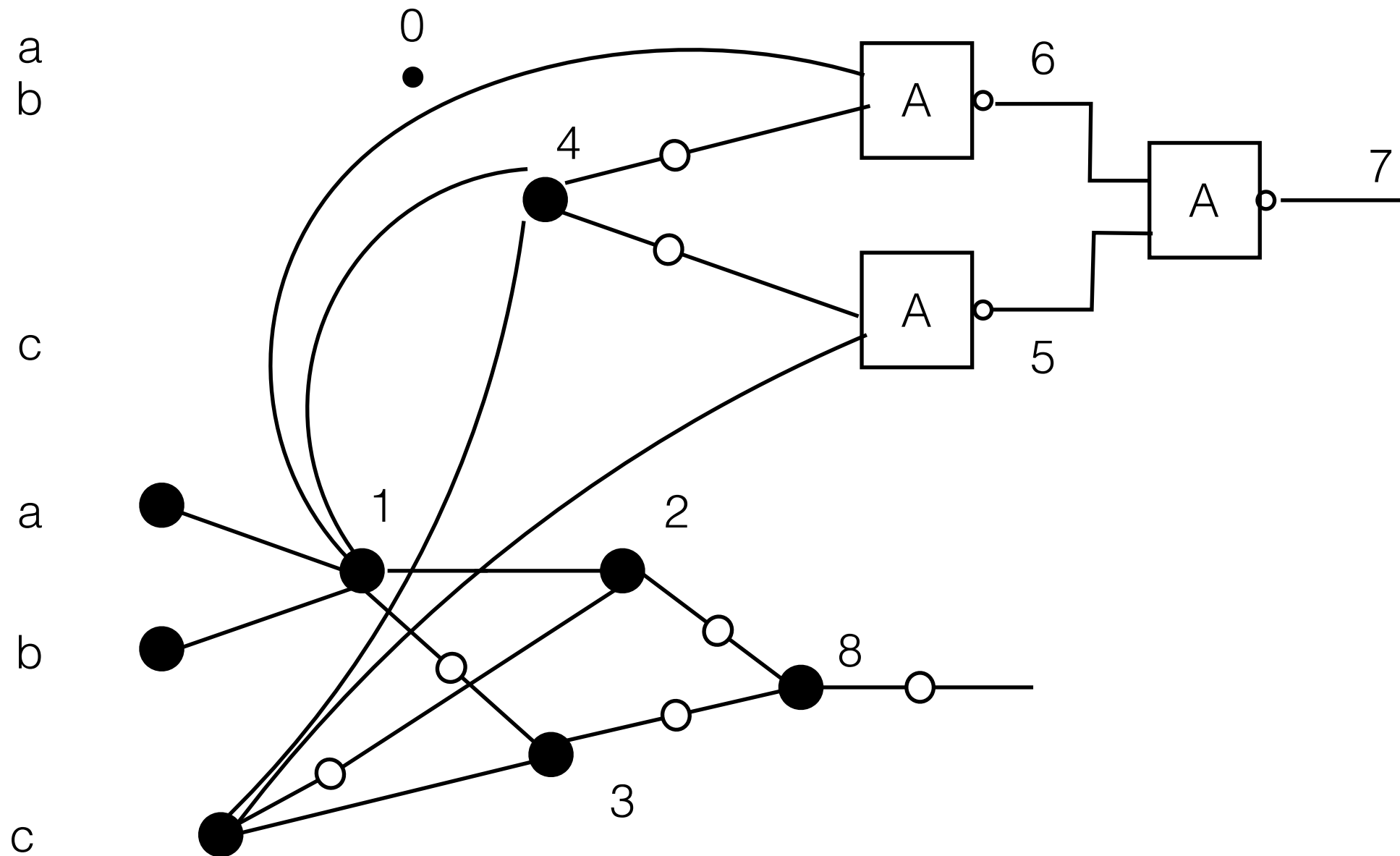
AIG (2)



