Hardware Verification 2IMF20

Julien Schmaltz

Lecture 03:

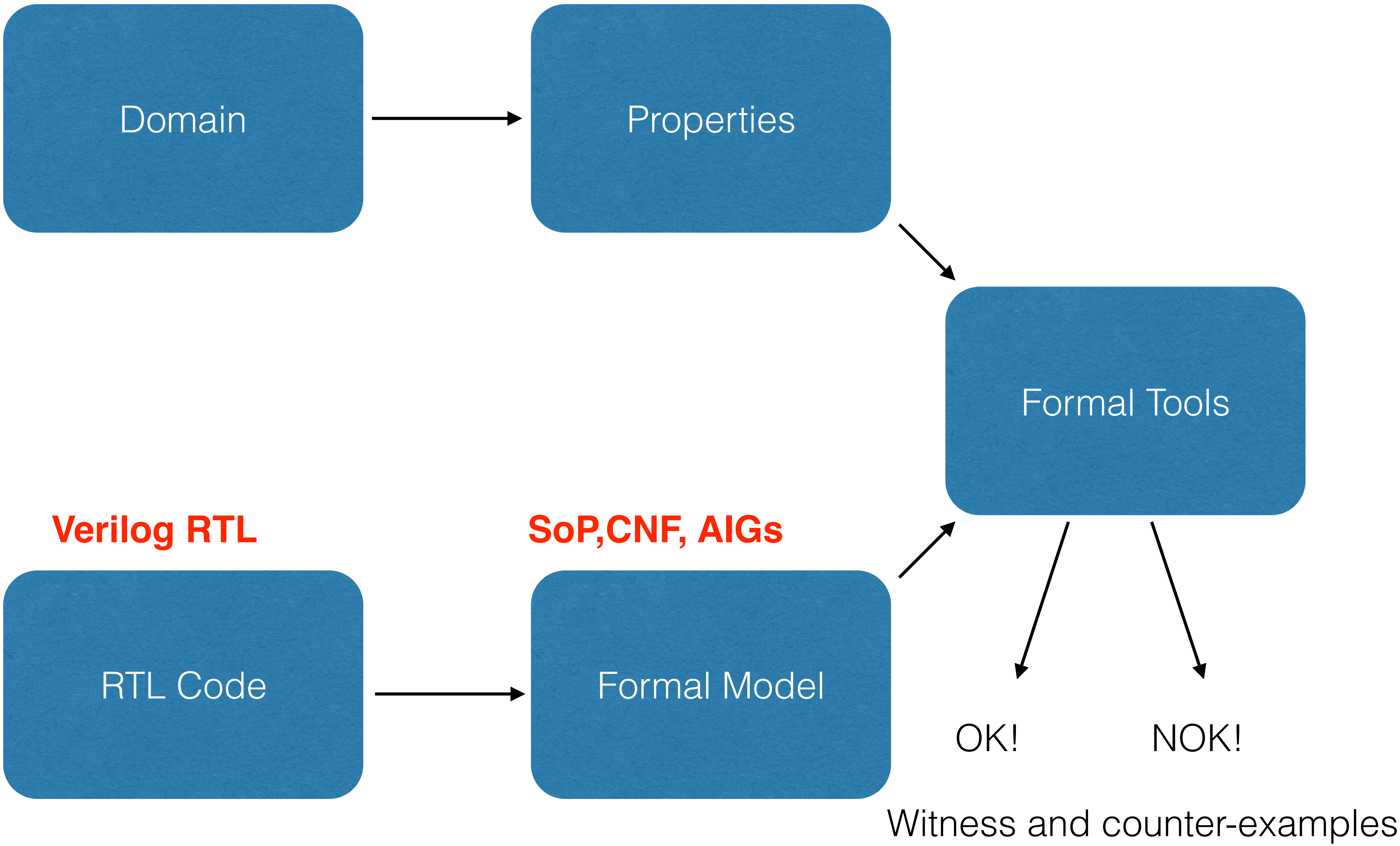
Reachability and SEC

(some slides courtesy of Brayton, Kuelman & Mishchenko)

Technische Universiteit Eindhoven University of Technology

Where innovation starts

Course content - Covered so far



Previously …

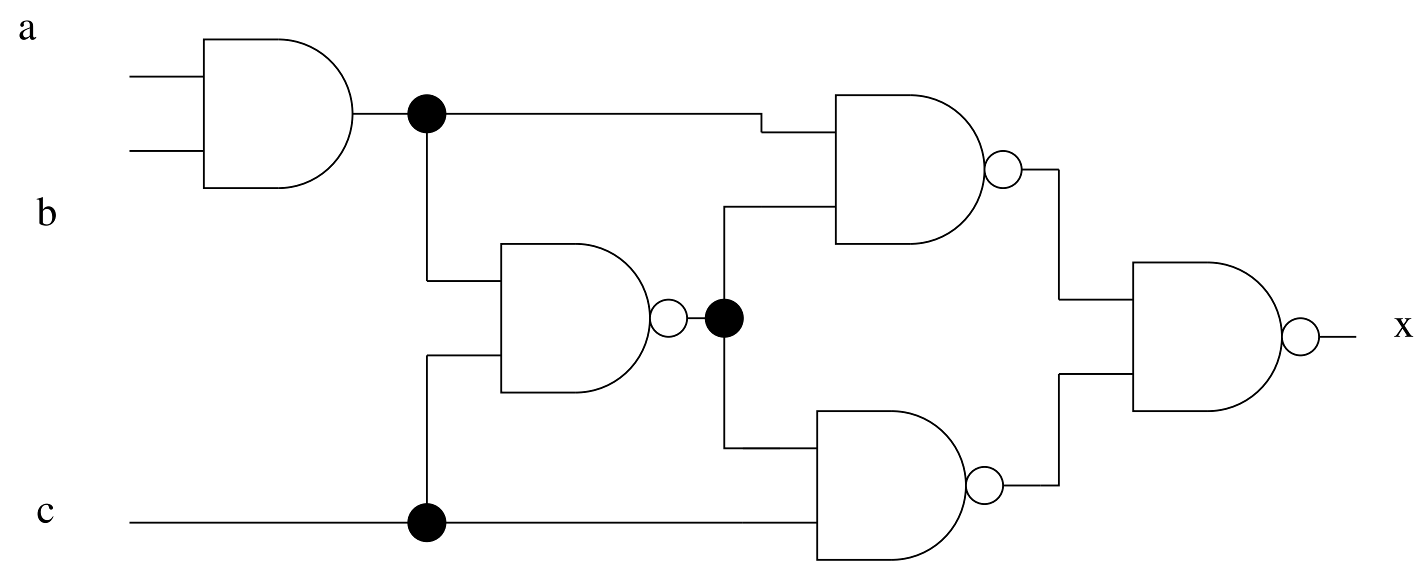
- Conversion from Hardware to Booleans

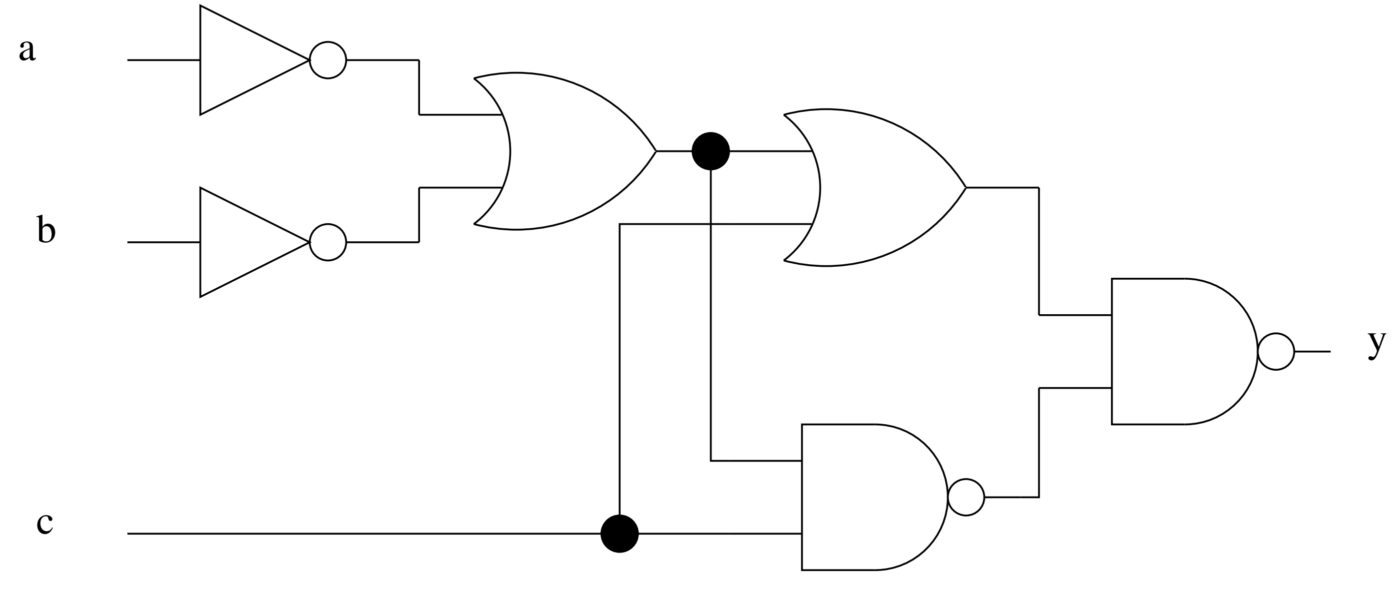
- Representation of Boolean functions
  - SoP
  - AIGs
  - BDDs

- Basic SAT solving (CNF)

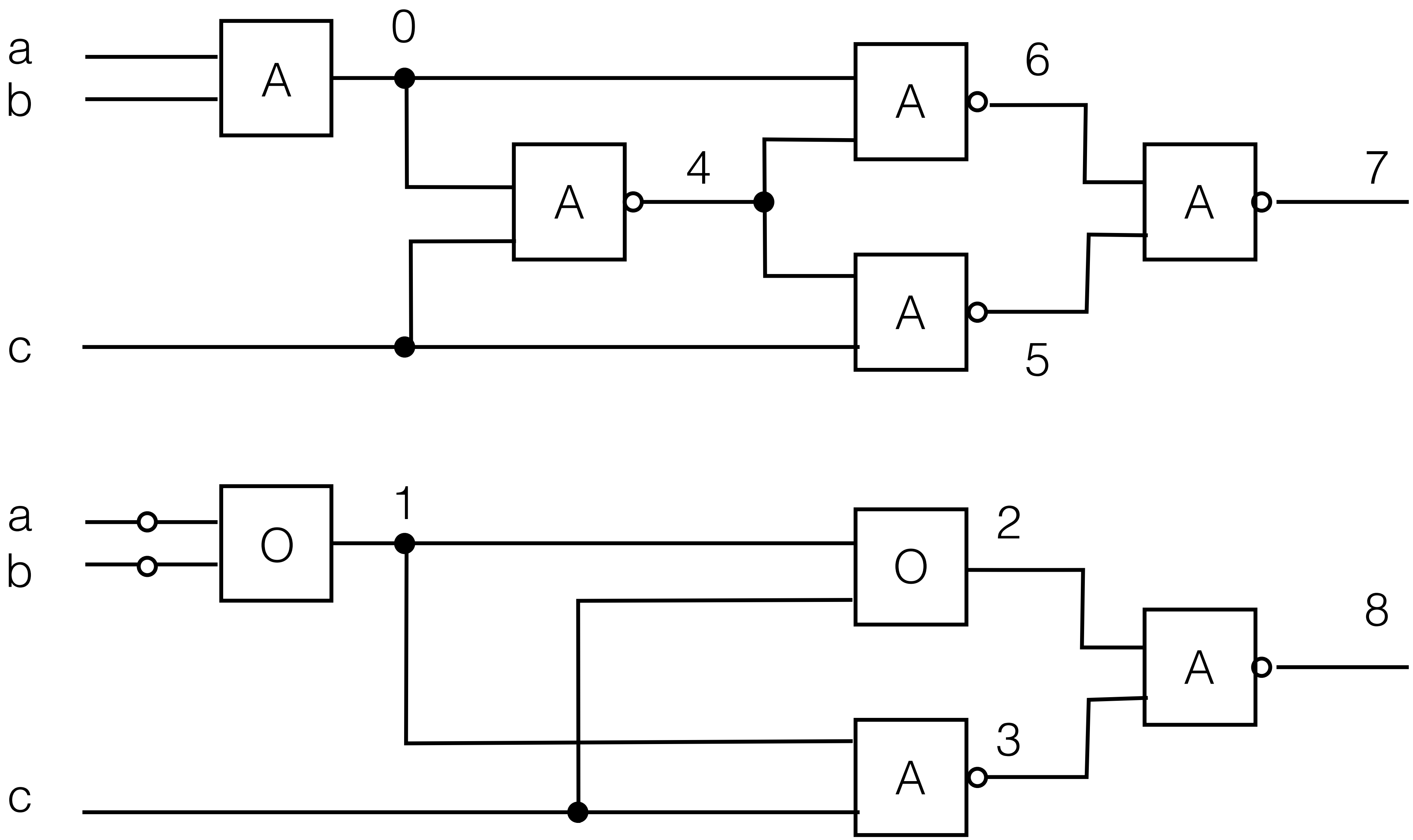
- Combinational Equivalence Checking
  - SoP, BDDs: normal form and equality
  - (FR)AIGs: prove equivalence using SAT/BDD sweeping

CEC - BDDs, AIGs, etc.