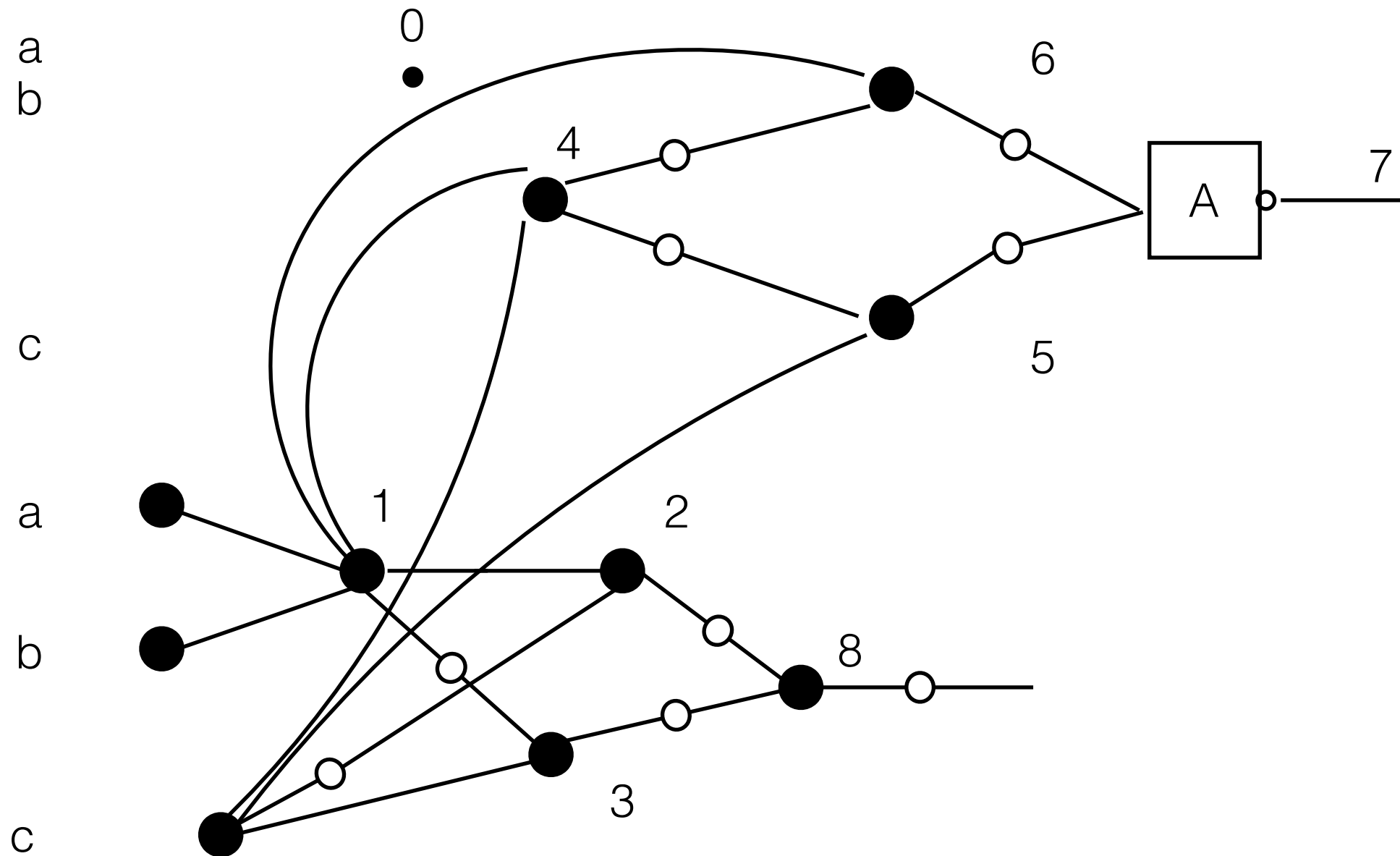
AIG (3)