Recall these two circuits



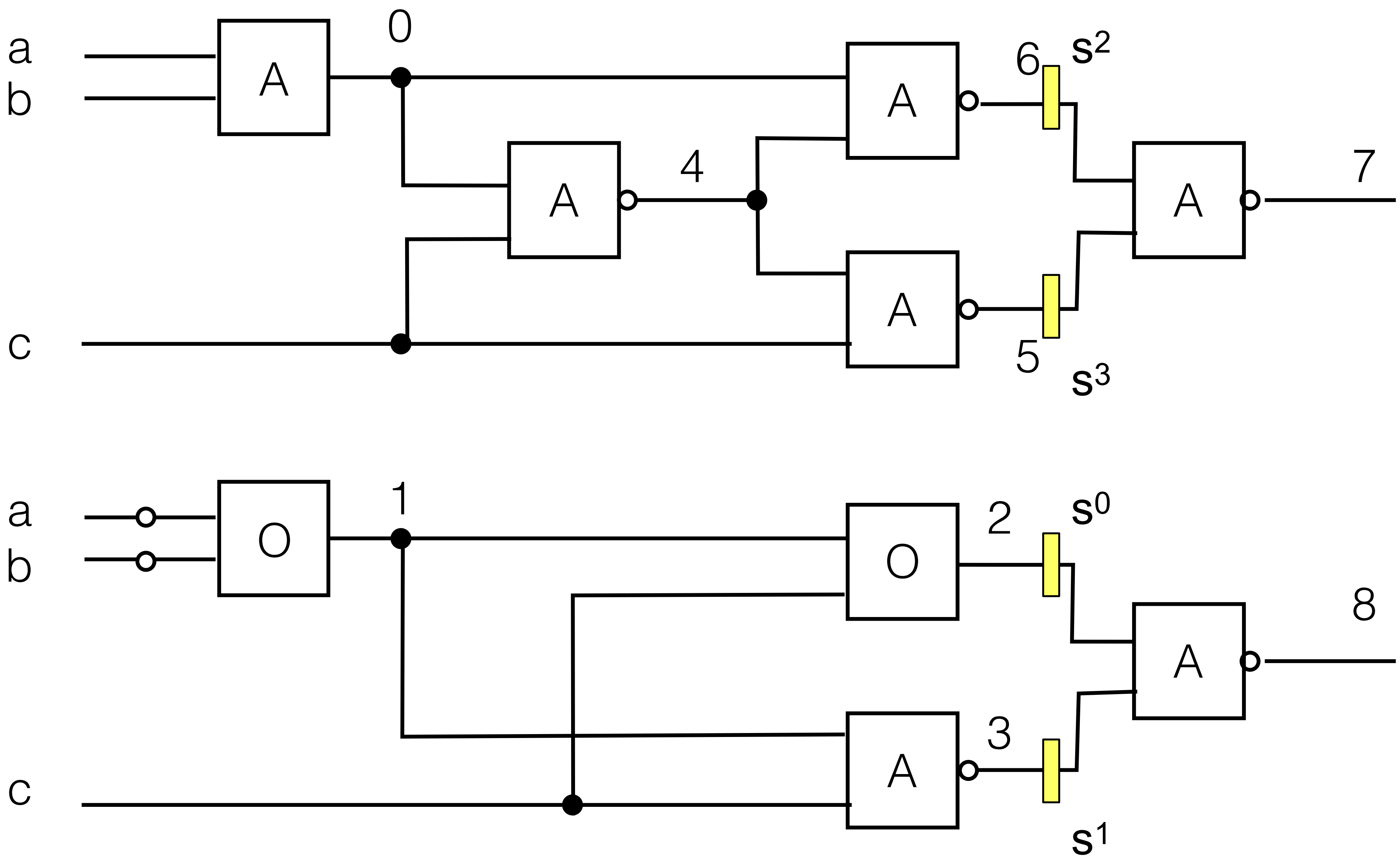
Equivalence classes

7,8

2,6

3,5

Registers at equivalent nodes



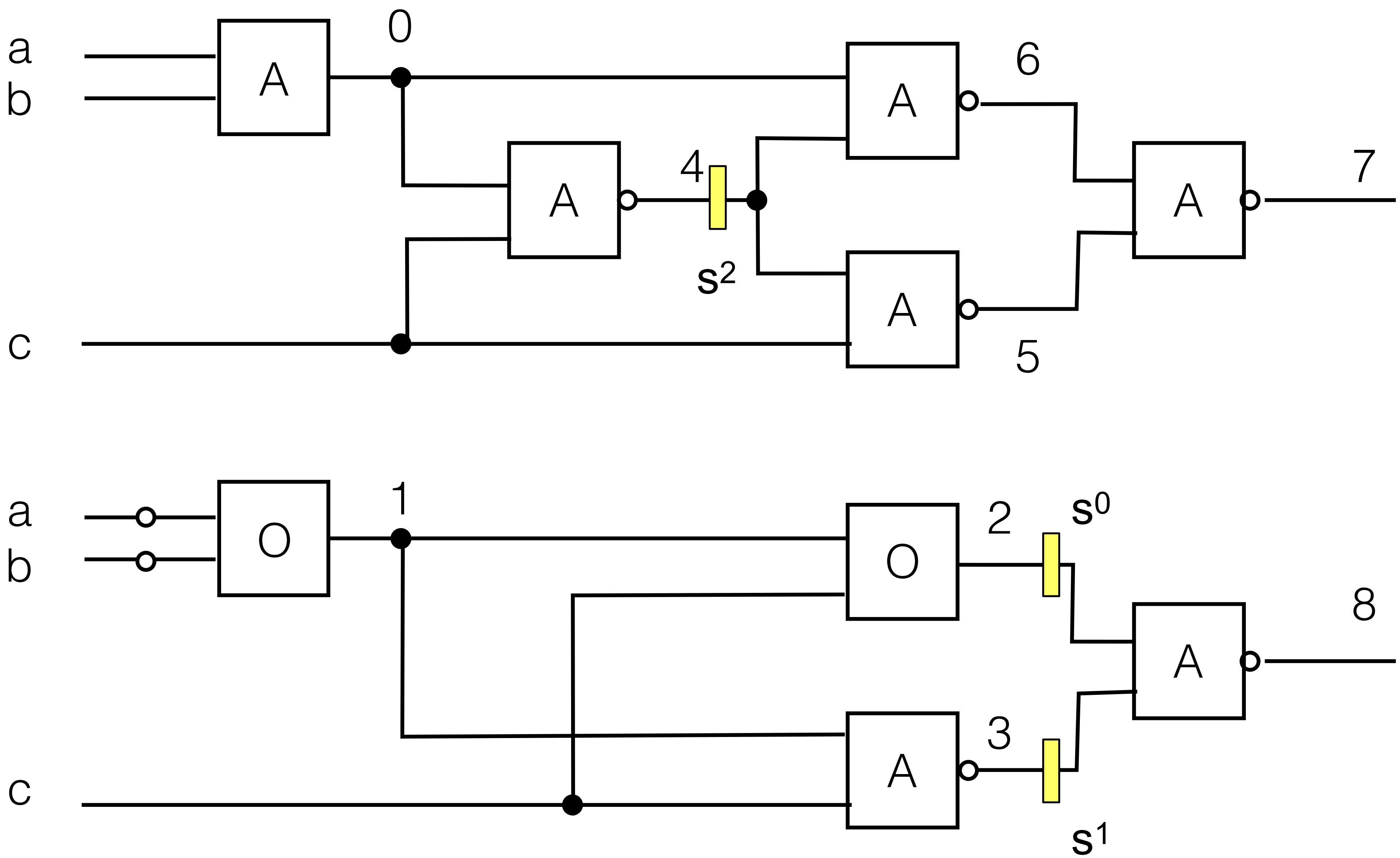
Equivalence classes

7,8

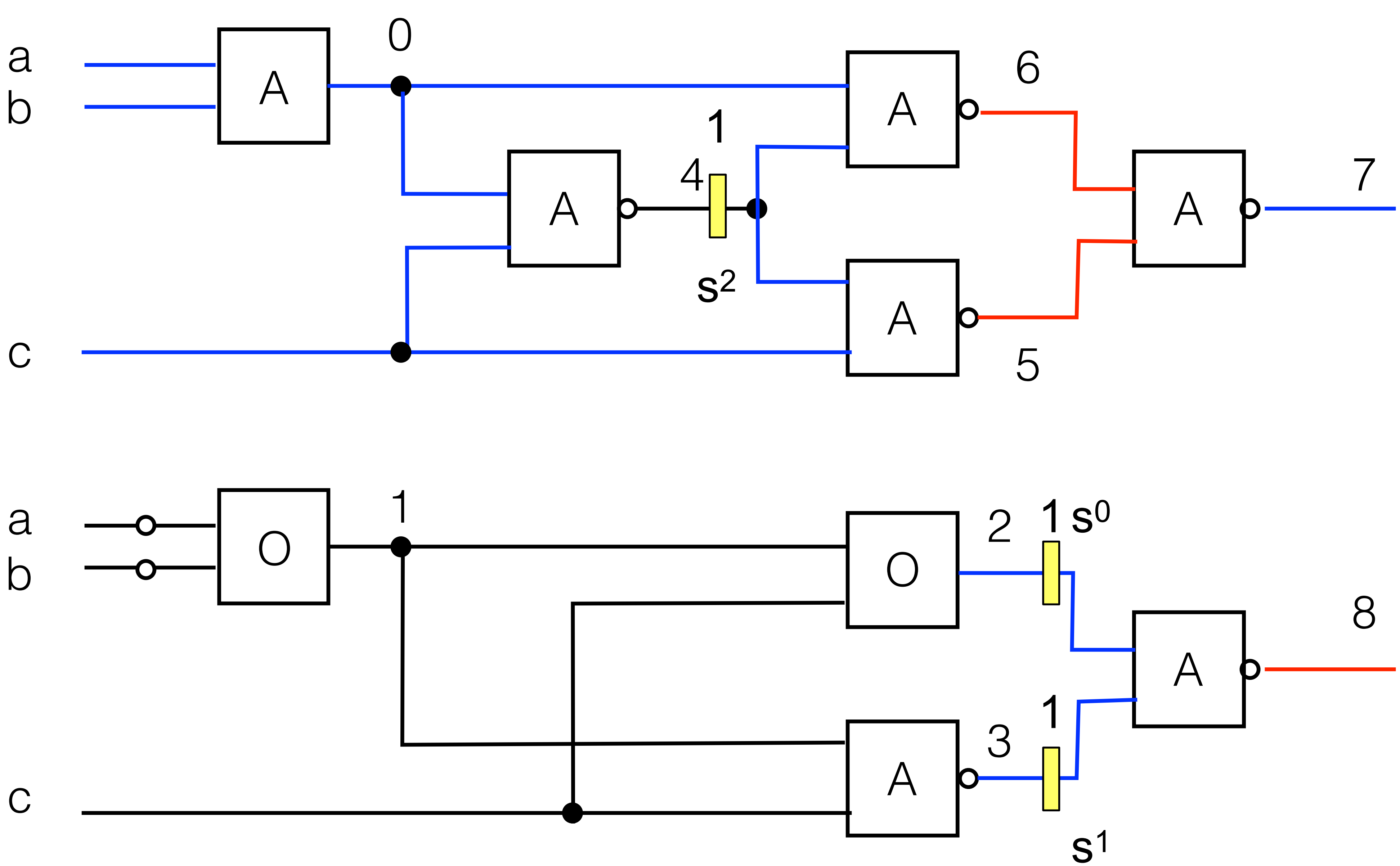
2,6

3,5

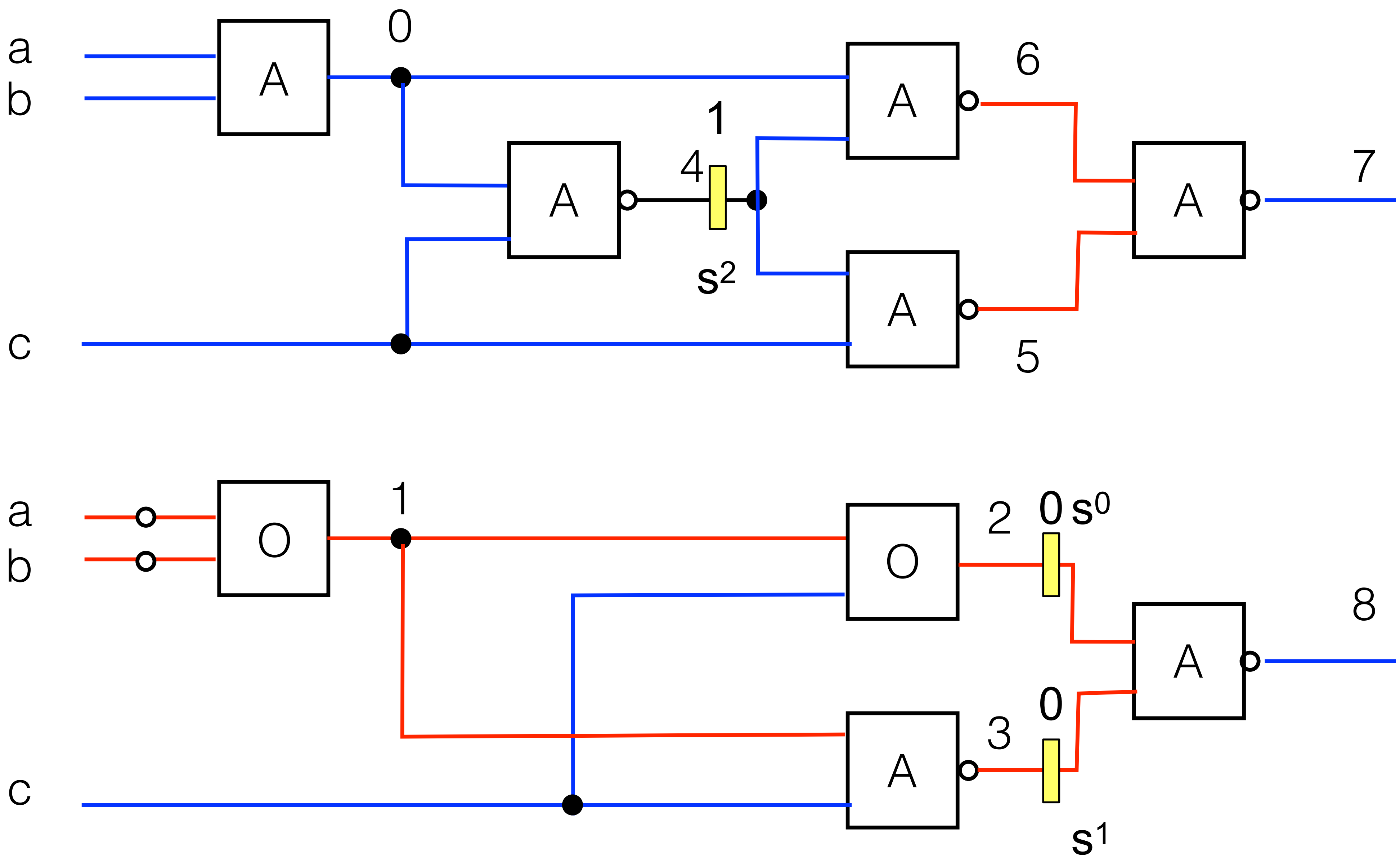
Registers at other nodes



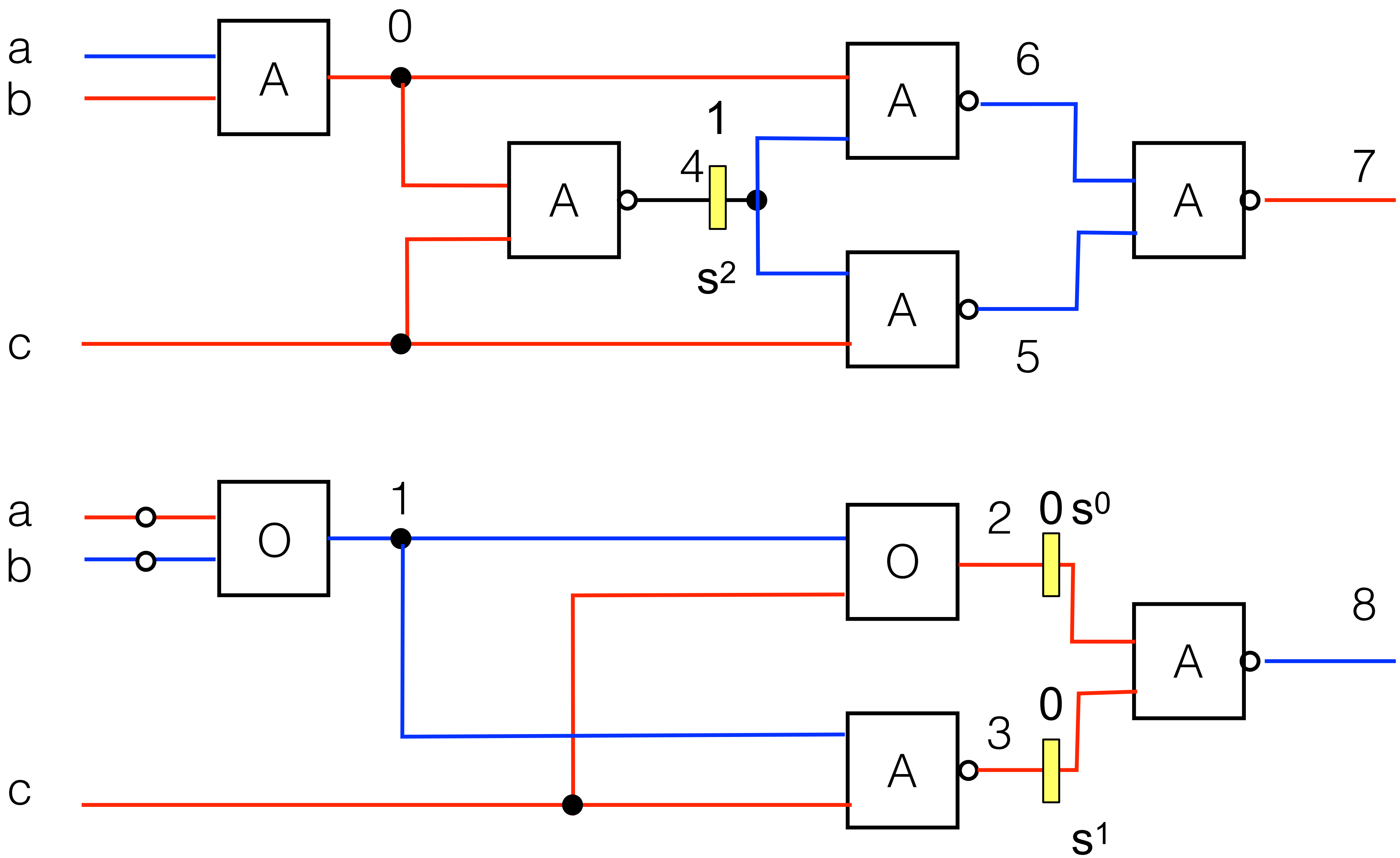
Initial values = 1, inputs = 1



Let's make the initial states equal



Let’s look at other inputs



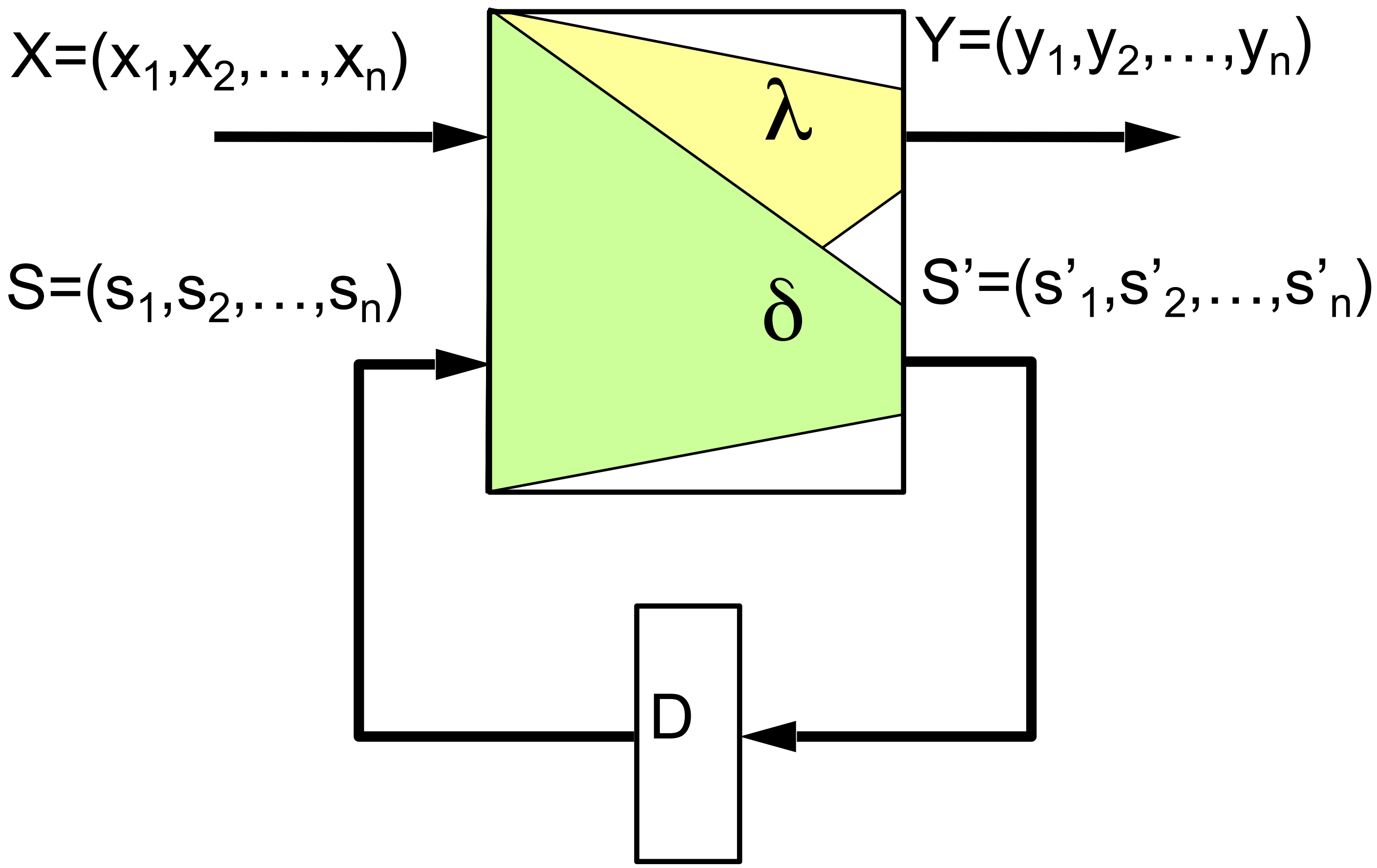
Conclusion

- SEC much harder than CEC
  - in practice, SEC less used than CEC
  - still, big steps forward
- General approach
  - try reducing to CEC (find name matching)
  - structural/functional register correspondence
  - **reachability analysis**

Program for today and next lecture

- Look at sequential circuits
- Equivalence defined as always producing the same output
  - **always = at all cycles**
- Reachability techniques for SEC
  - Forward, backward reachability
  - Symbolic reachability with BDD
  - Induction and k-induction with SAT

Finite State Machines (Mealy)



$M(X,Y,S,S_0,\delta,\lambda)$:

X: Inputs

Y: Outputs

S: Current State

S_0: Initial State(s)

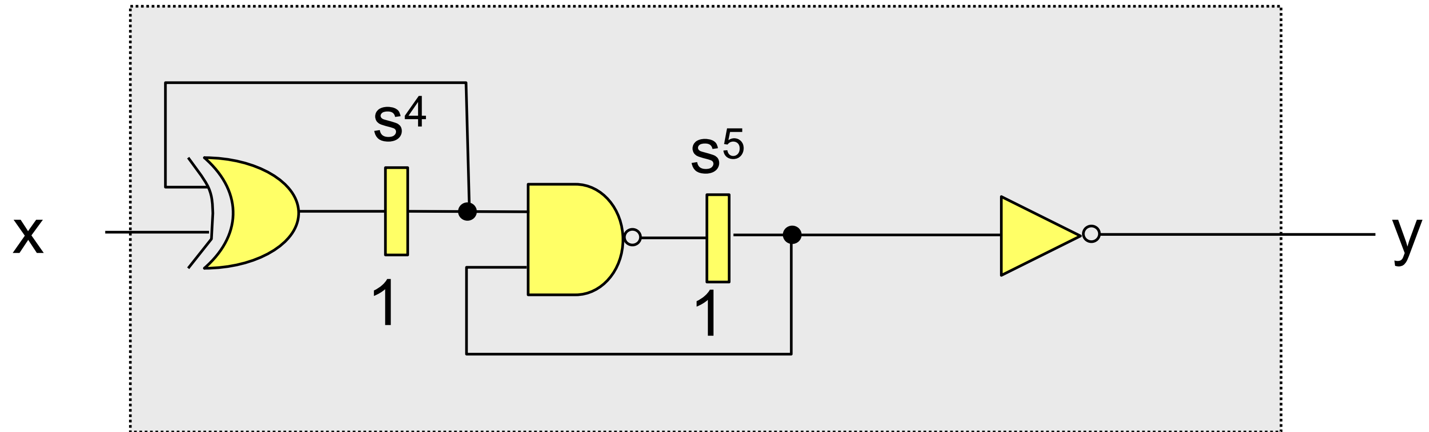
δ: $X \times S \rightarrow S$ (next state function)

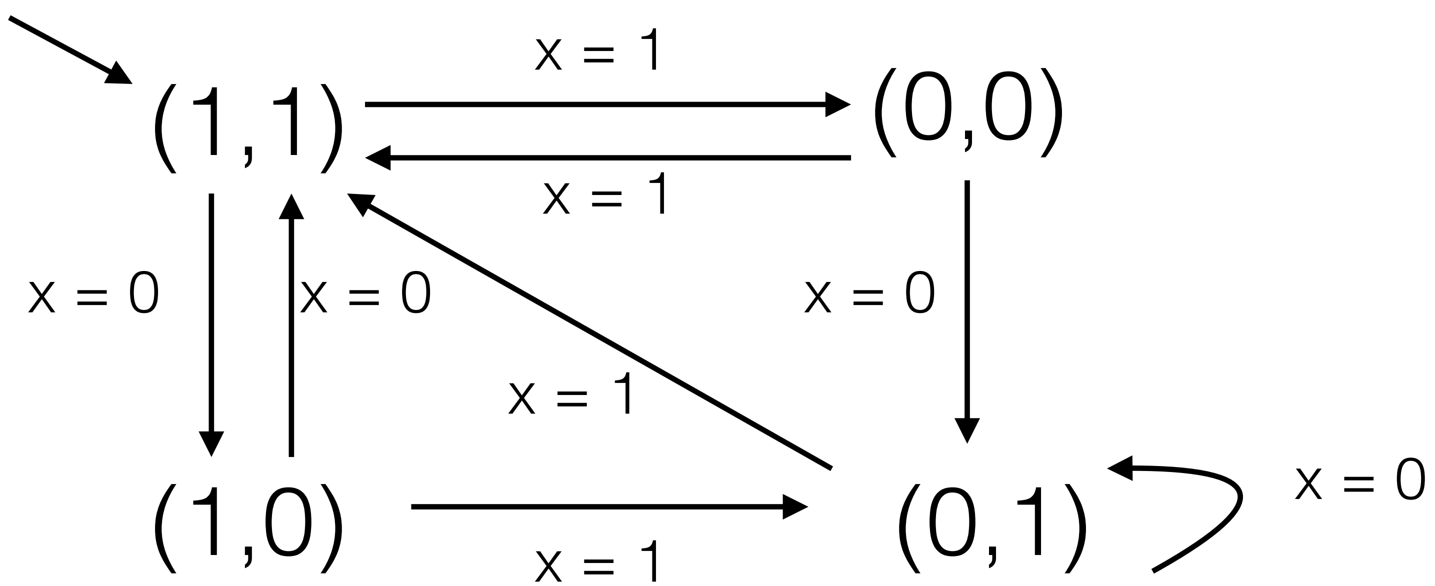
λ: $X \times S \rightarrow Y$ (output function)

Delay element(s):

- Clocked: synchronous
  - single-phase clock, multiple-phase clocks
- Unclocked: asynchronous