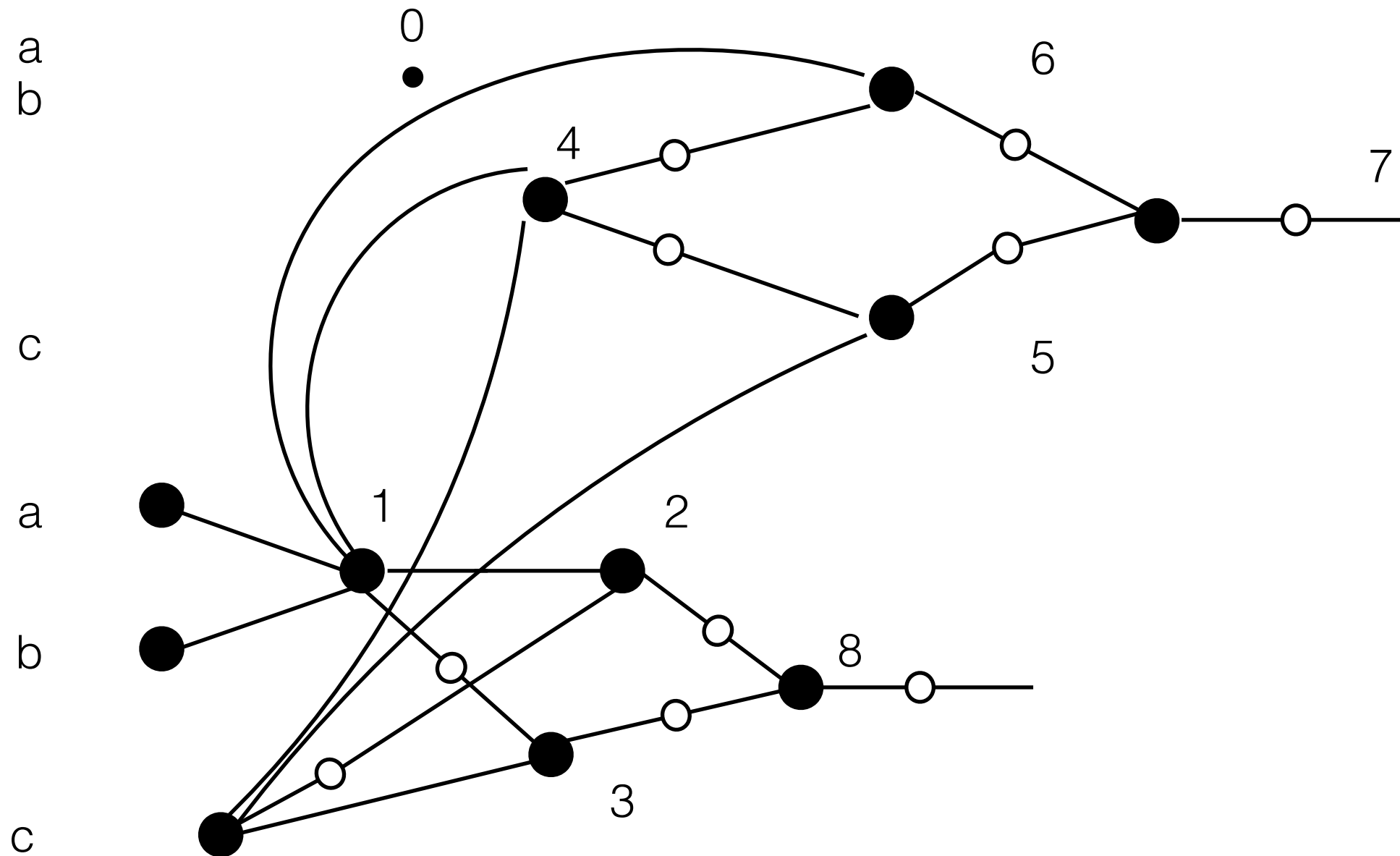
AIG (4)