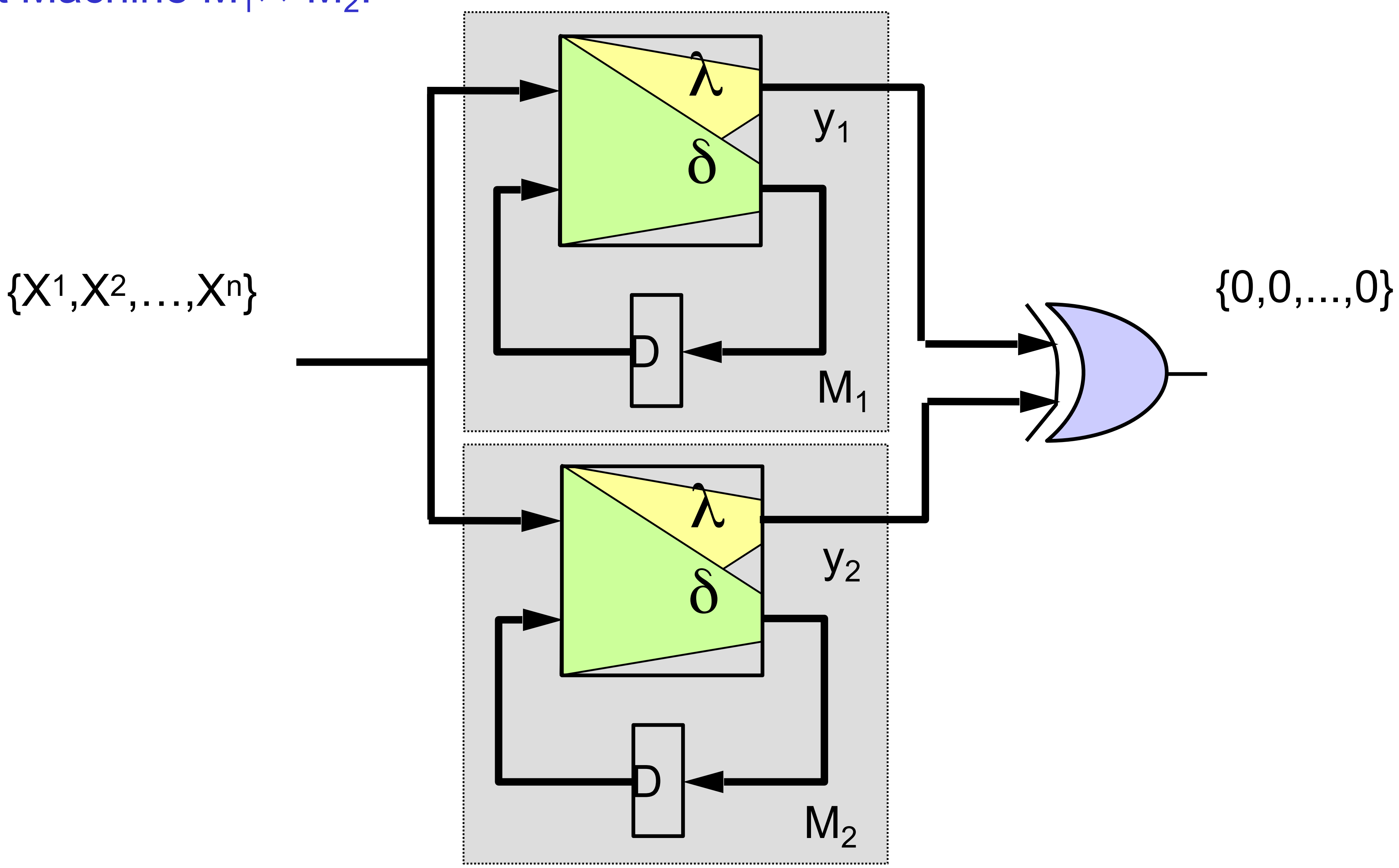
Sequential circuit and its state graph





Finite State Machines Equivalence

Build Product Machine $M_1 \times M_2$:



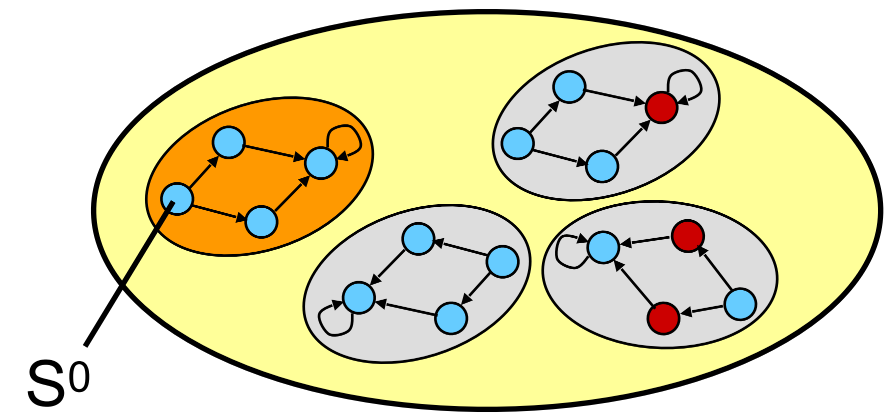
Definition:

M_1 and M_2 are functionally equivalent iff the product machine $M_1 \times M_2$ produces a constant 0 for all valid input sequences $\{X_1,\ldots,X_n\}$.

State Traversal Techniques

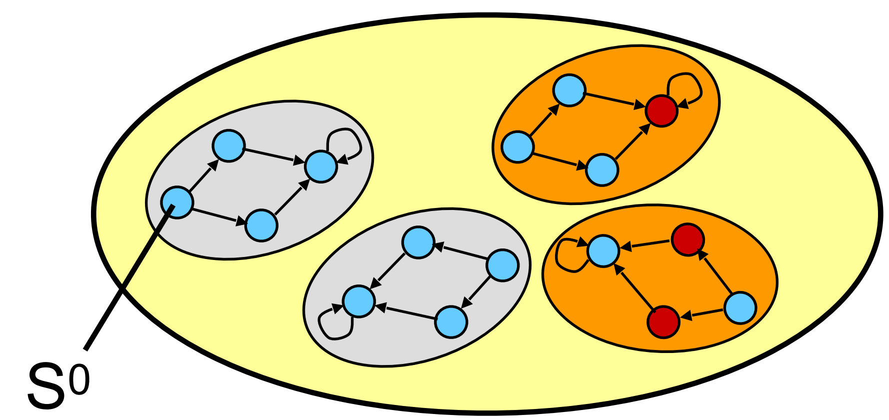
Forward Traversal:

- start from initial state(s)
- traverse forward to check whether “bad” state(s) is reachable



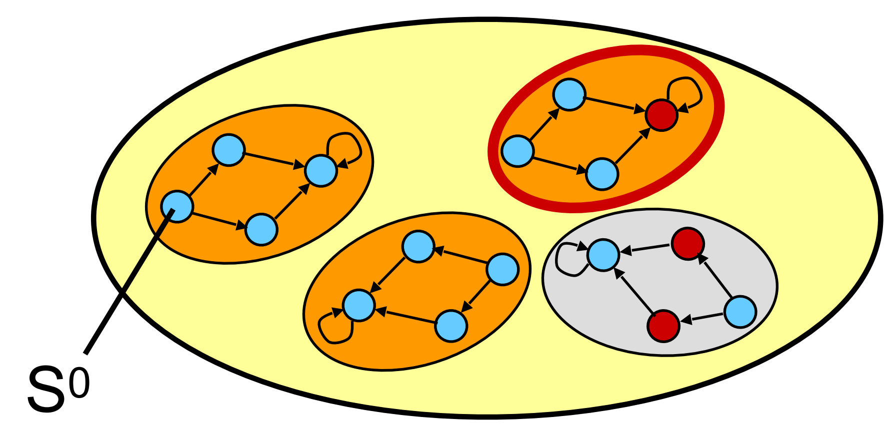
Backward Traversal:

- start from bad state(s)
- traverse backward to check whether initial state(s) can reach them

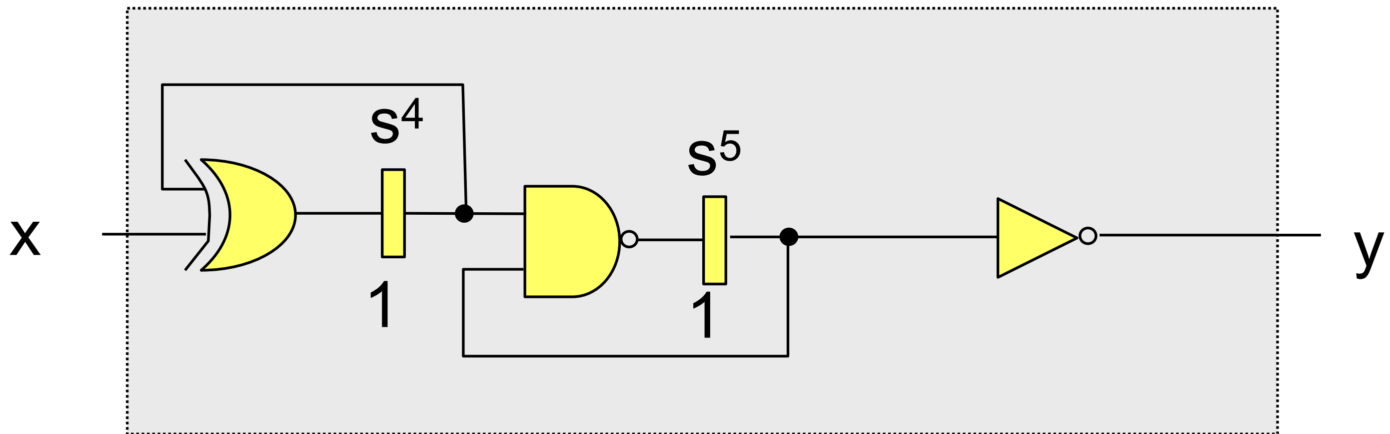


Combines Forward/Backward traversal:

- compute over-approximation of reachable states by forward traversal
- for all bad states in over-approximation, start backward traversal to see whether initial state can reach them



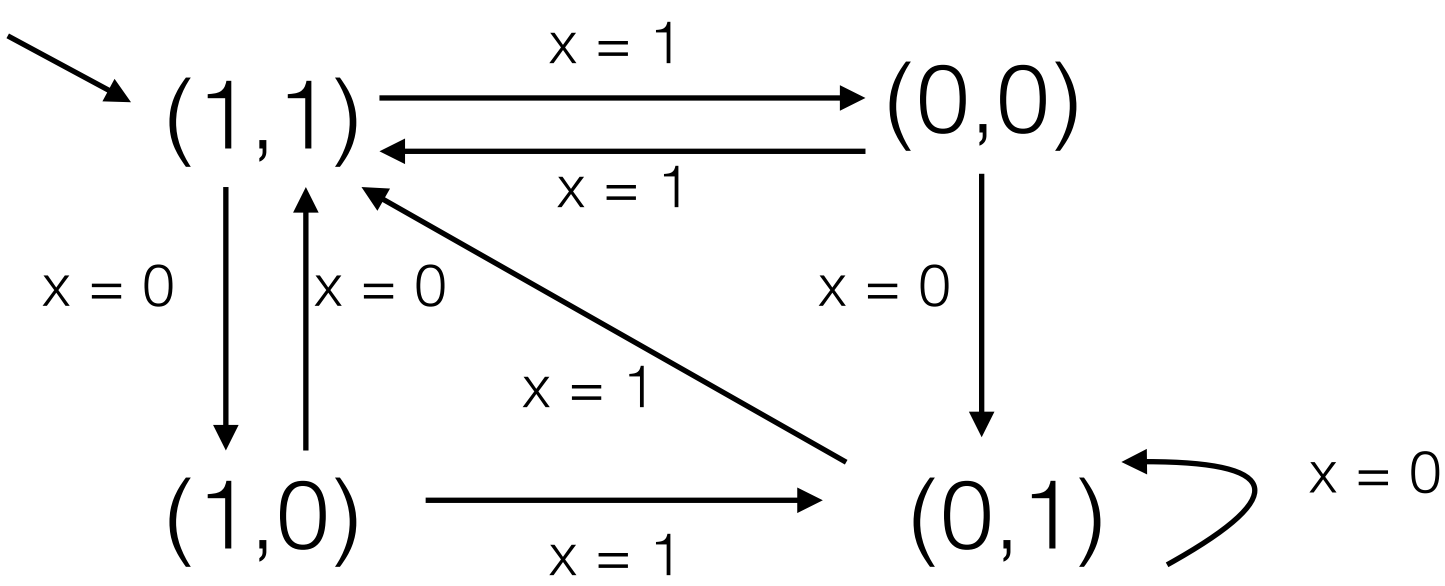
Transition relation *T*



sequential circuit

Definition.

$$(s', s) \in T \equiv \exists x. s' = \delta(x, s)$$

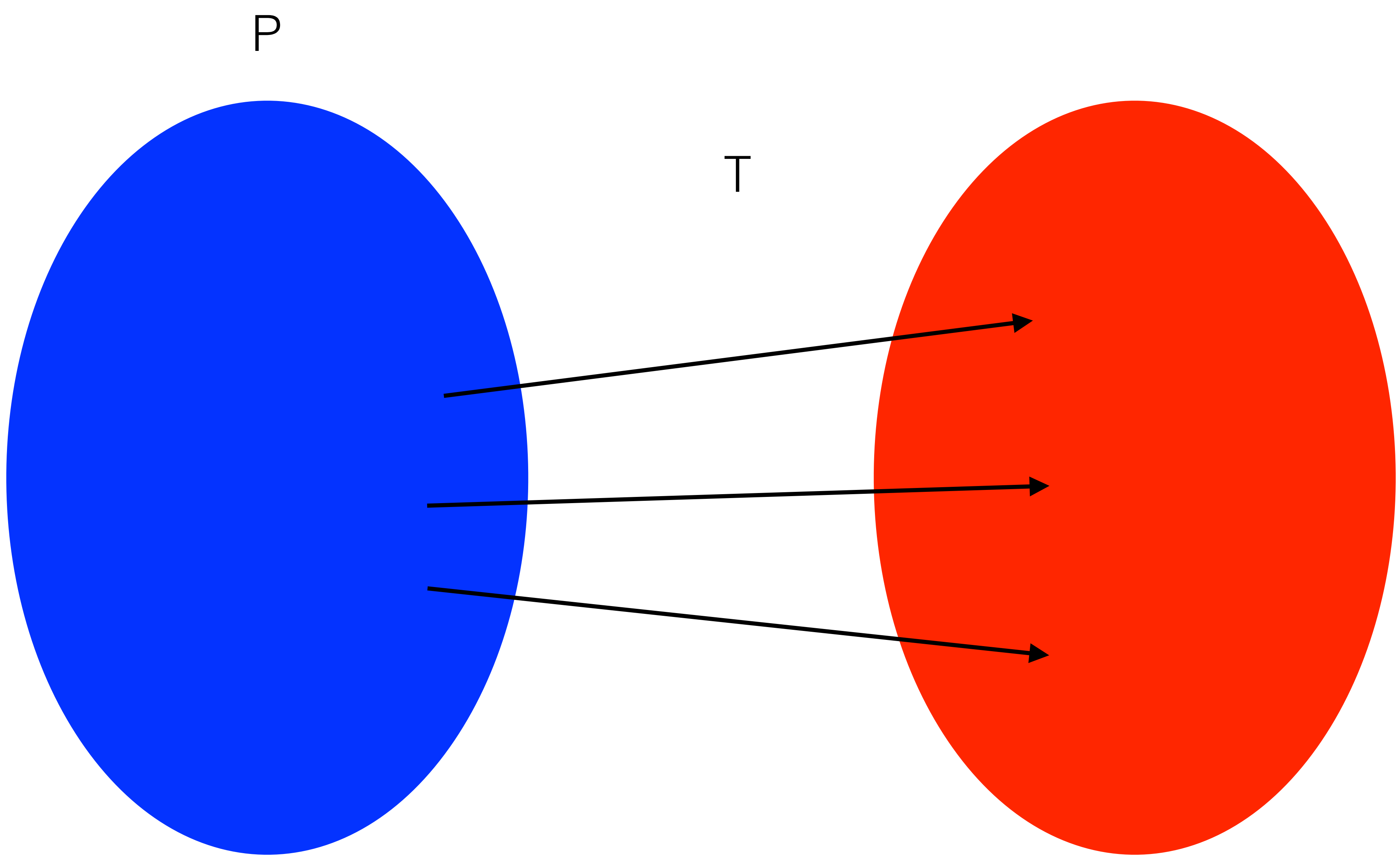


state graph representation

Example.

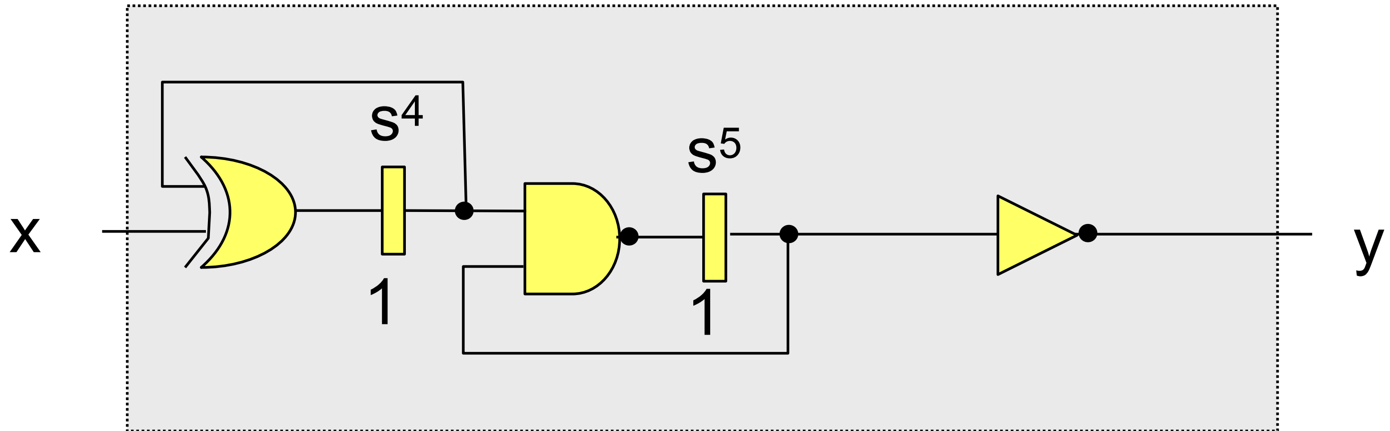
T = { (11,10),(11,00),(00,01),(00,11),
(10,11),(10,01),(01,01),(01,11)}

Forward image of a set of states



$$Fwd(P, T) = \{s' | \exists s. s \in P \land (s', s) \in T\}$$

Fwd image - Example.

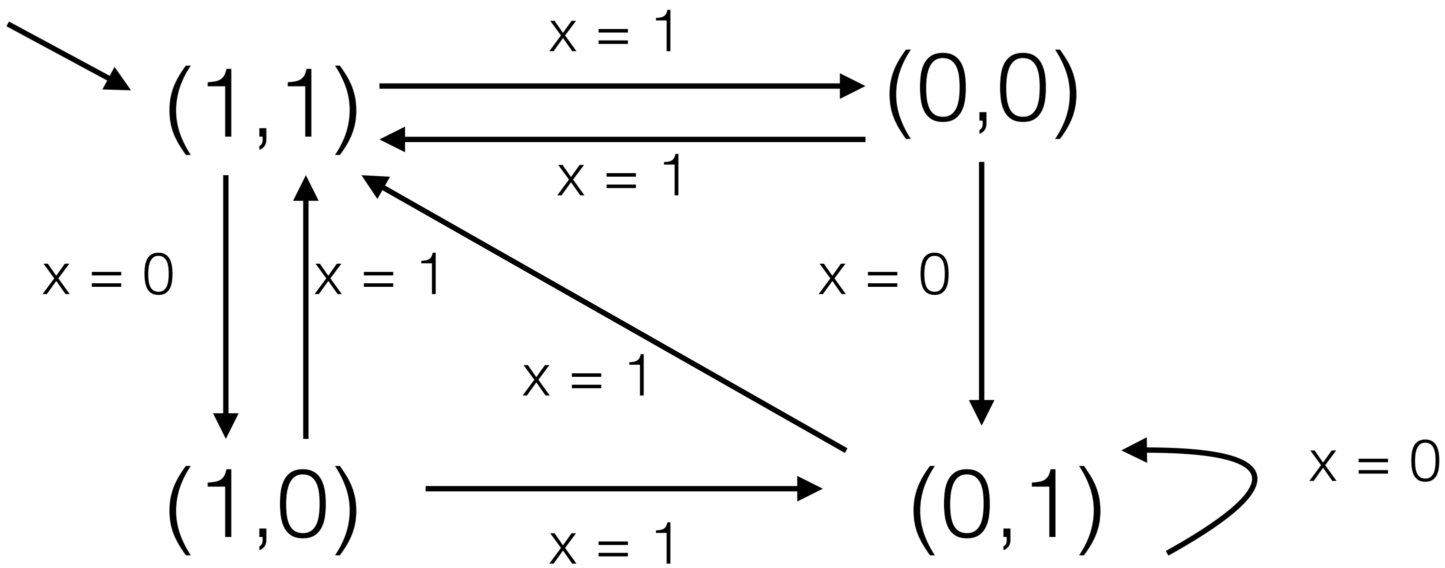


sequential circuit

Compute.

Fwd({(11)}, T) = ?

Fwd({(10),(00)}, T) = ?