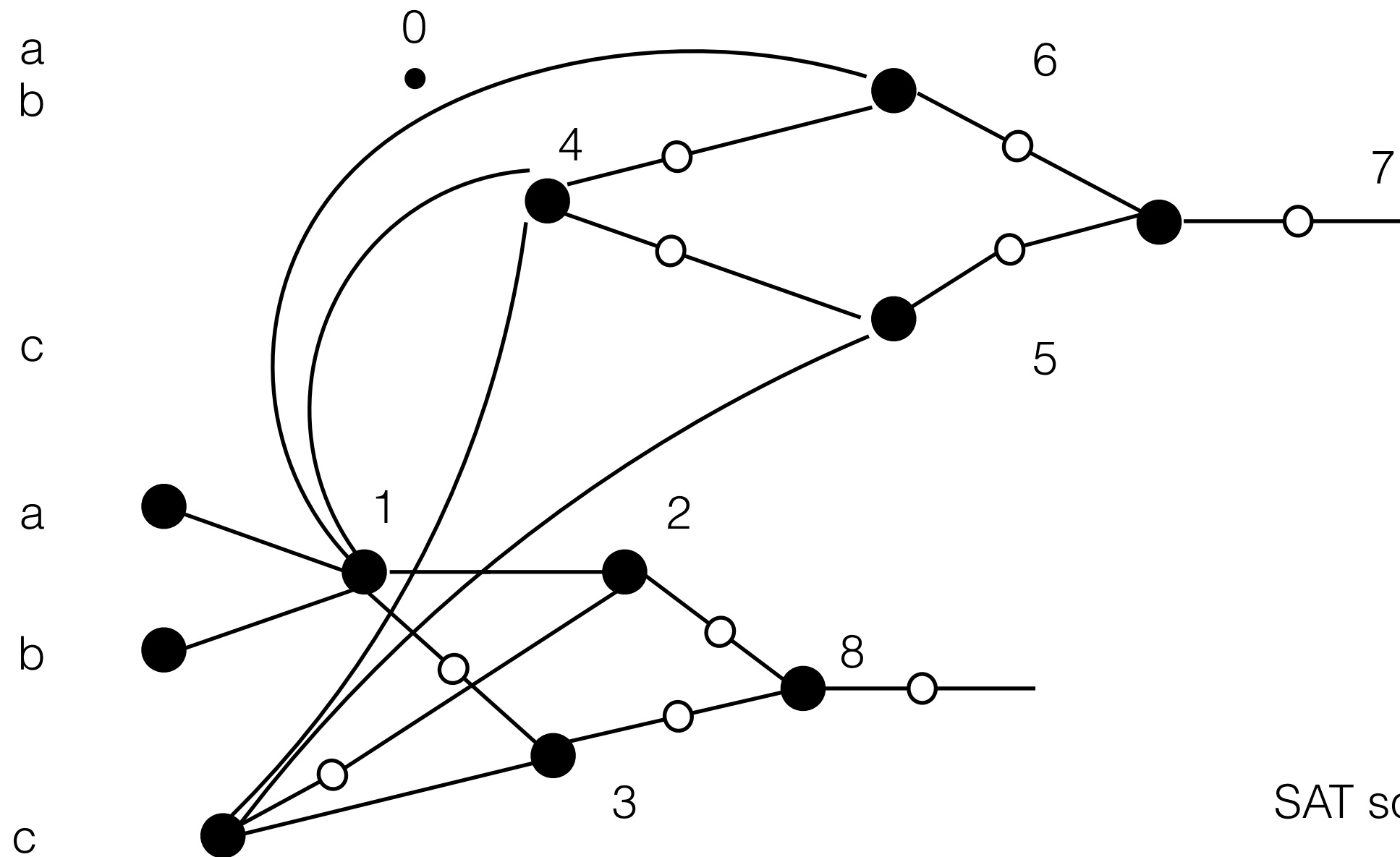
AIG (4)



AIG (5)



SAT Sweeping (1)



Equivalence classes

7,8

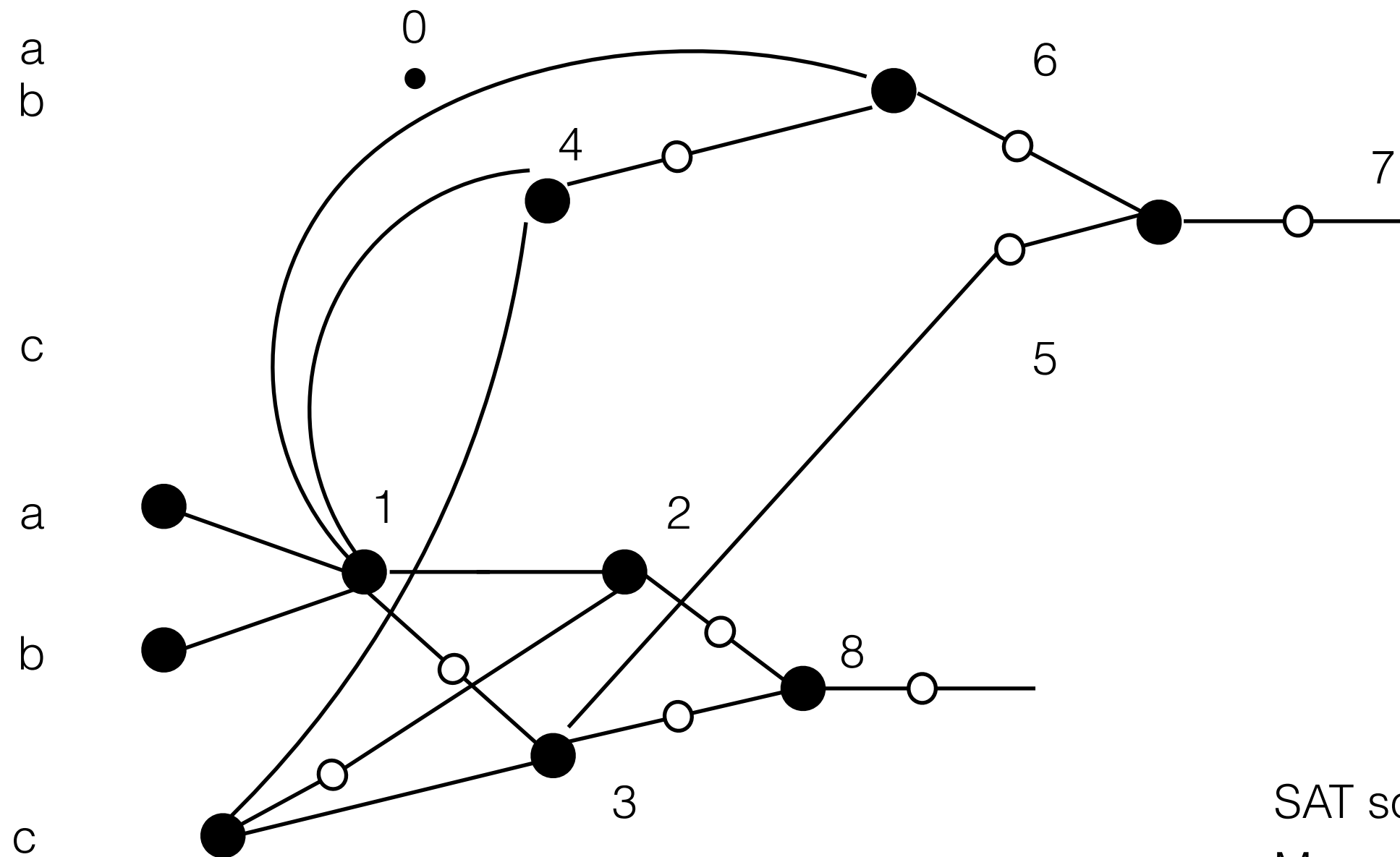
2,6

3,5

SAT solver: 3 = 5

$$\neg(a \wedge b) \wedge c = \neg((a \wedge b) \wedge c) \wedge c$$

SAT Sweeping (2)