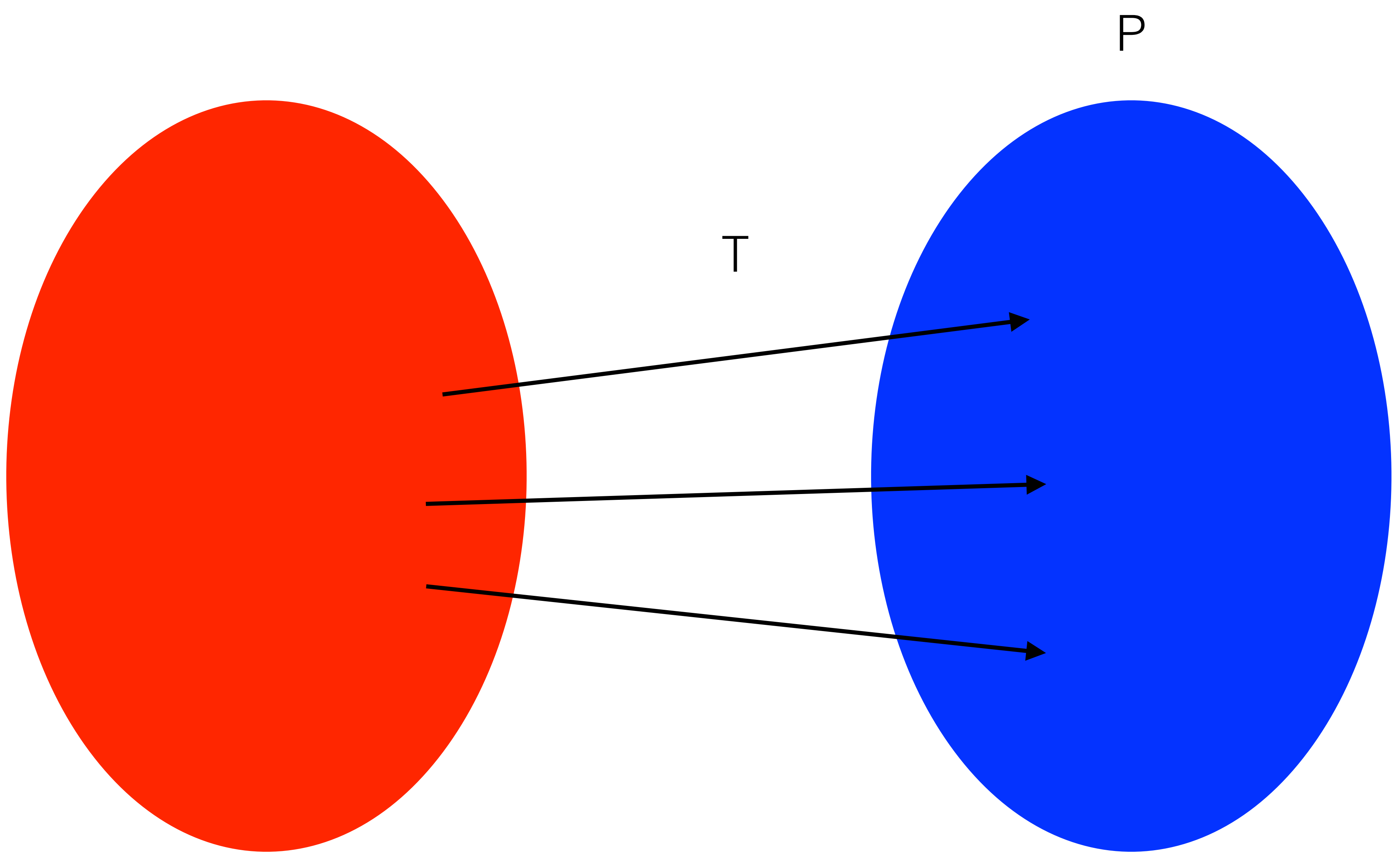


Example.

T = { (11,10),(11,00),(00,01),(00,11),
(10,11),(10,01),(01,01),(01,11)}

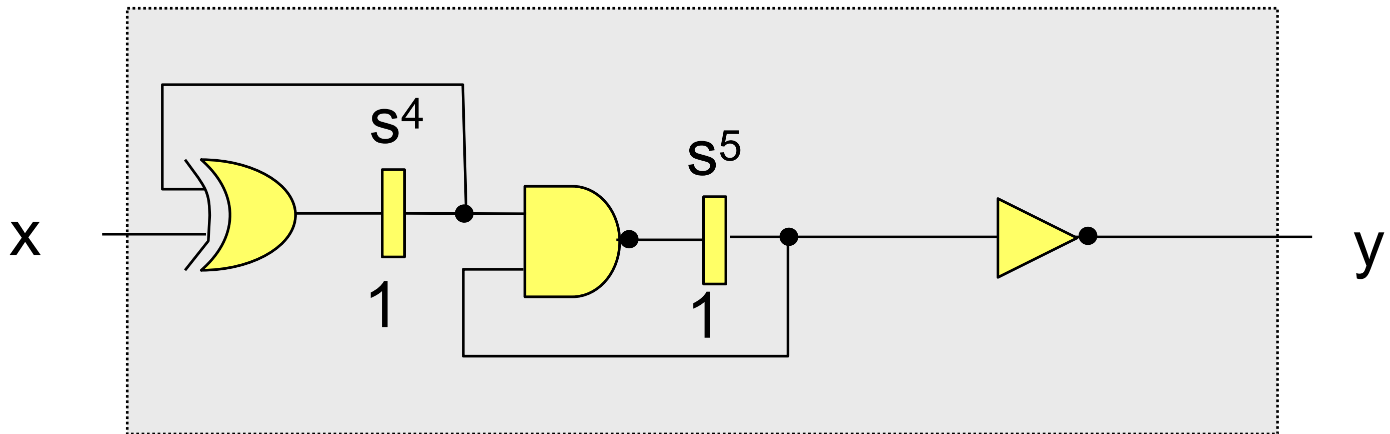
$$Fwd(P,T) = \{s' | \exists s . s \in P \wedge (s', s) \in T\}$$

Backward image of a set of states



$$Bwd(P,T) = \{s | \exists s'.s' \in P \wedge (s',s) \in T\}$$

Bwd image - Example.

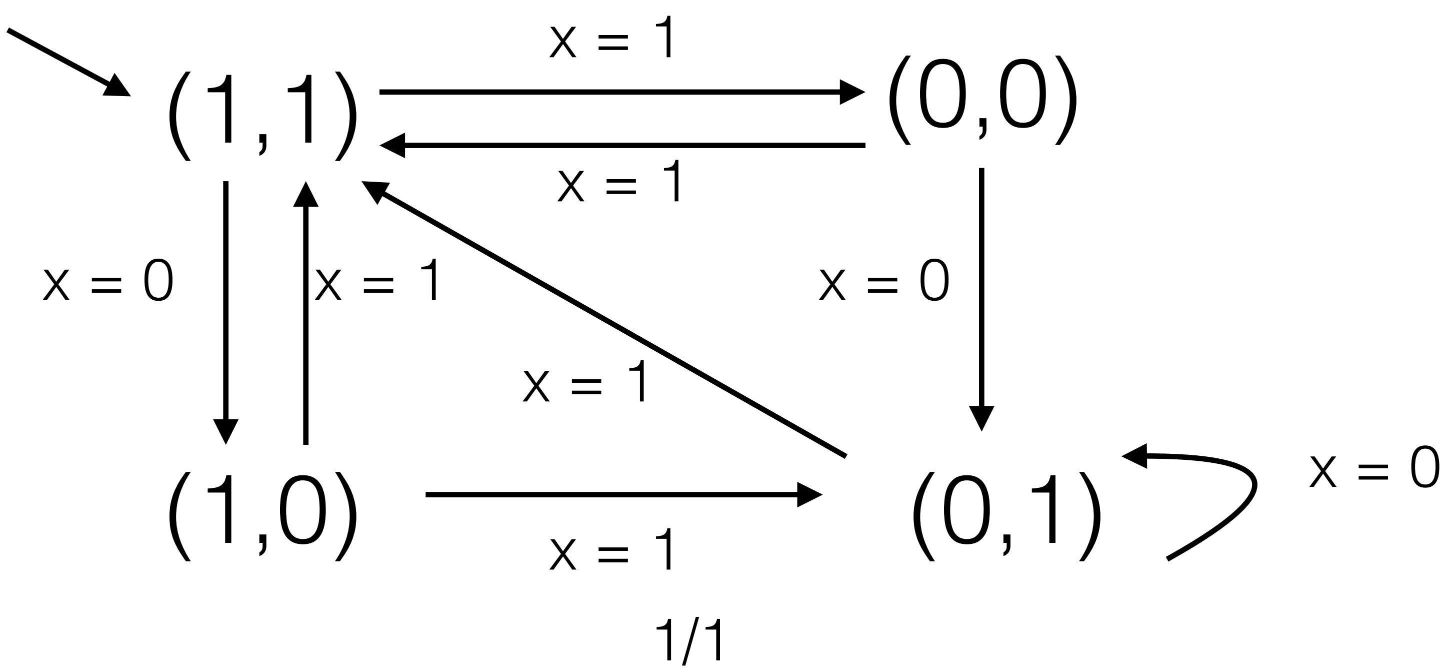


sequential circuit

Compute.

Bwd({(01)}, T) = ?

Bwd({(00),(10)}, T) = ?



Example.

T = { (11,10),(11,00),(00,01),(00,11),
(10,11),(10,01),(01,01),(01,11)}

$$Bwd(P,T) = \{s \mid \exists s'.s' \in P \wedge (s',s) \in T\}$$

Forward state traversal

```
Algorithm TRAVERSE_FORWARD(t, λ ,S⁰) {
  reached = ∅
  current = S⁰                                // start from init
  while (reached ≠ (reached ∨ current)) {    // fixed point
    reached = reached ∨ current               // add new states
    next    = IMG(t,current)                  // one trans.
    current = next                            // rename variab.
  }
  return ∃x.(λ(x,s) ∧ reached)               // not equivalent?
}
```

Termination guaranteed because **finitely** many states.

Backward state traversal

```
Algorithm TRAVERSE_BACKWARD(t, λ ,S⁰) {
  reached = ∅
  current = ∃x.λ(x,s)                      // start from bad
  while (reached ≠ (reached ∨ current)) { // fixed point
    reached  = reached ∨ current           // add new states
    previous = PRE_IMG(t,current)          // one trans.
    current  = previous                    // rename variab.
  }
  return (S⁰ ∧ reached)                   // reached init?
}
```

Termination guaranteed because **finitely** many states.

Symbolic reachability with BDDs

- Explicit state traversal
  - suffers from the state-explosion problem
  - sometimes faster than BDDs
- Efficient alternative
  - represent state transitions using BDDs
  - suffers from high memory usage (GBs in practice)

Symbolic Reachability with BDD’s

(RO)BDD's (Reduced Ordered) Binary Decision Diagrams

[Bryant 1986]

- Canonical form representation for Boolean functions
- Substantially more compact than CNF or DNF
- Efficient manipulation of BDD's

Shannon and Binary Decision Trees

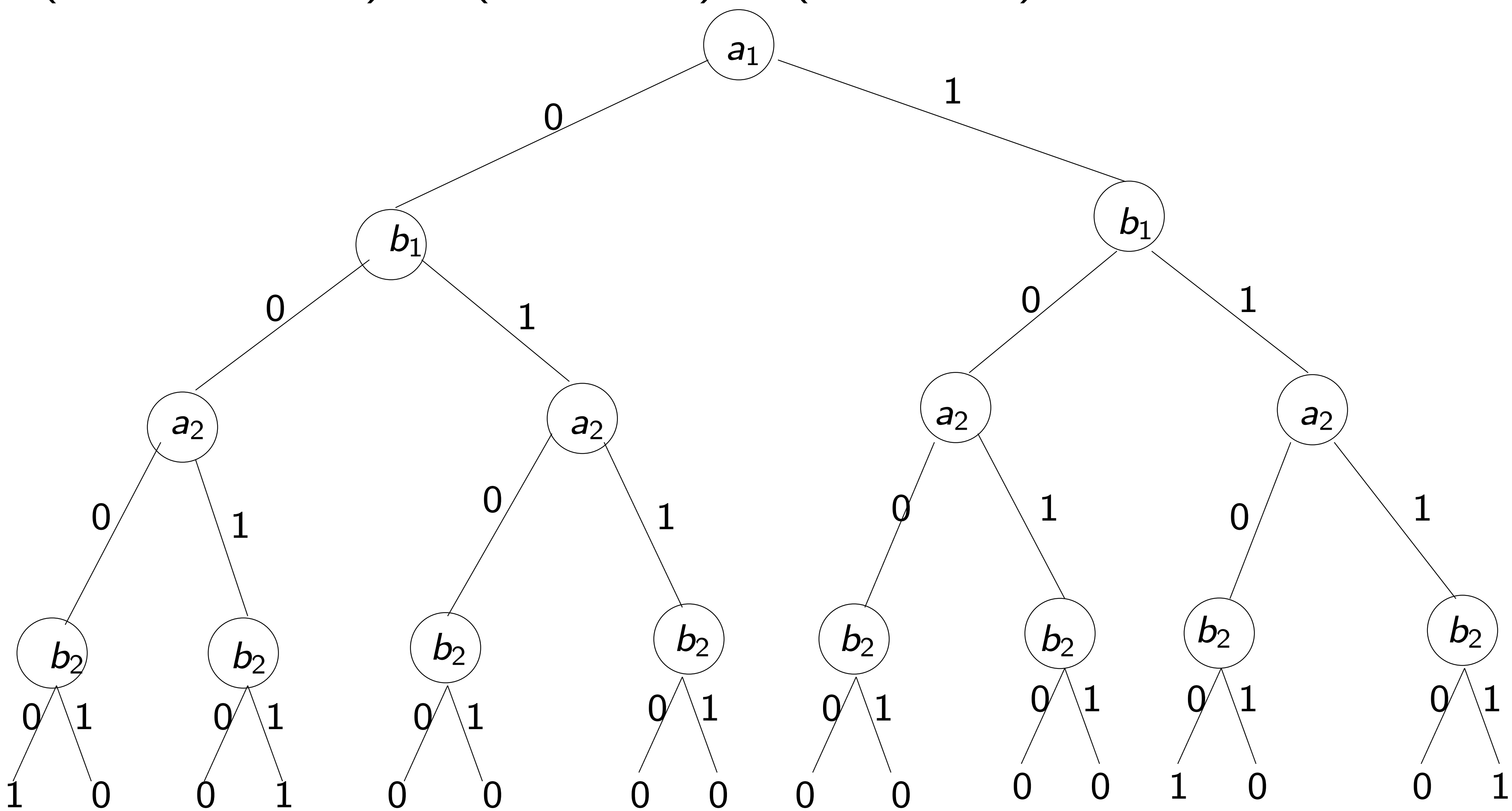
- Shannon expansion for Boolean function f

$$f = (\neg a \wedge f|_{a=0}) \vee (a \wedge f|_{a=1})$$

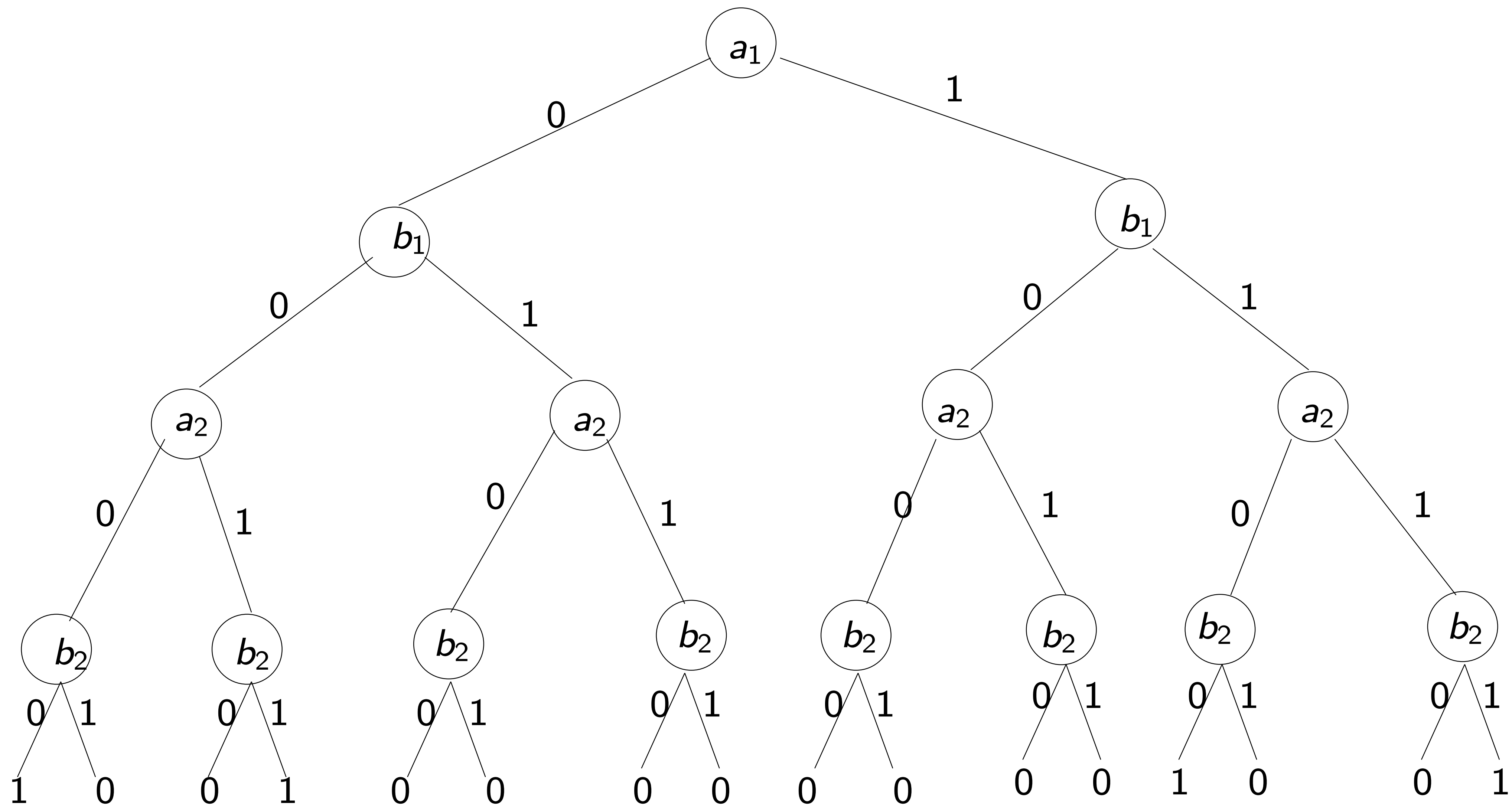
- Using this expansion and a variable ordering, one can build a binary decision tree
- Binary Decision Trees are not very compact (same size as truth tables)

Binary Decision Tree for a 2-bit comparator

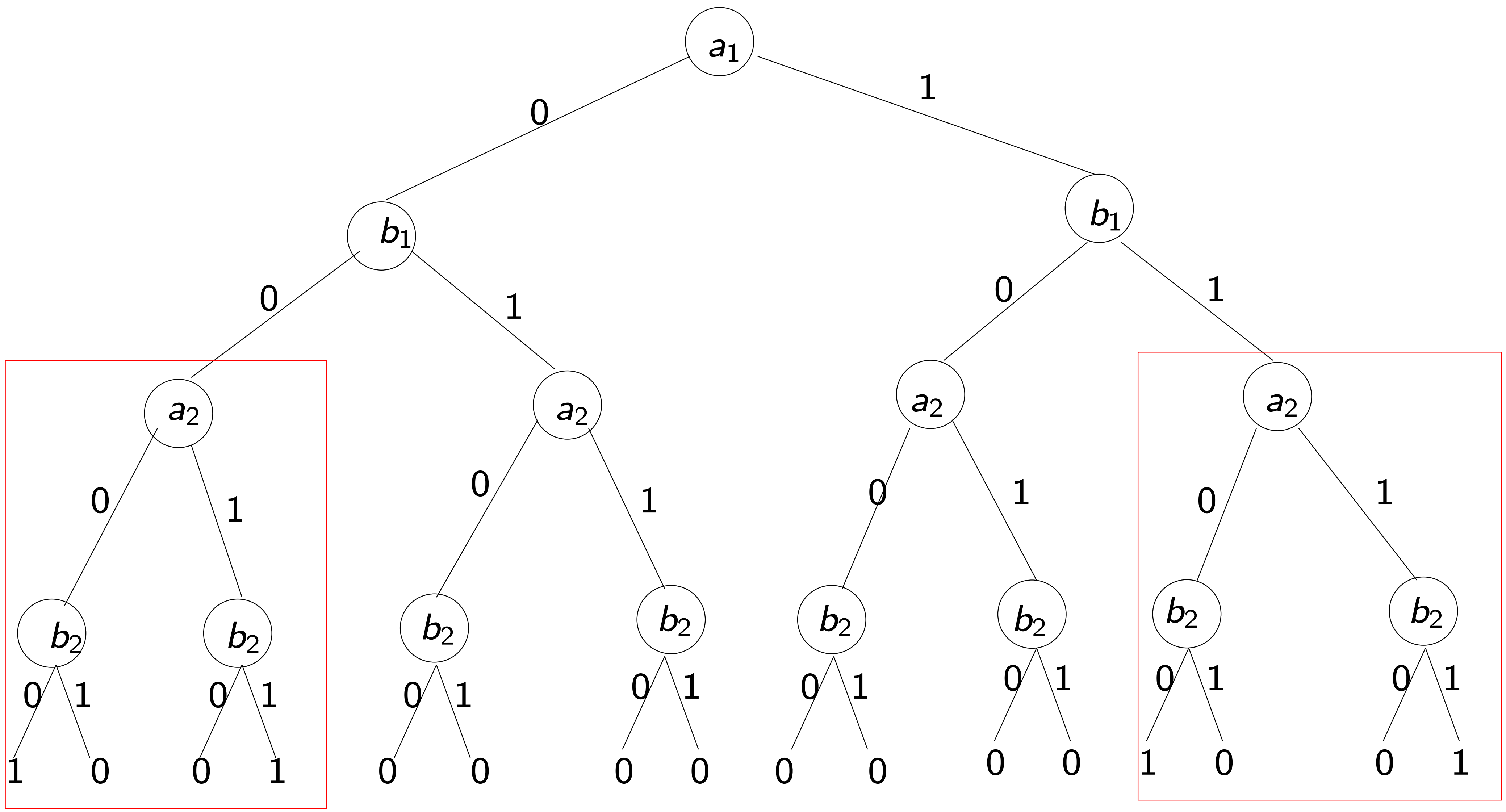
$$f(a_1, a_2, b_1, b_2) = (a_1 \Leftrightarrow b_1) \wedge (a_2 \Leftrightarrow b_2)$$