Equivalence classes

7,8

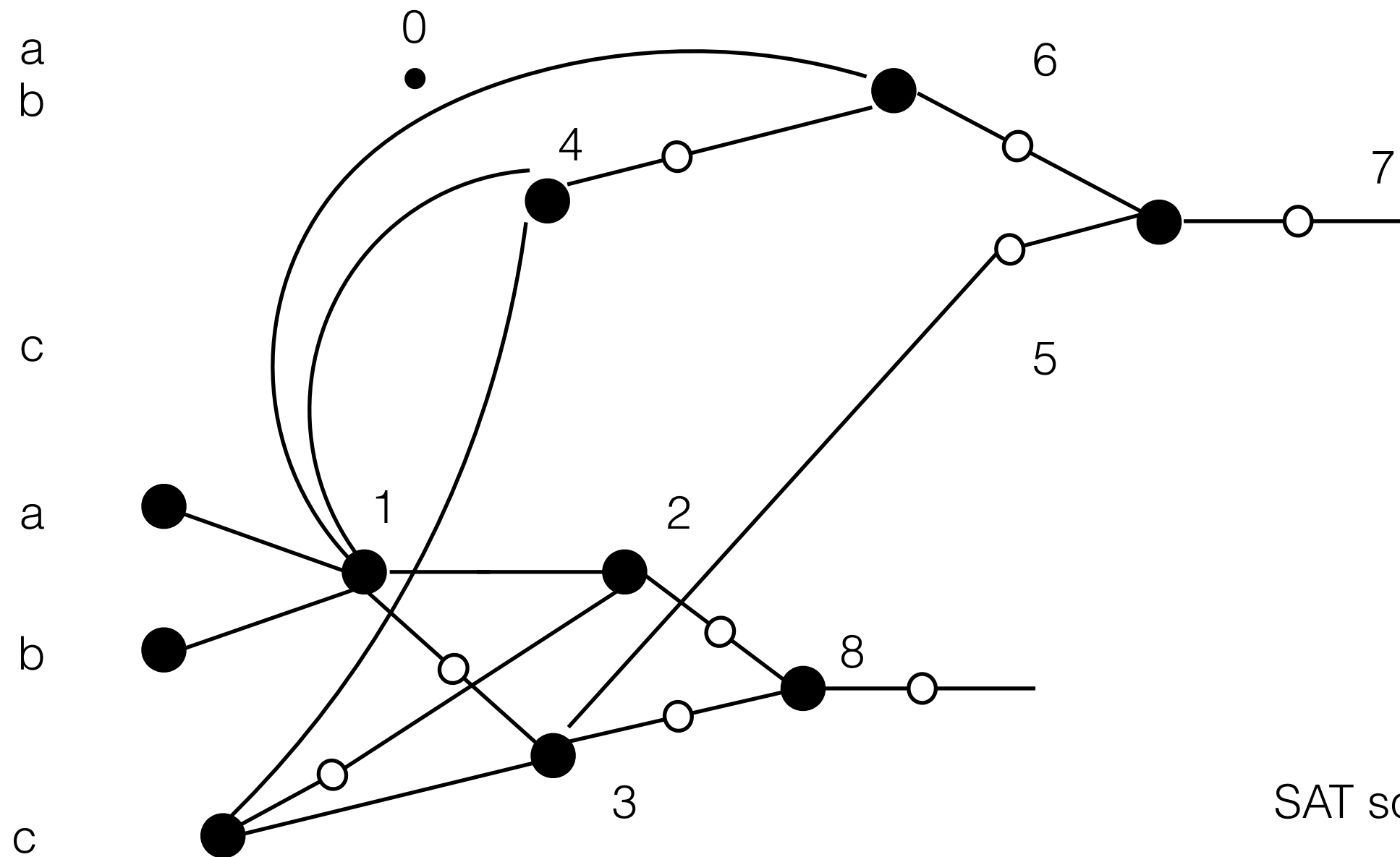
2,6

3,5

SAT solver: $3 = 5$

Merge nodes 3 and 5

SAT Sweeping (3)



Equivalence classes

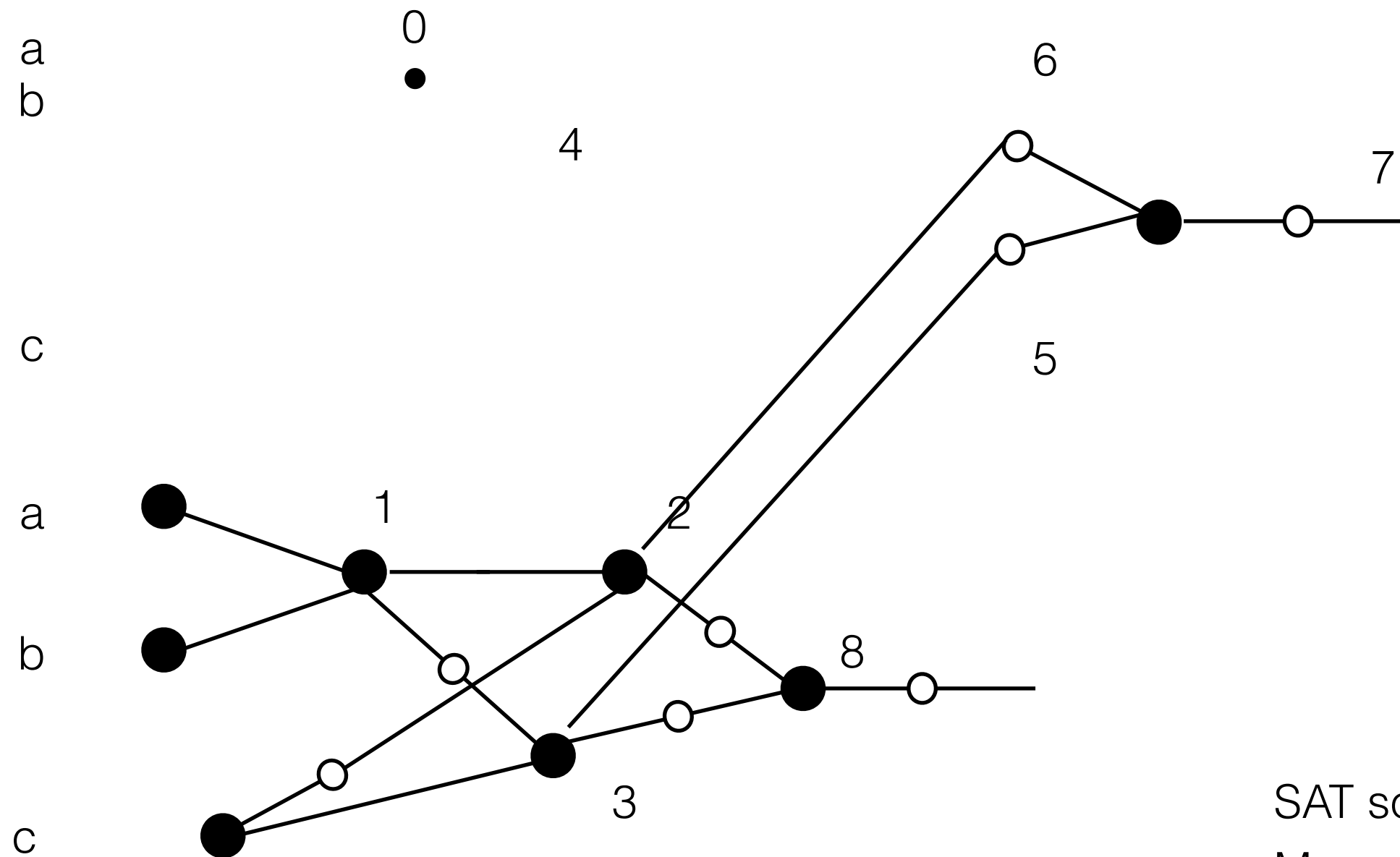
7,8

2,6

3,5

SAT solver: $2 = 6$

SAT Sweeping (4)