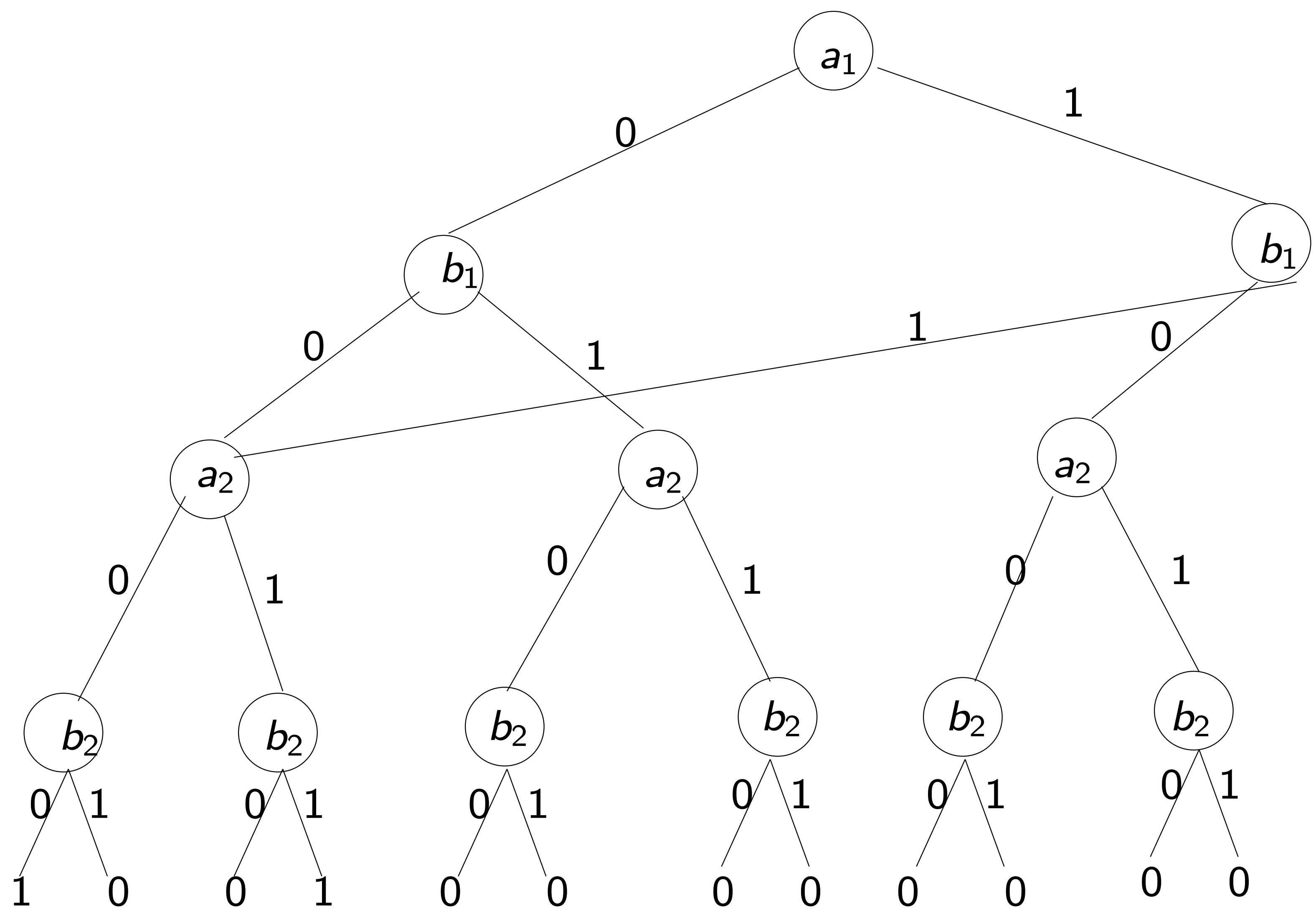
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



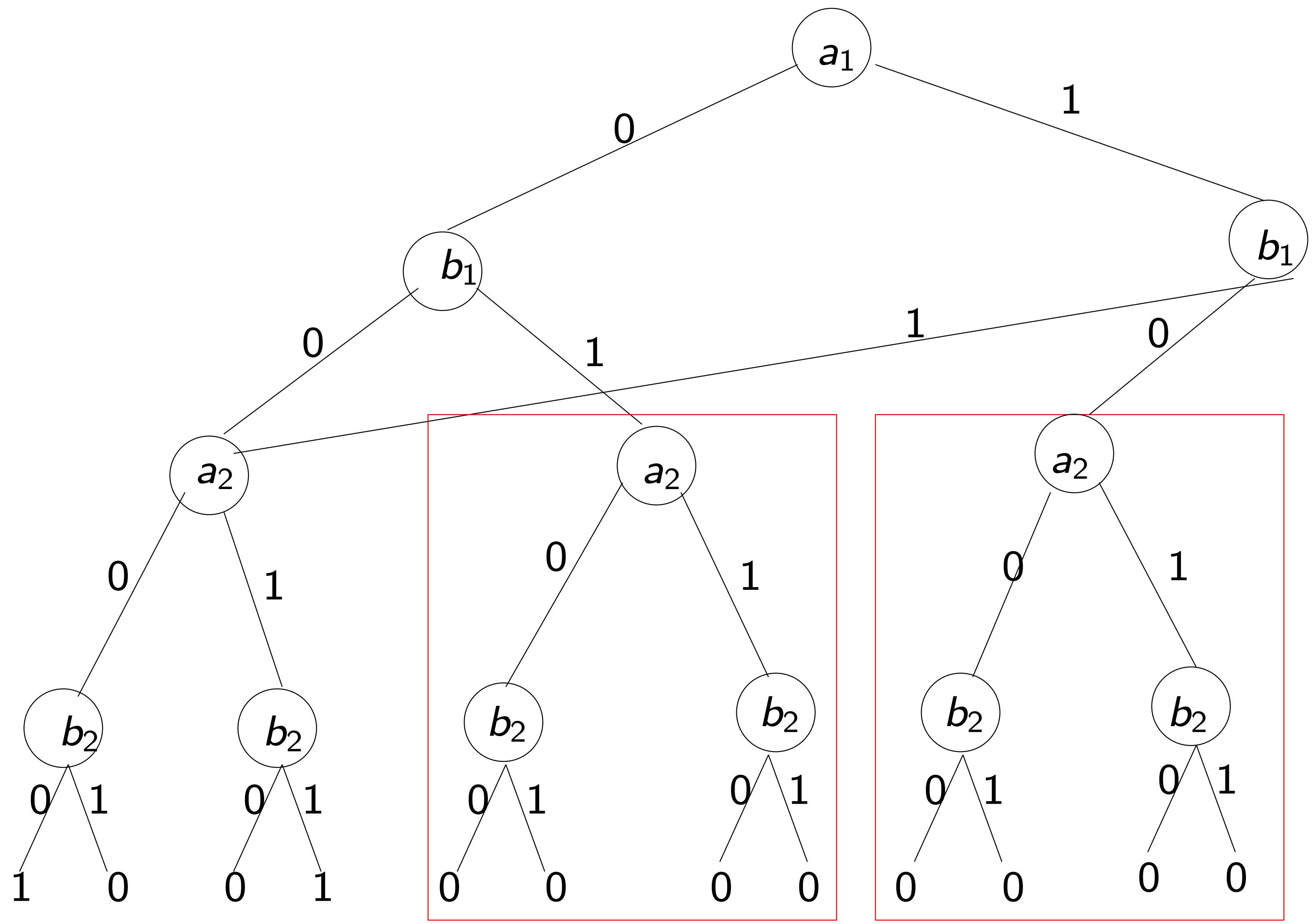
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



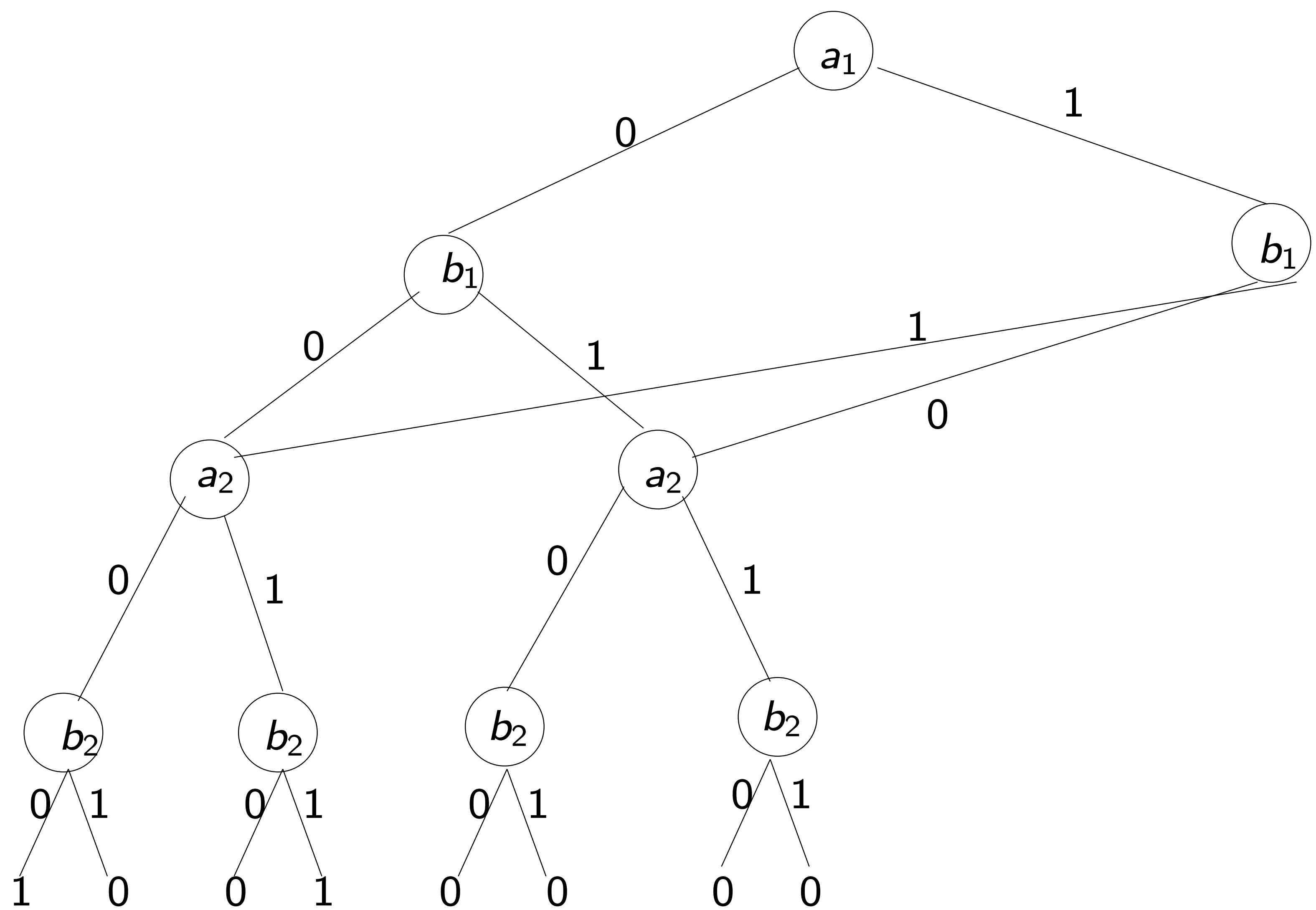
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



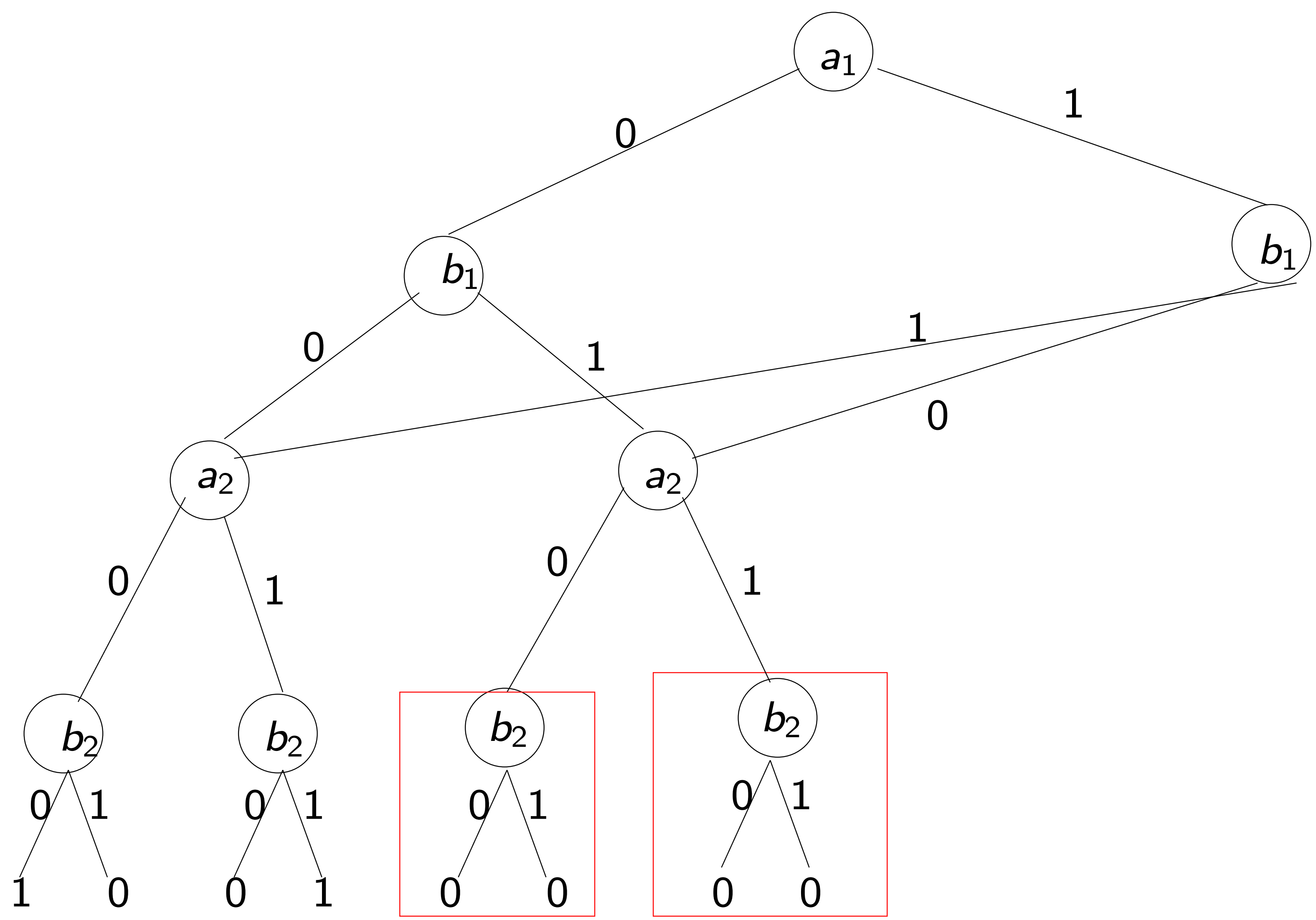
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



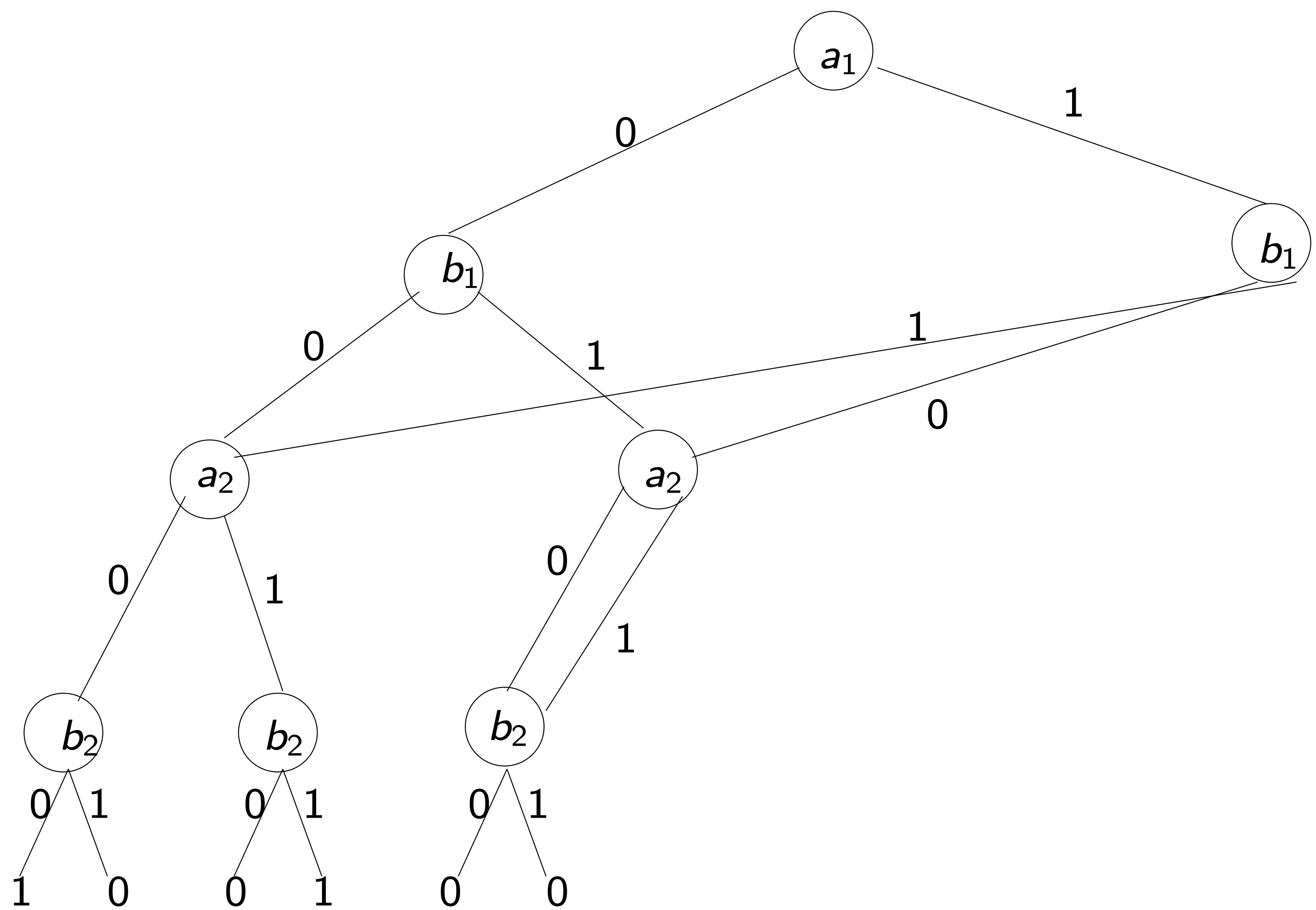
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



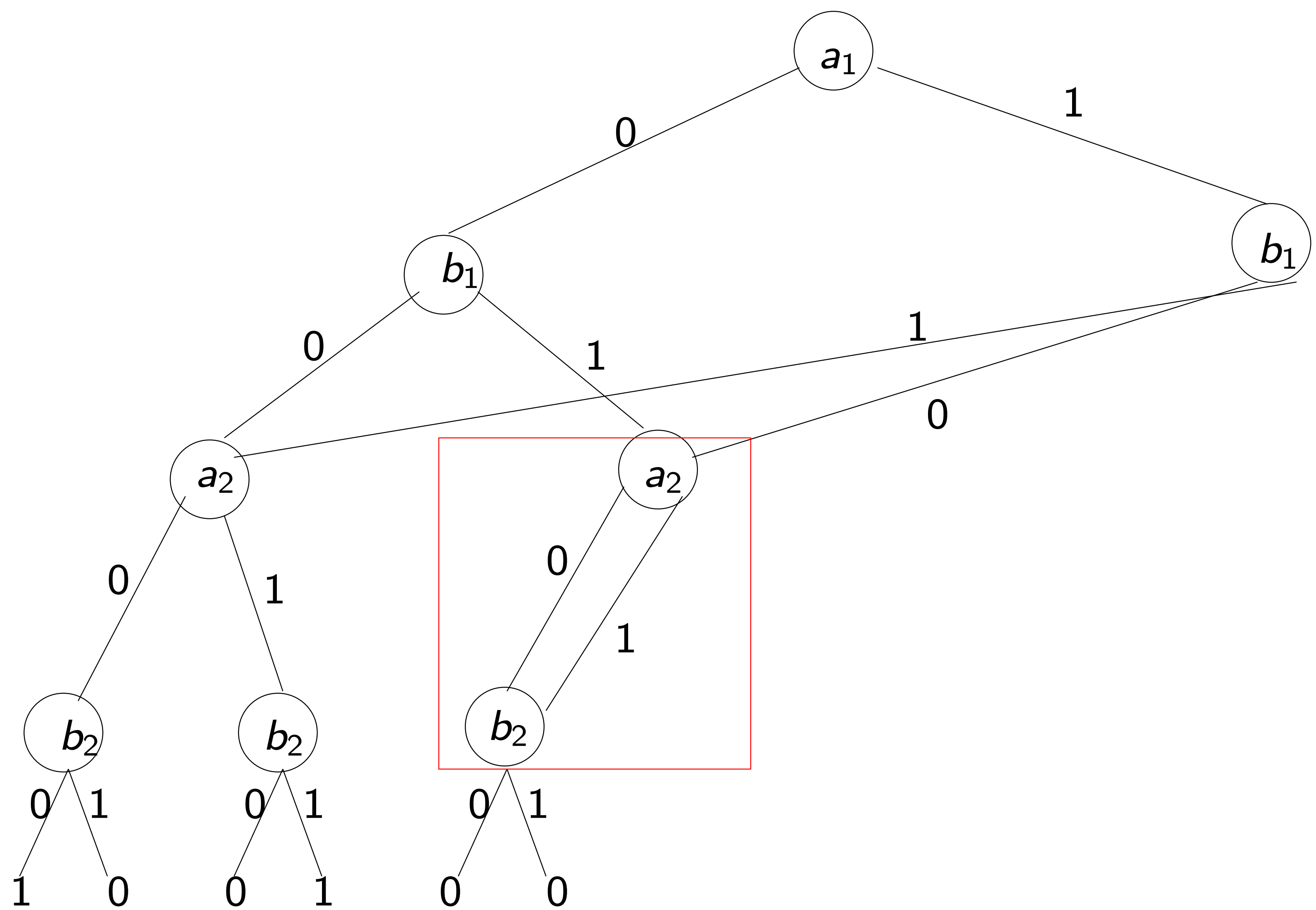
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



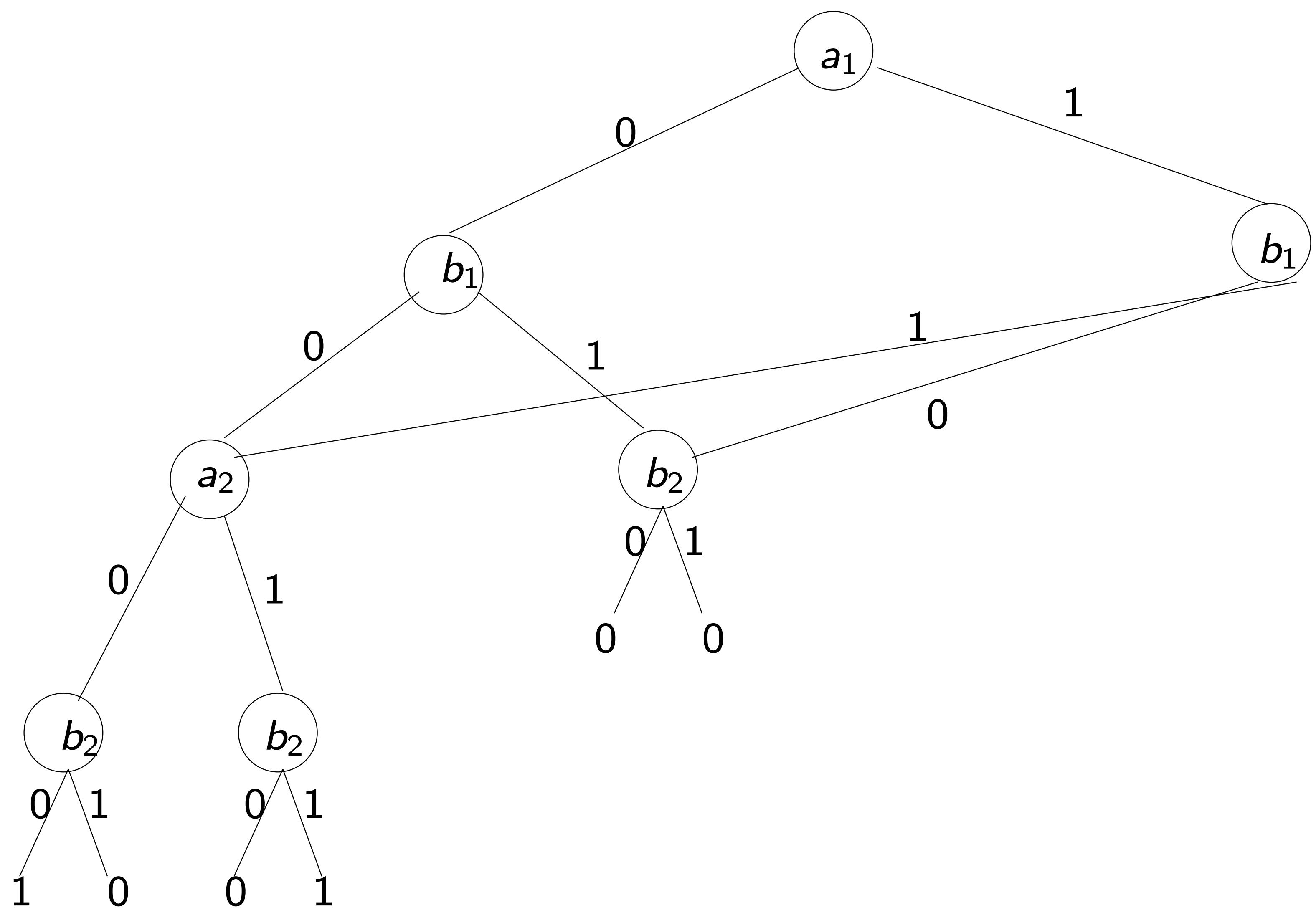
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



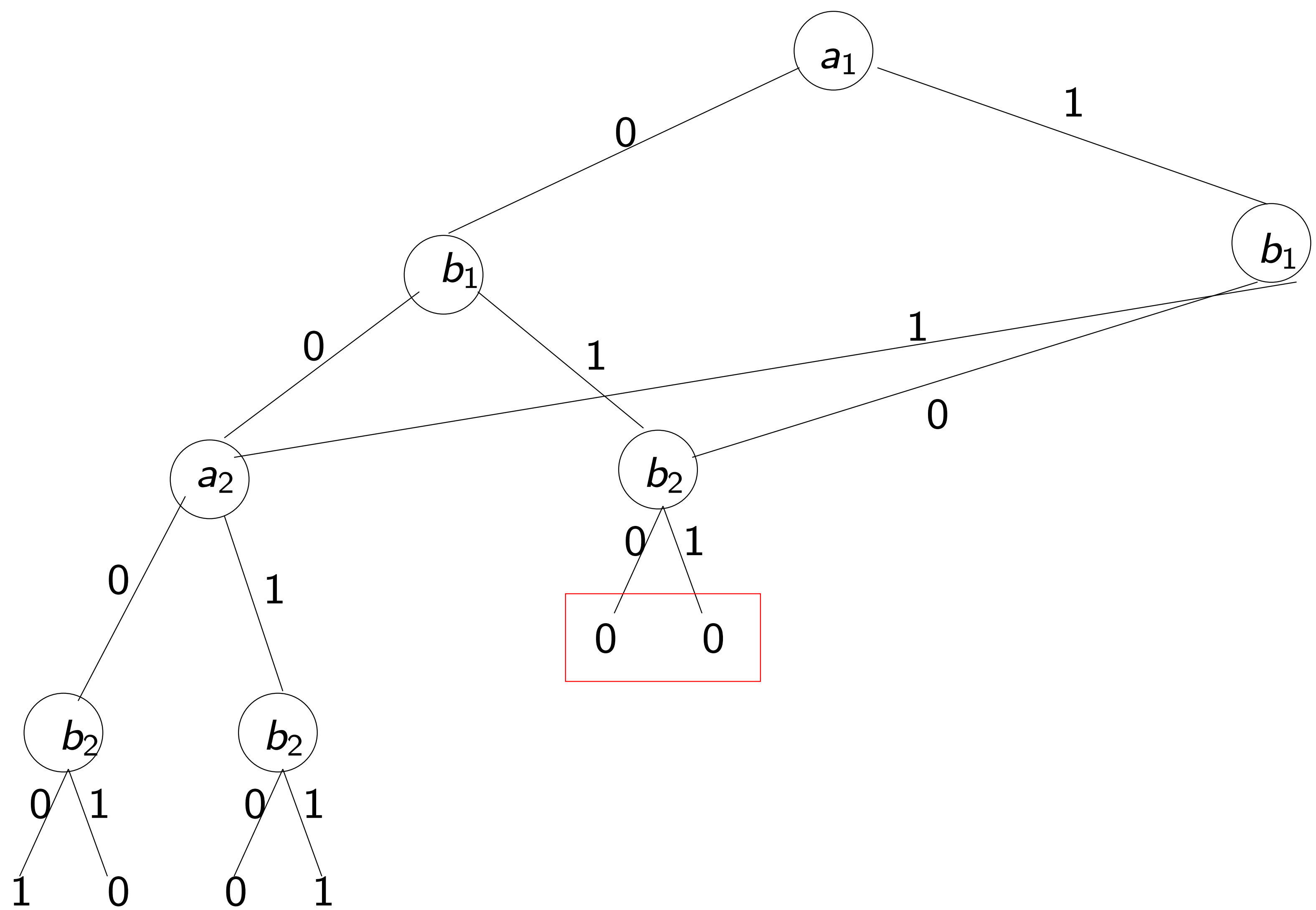
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



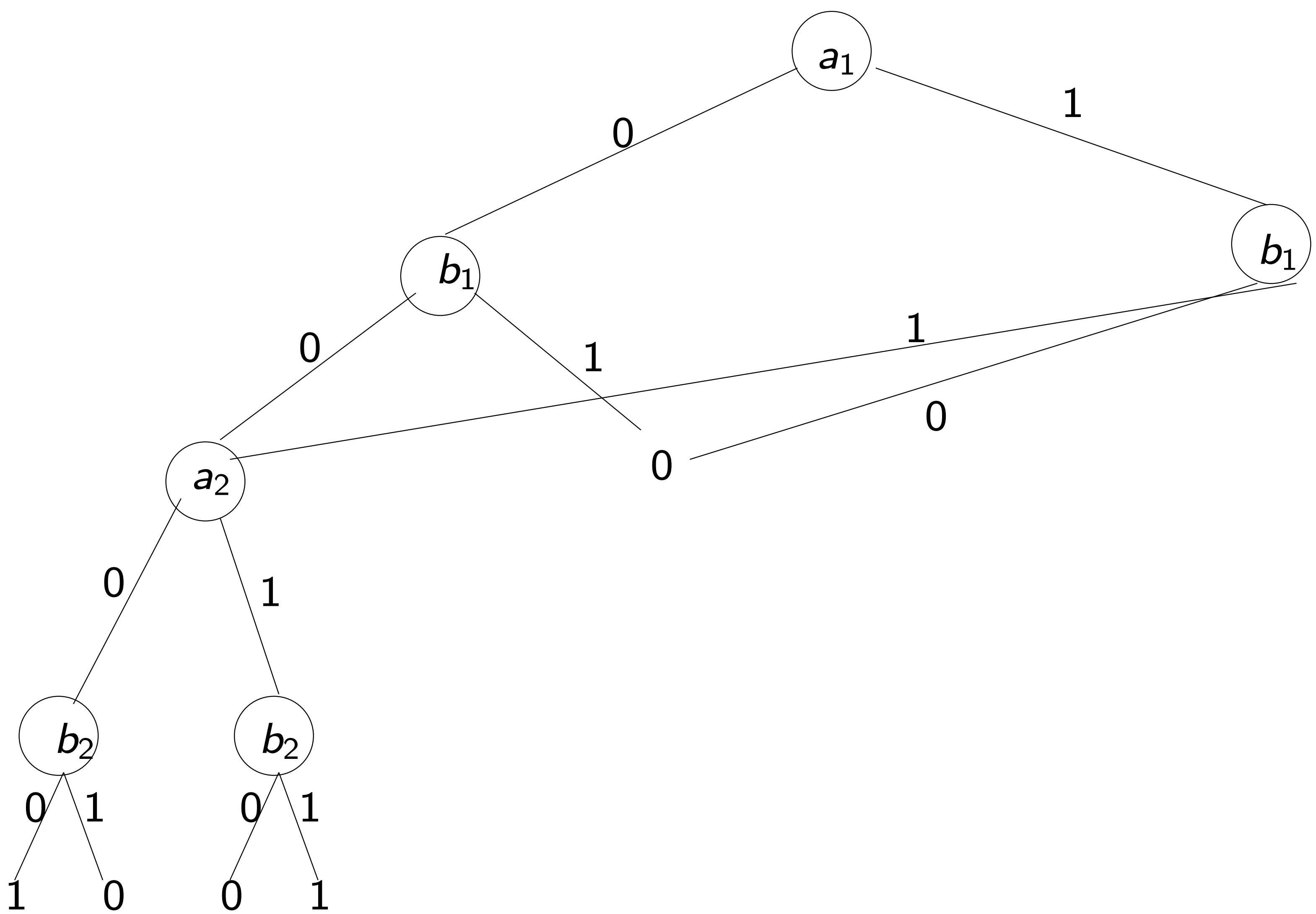
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



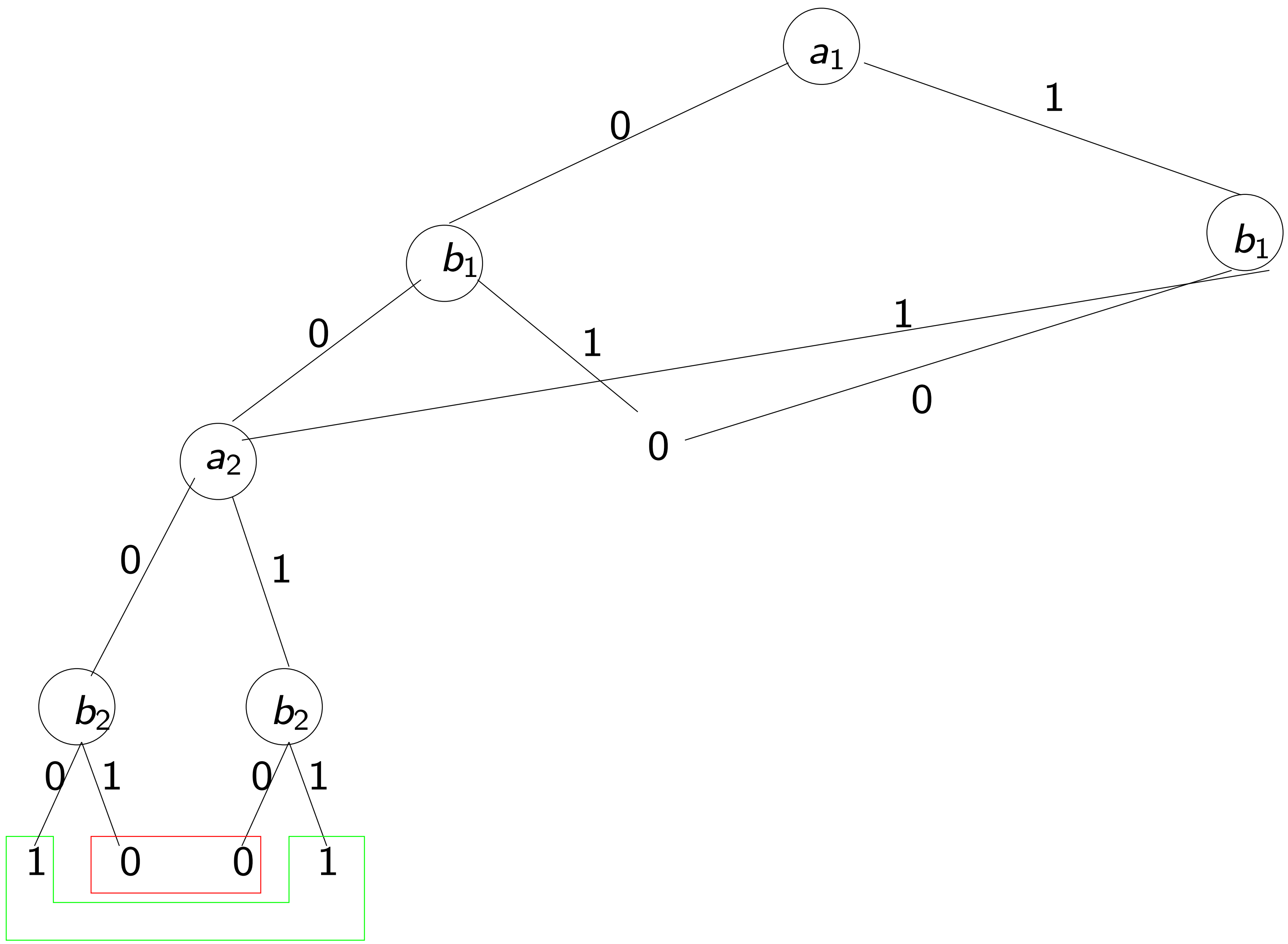
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



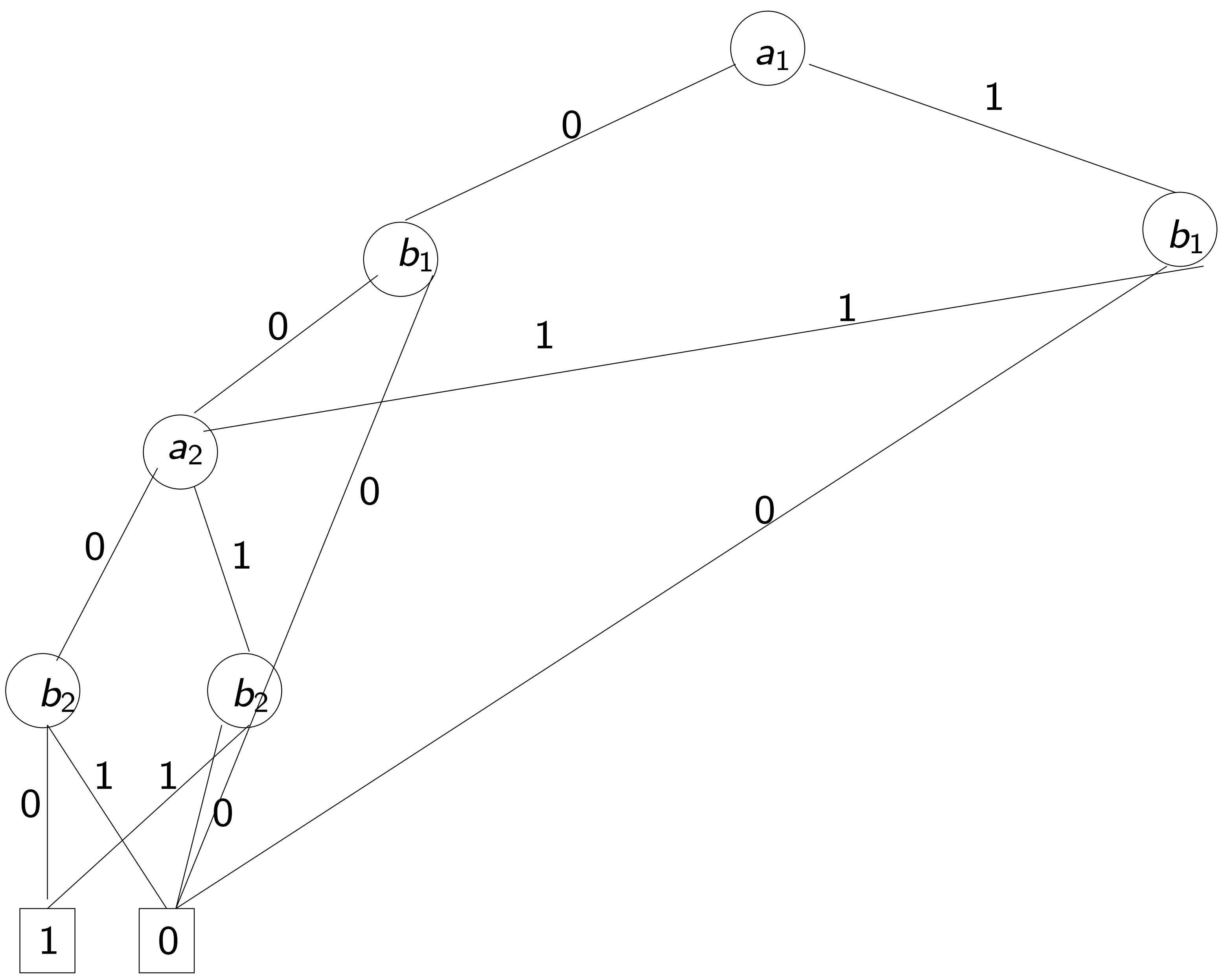
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



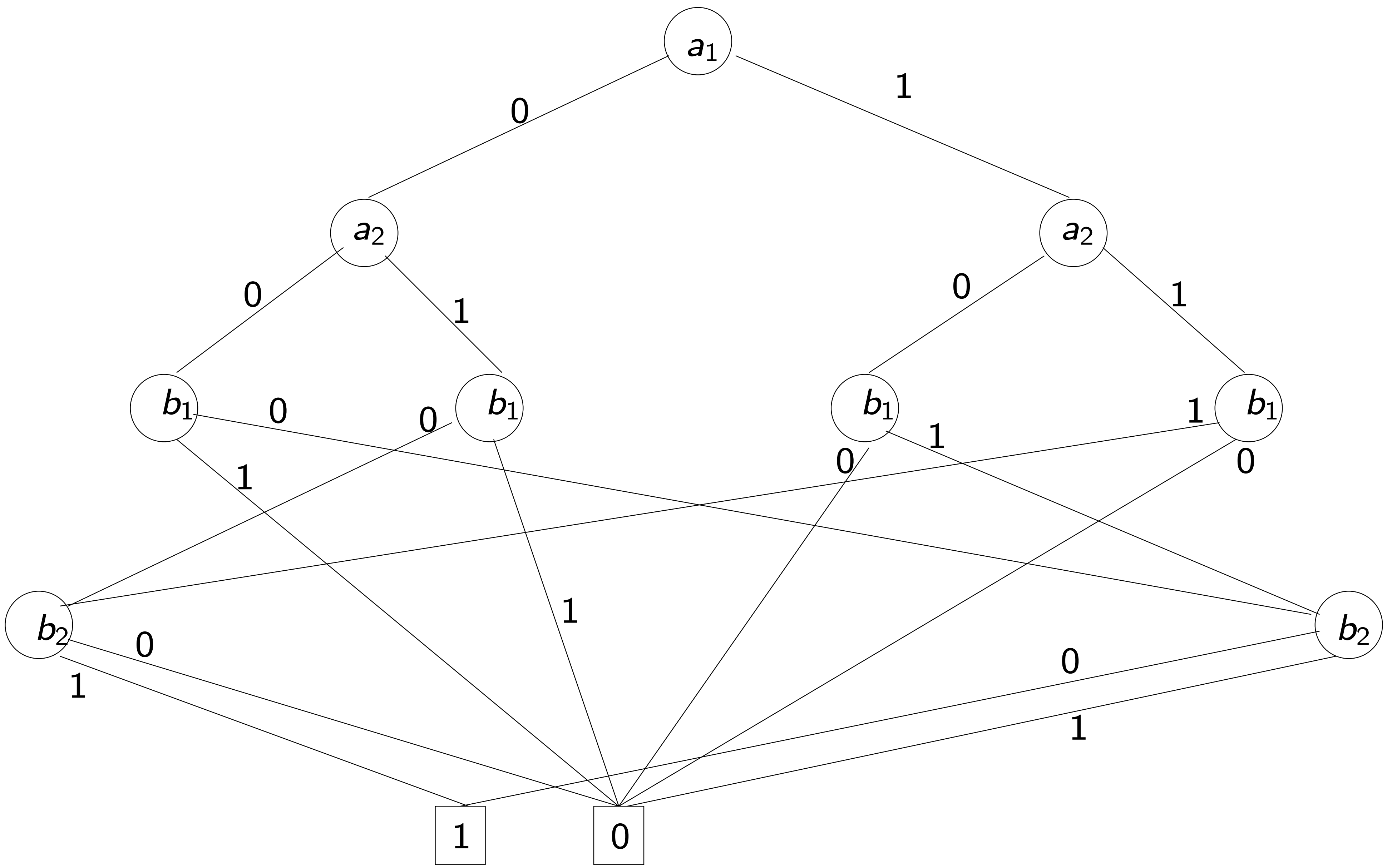
ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