Equivalence classes

7,8

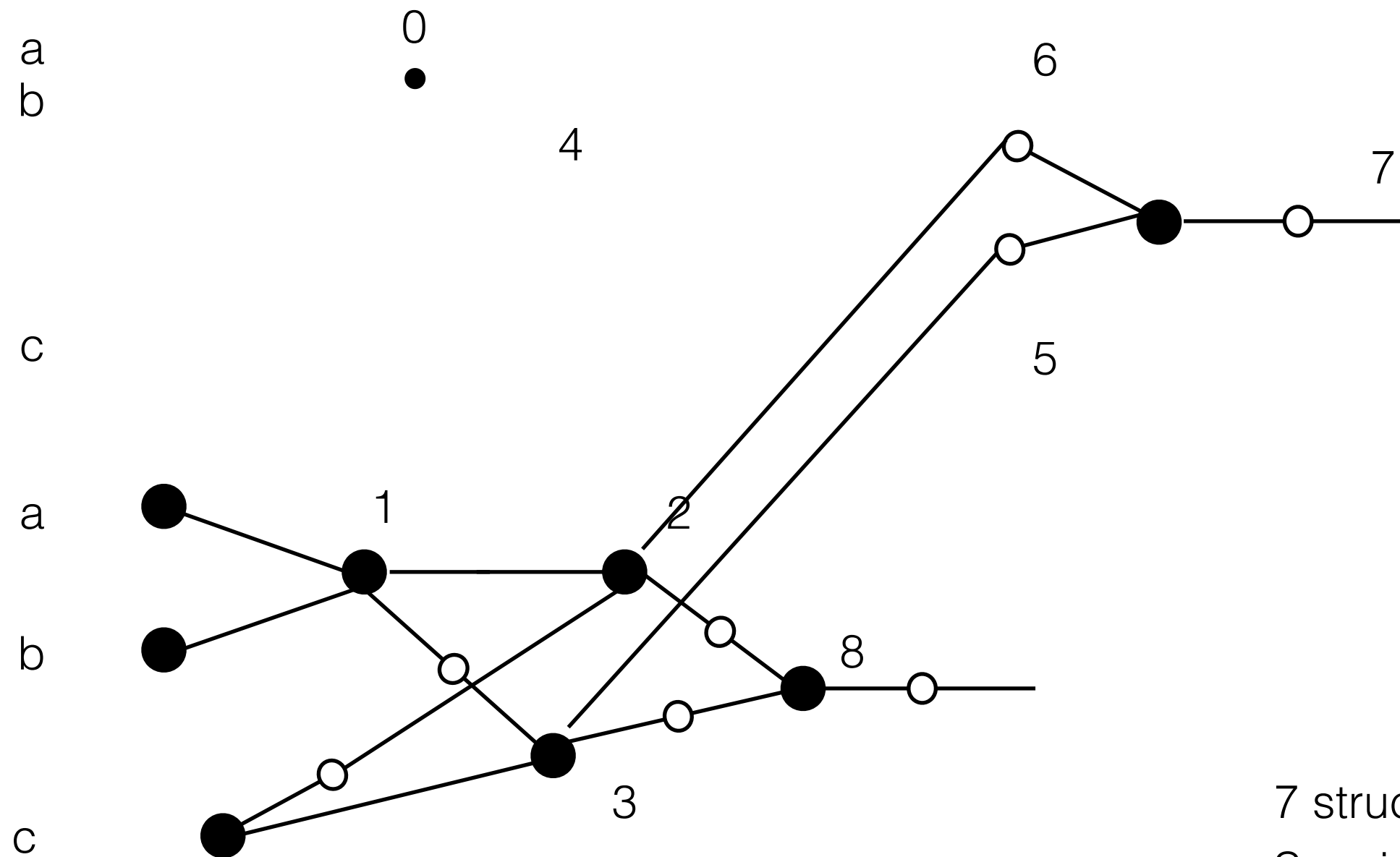
2,6

3,5

SAT solver: $2 = 6$

Merge nodes 2 and 6

SAT Sweeping (5)



7 structurally hashes to 8
So, circuits are equivalent

FRAIGS

- » Instead of SAT sweeping
- » On-the-fly build a Functionally Reduced AIG
 - » Structural hashing, one or two-levels
 - » Simulation with test-vectors
 - » Call SAT for possibly equivalent nodes
 - » Keep functional equivalent nodes, but re-use just one of them

Simple exercises to practice

- » See reader (Chapter 2) on the website