ROBDD for 2-bit comparator with ordering $a_1 < b_1 < a_2 < b_2$



ROBDD for 2-bit comparator with ordering $a_1 < a_2 < b_1 < b_2$



Logical operations on ROBDD's (1)

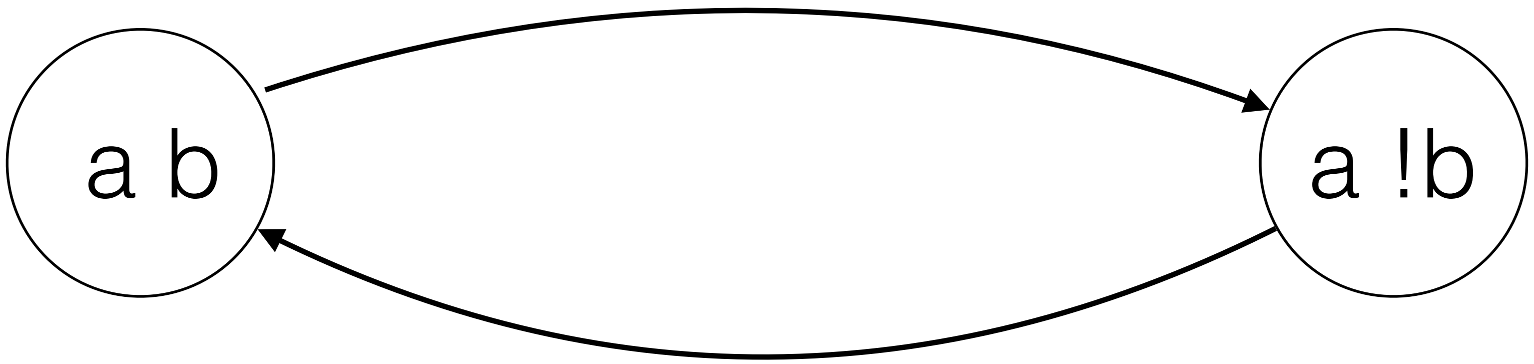
- Logical negation $\neg f(a, b, c, d)$

  Replace each leaf by its negation
- Logical conjunction $f(a, b, c, d) \wedge g(a, b, c, d)$
  - Use Shannon's expansion as follows

    $$f \wedge g = \neg a \wedge (f|_{\neg a} \wedge g|_{\neg a}) \vee a \wedge (f\ |_a \wedge g\ |_a)$$

    to break the problem into two sub-problems. Solve sub-problems recursively.
  - Always combine isomorphic subtrees and eliminate redundant nodes
  - Hash tables stores previously computed sub-problems
  - Number of sub-problems bounded by $|f| \cdot |g|$

Simple example



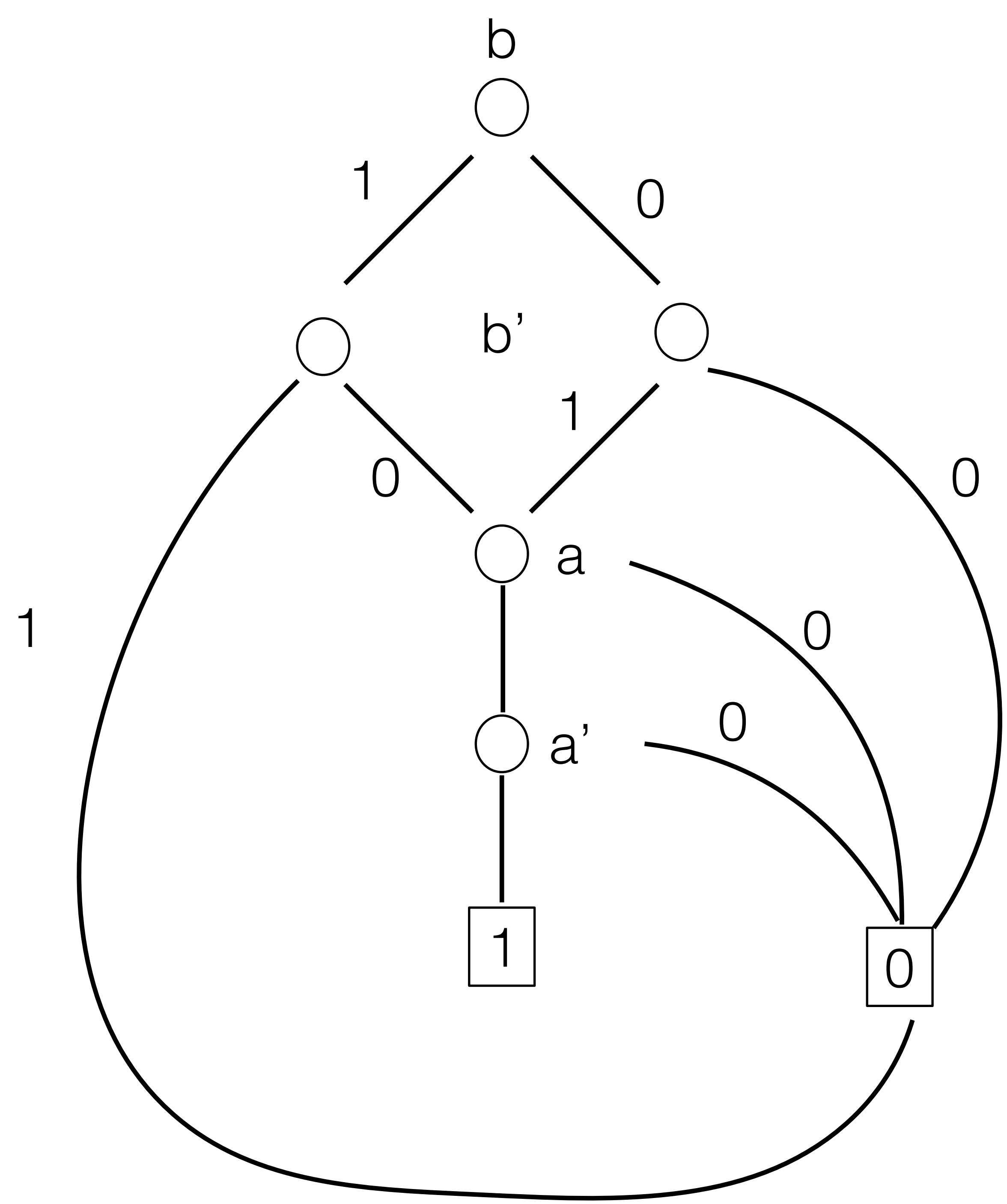
Transition relation as characteristic function

T(a,b,a’,b’) = (a & !b & a’ & b’) | (a & b & a’ & !b’)

Represent as a ROBDD!

ROBDD for the example

Ordering = b b’ a a’



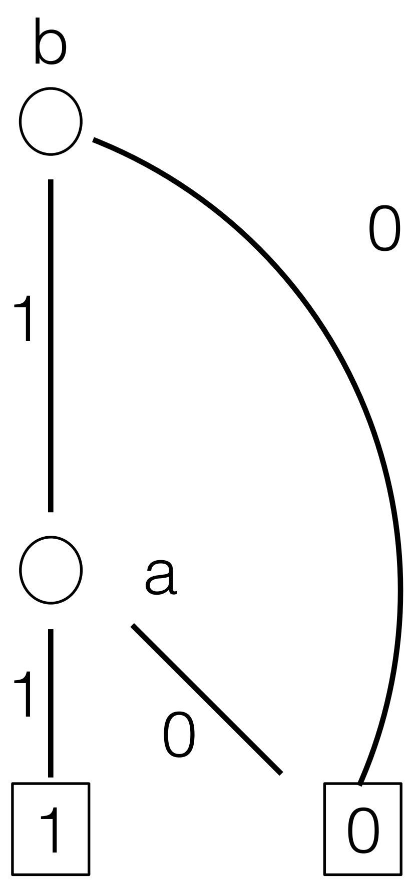
Forward image as existential quantifier

$$Fwd(P, T) = \{s' | \exists s . s \in P \land (s', s) \in T\}$$

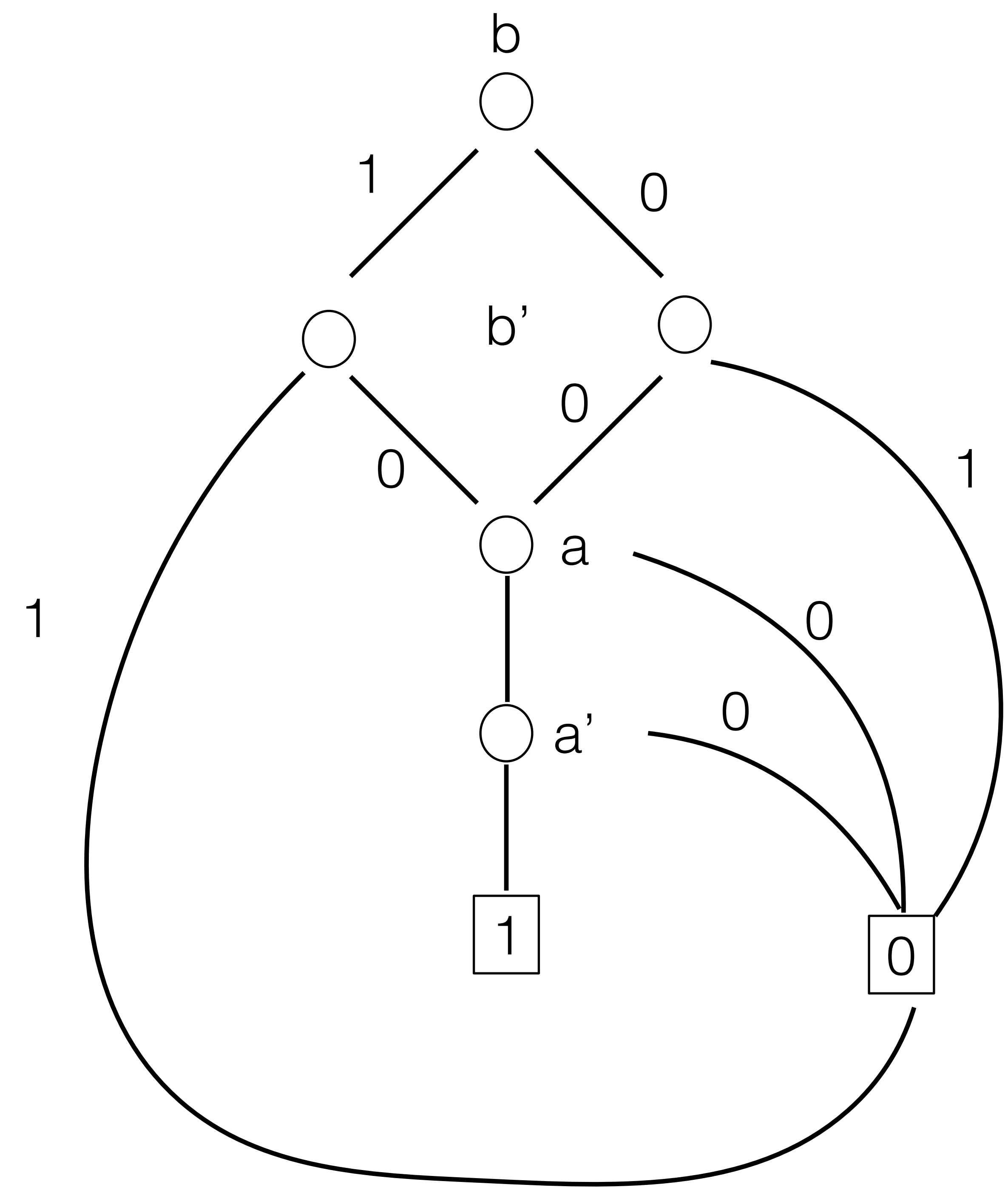
Operation on ROBDD:

- By definition: Exists a: f = f | !a or f | a
- Replace all a-nodes by negative sub-tree
- Replace all a-nodes by positive sub-tree

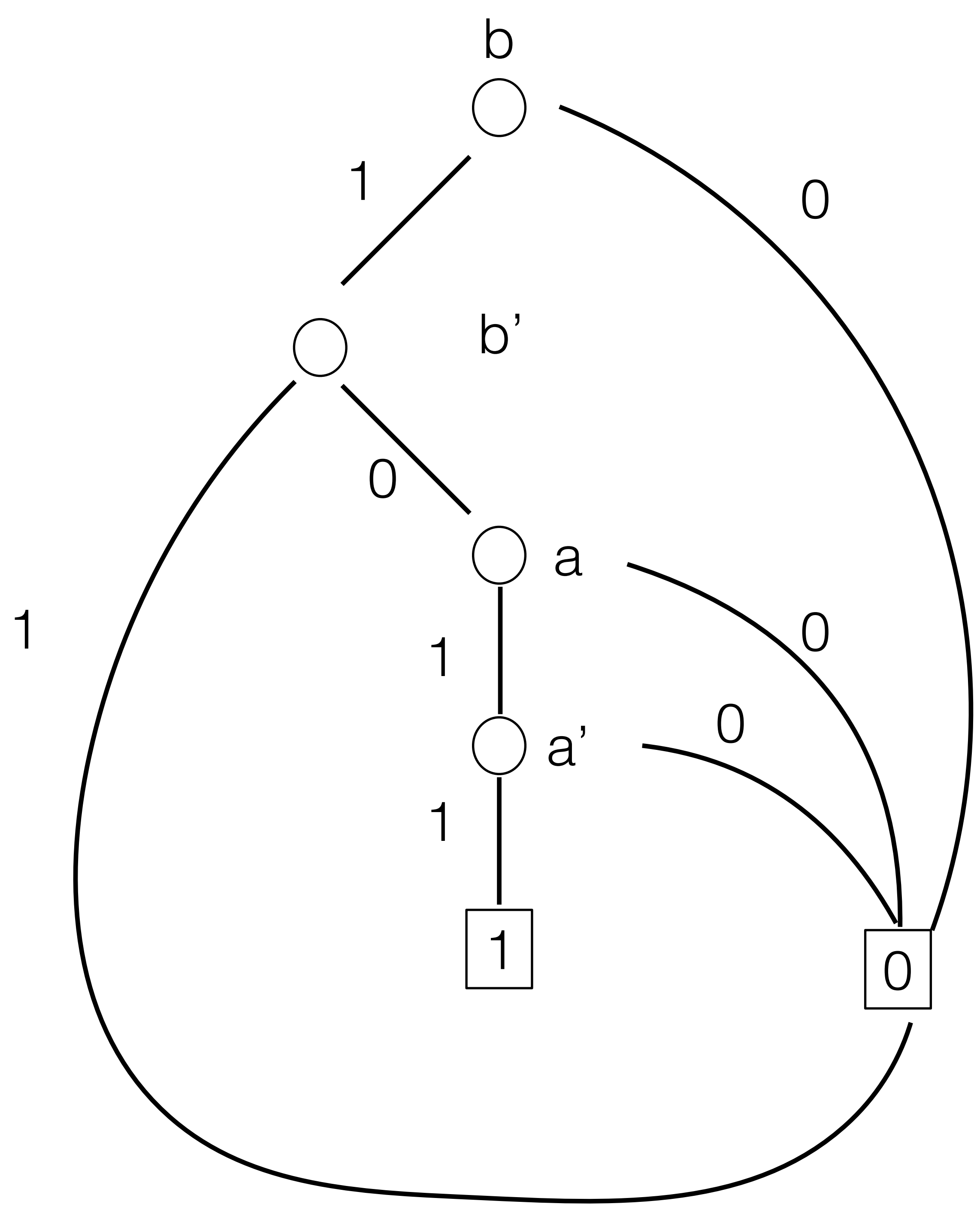
States in current & in transition relation



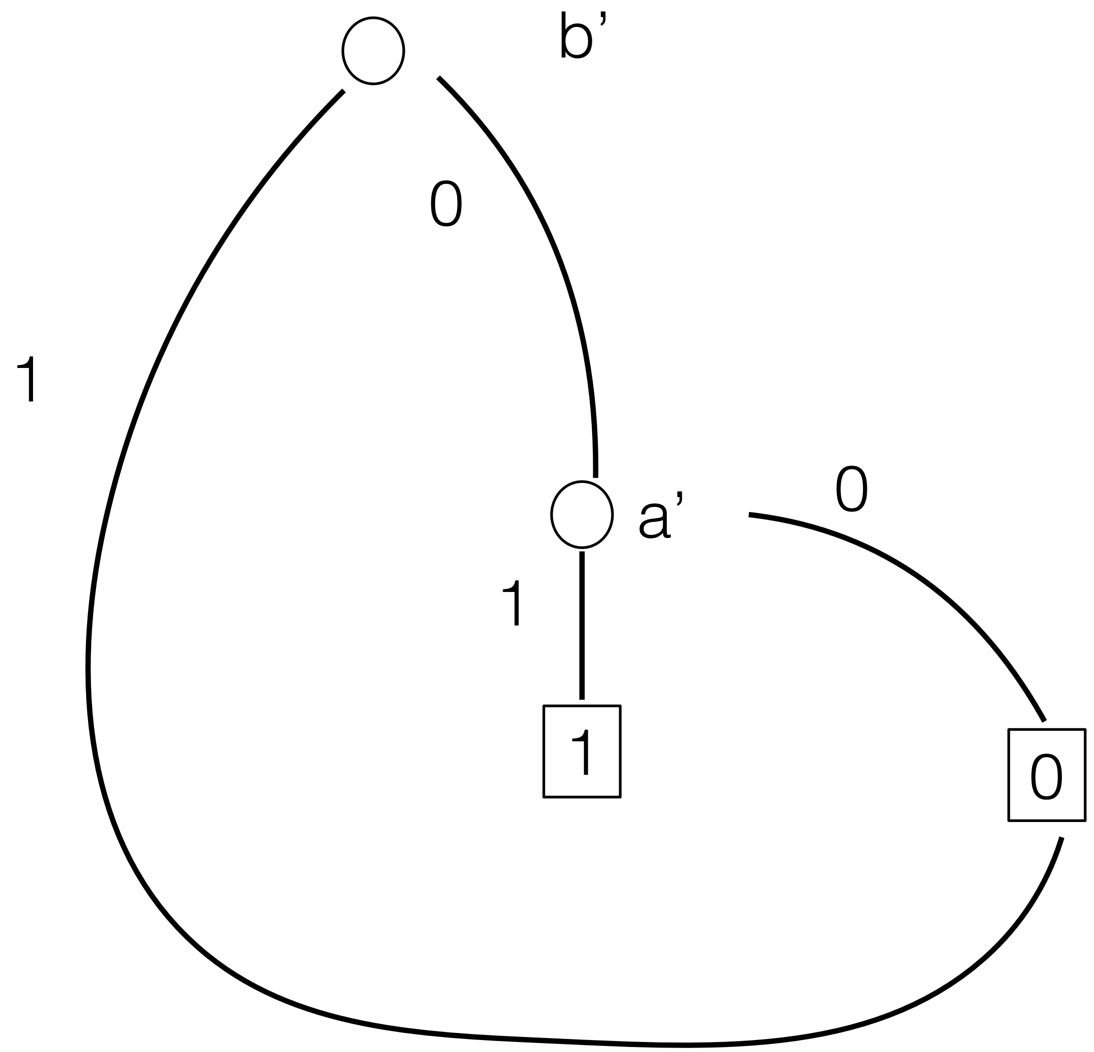
&



States in current & in transition relation

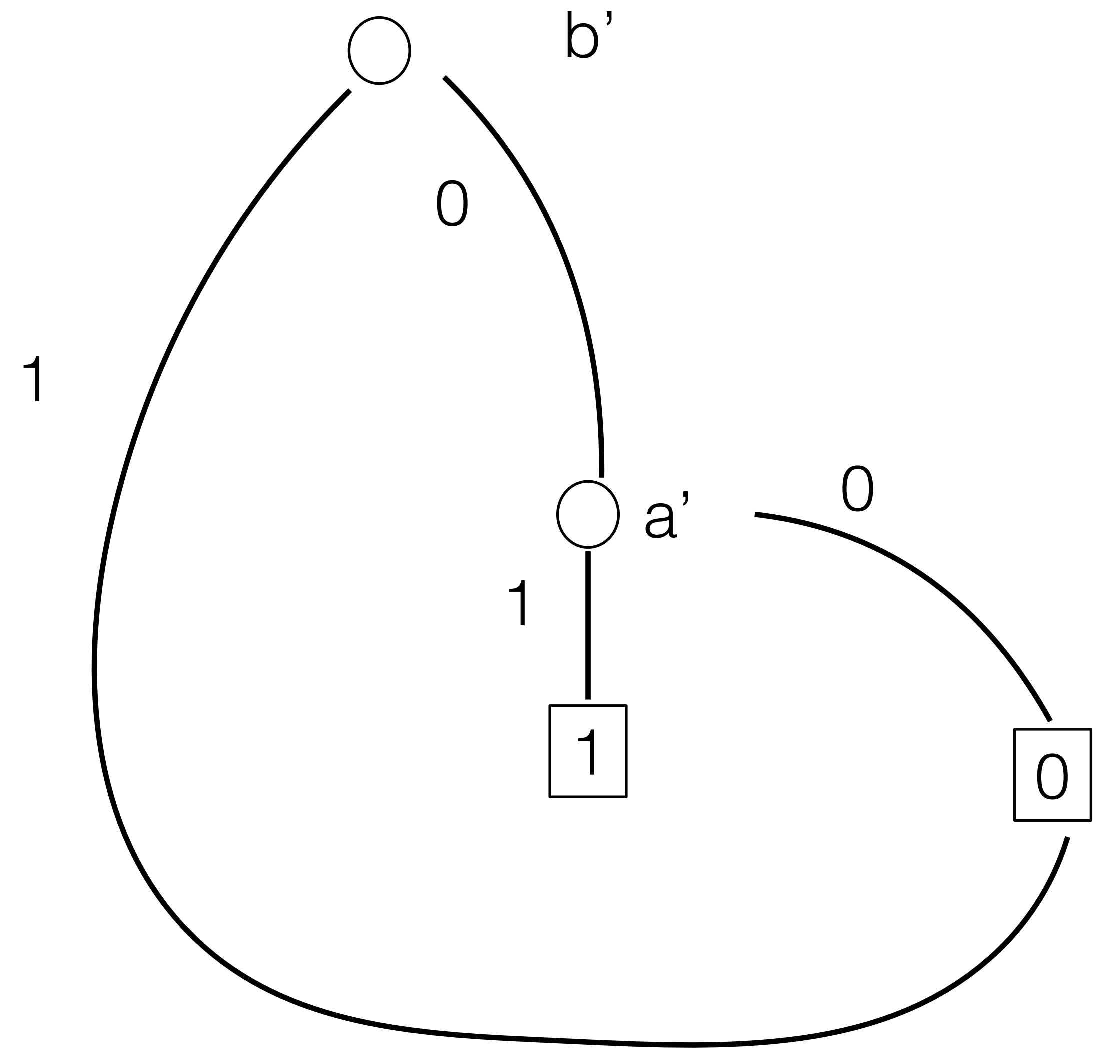


Existential quantifier on a and b



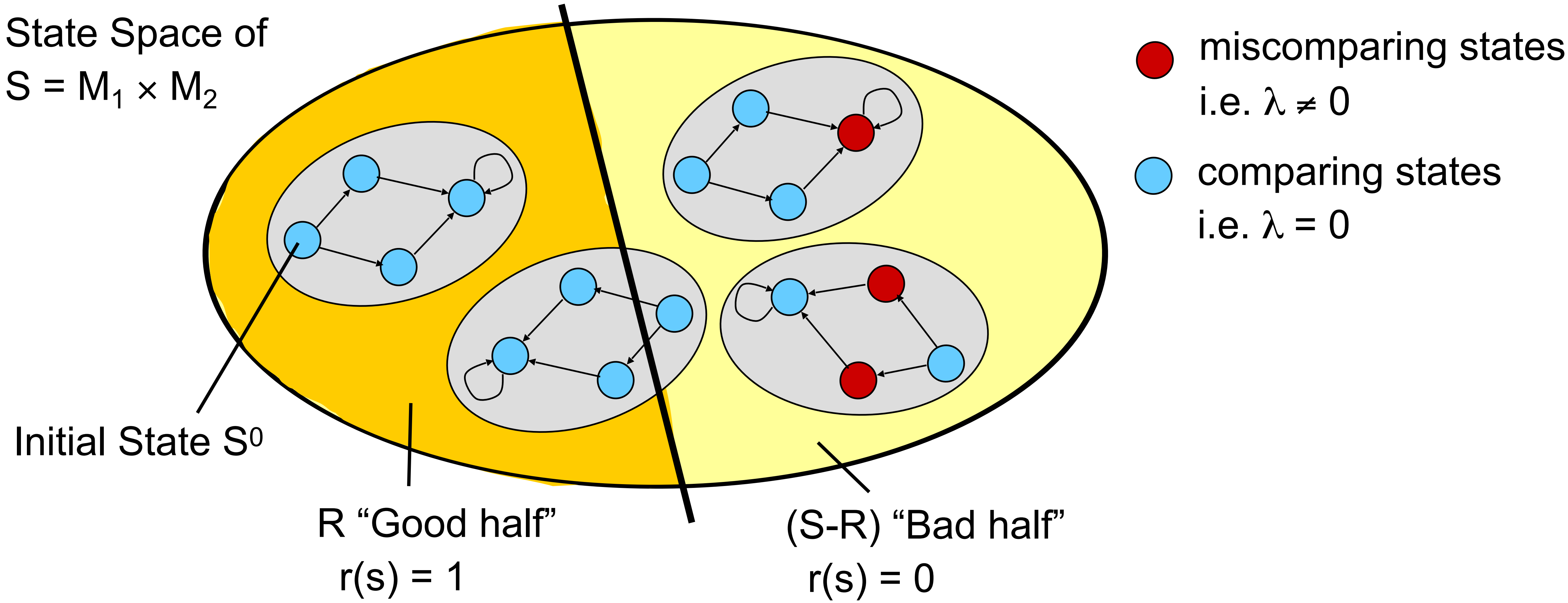
Exercise compute one more

As an exercise, compute the set of states reach from this state.



Reachability using SAT

General Approach to SEC



Inductive proof of equivalence:

Find subset $R \subseteq S$ with characteristic function $r: S \rightarrow \{0,1\}$ such that:

1. $r(s^0) = 1$ (initial state is in good half)
2. $(r(s) = 1) \Rightarrow r(\delta(x,s)) = 1$ (all states from good half lead go to states in good half)
3. $(r(s) = 1) \Rightarrow \lambda(x,s)) = 0$ (all states in good half are comparing states)

Soundness and Completeness

- With a candidate for R we can:
  - prove equivalence
    - that means the method is “sound”
    - we will not produce “false positives”
  - but not disprove it:
    - that means the method is “incomplete”
    - we may produce “false negatives”

Inductive proofs

Base case: P(s0)

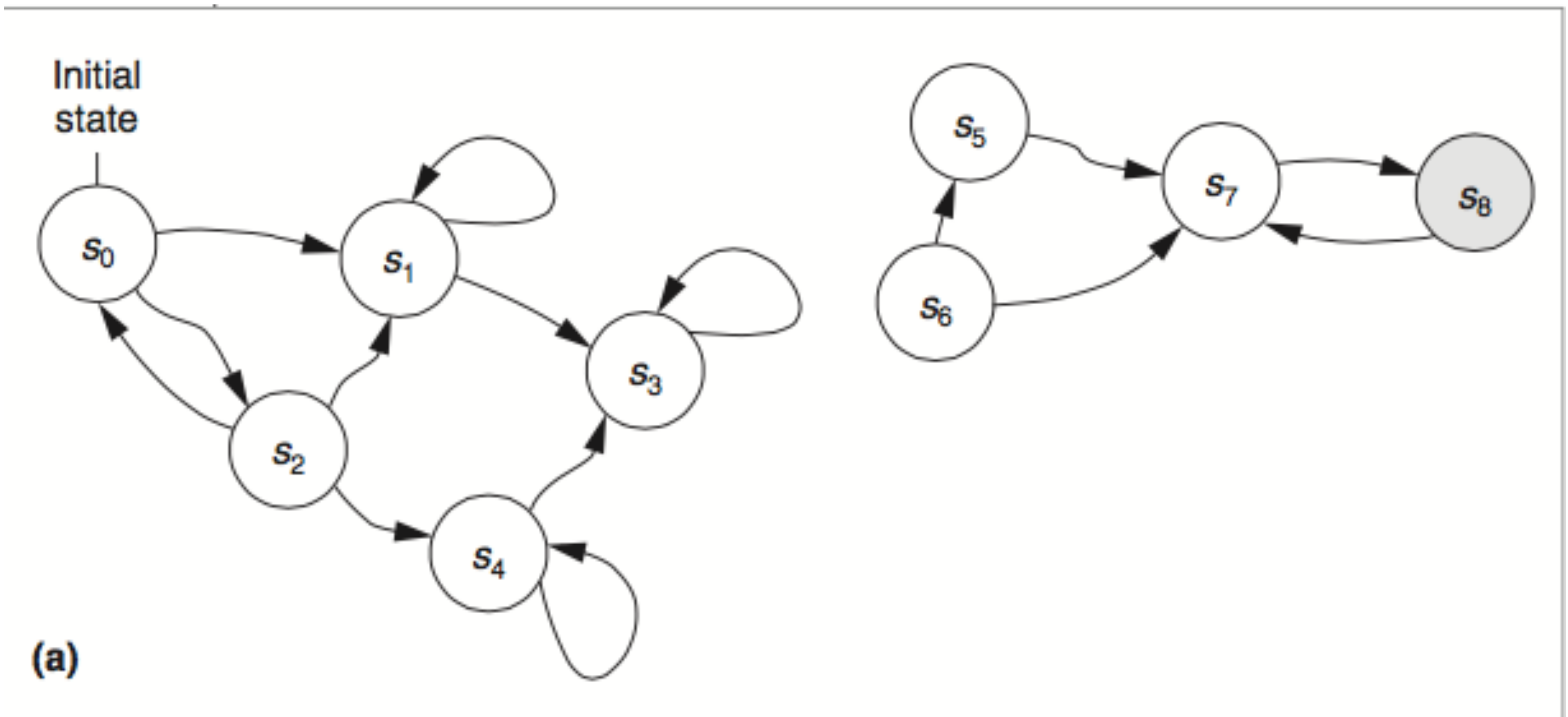
Induction step: P(s) & T(s,s’) implies P(s’)

Note that both steps can be solved using a SAT solver.
We will come back to this when we will talk about BMC.

Issues: properties not always inductive

There might be a transition from an unreachable state to a bad state.

Example violating induction step



Transition from s7 to s8 (bad state)

k-induction

Generalise to a given number of steps, called the induction depth.

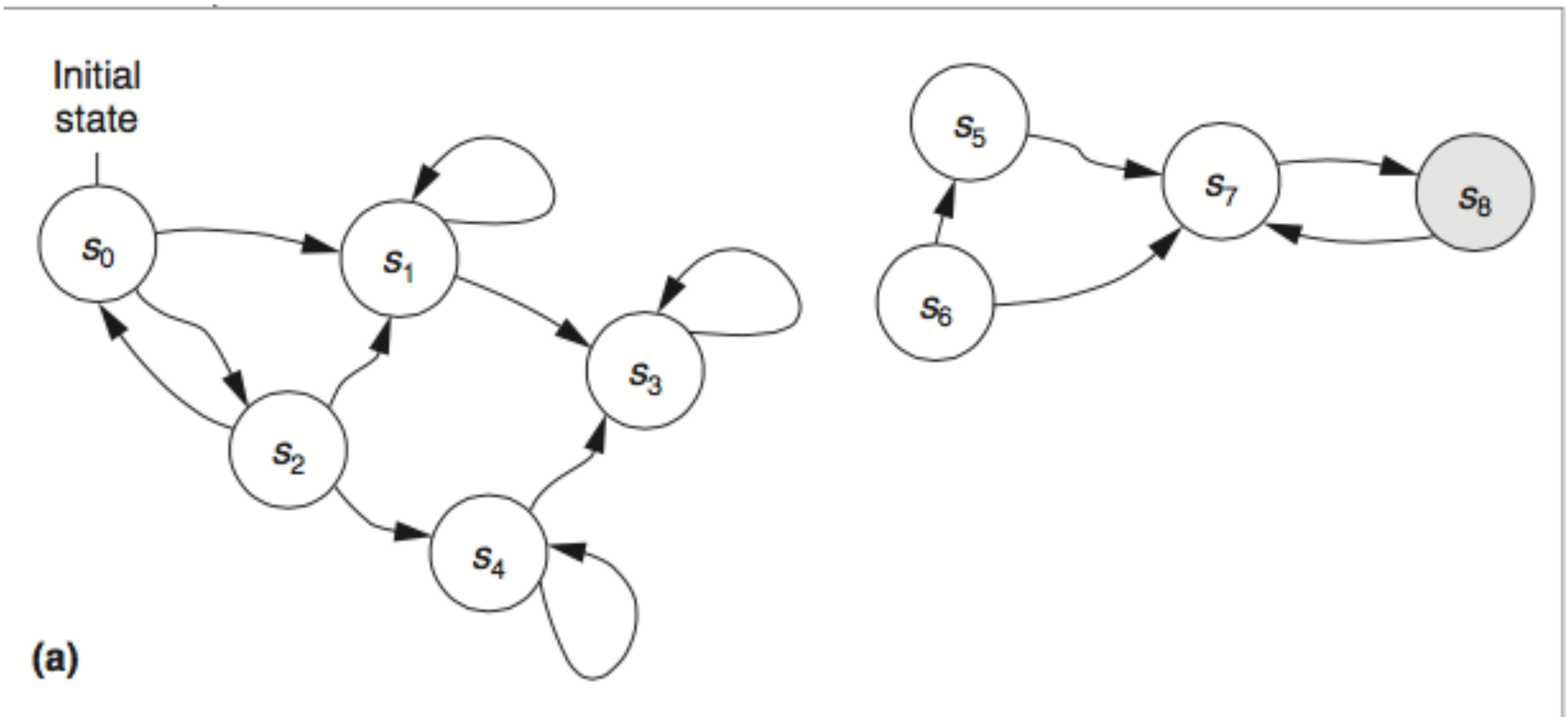
Base case at depth i:

$$P(s_0) \wedge \exists \pi(s_0, s_i) \rightarrow P(s_0) \wedge ... \wedge P(s_i)$$

Induction step:

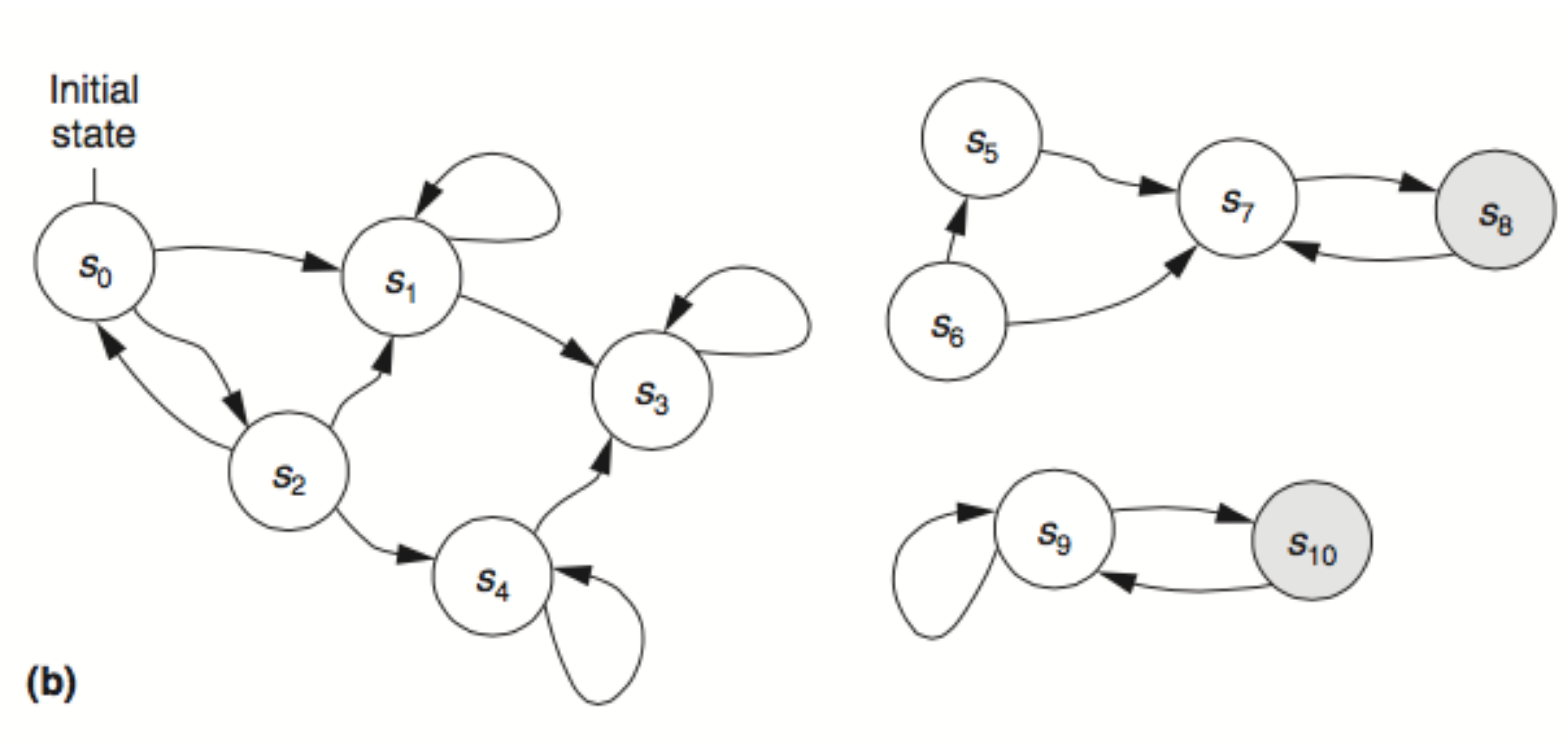
$$\exists \pi(s_0, s_{i+1}) \rightarrow P(s_{i+1})$$

Show again on example from paper



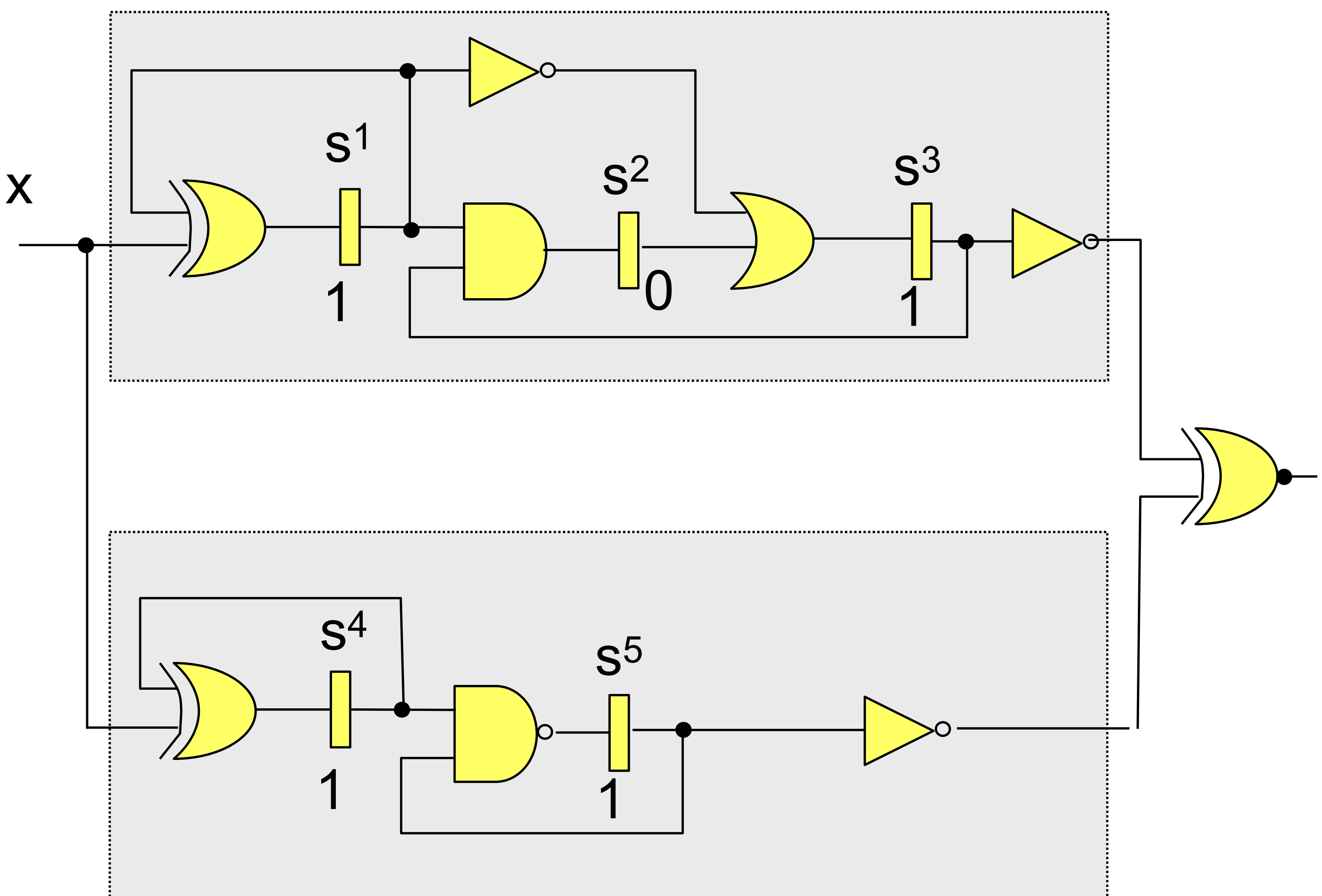
Transition from s6 to s8 (bad state). To succeed depth = 4.

Sometimes no depth work



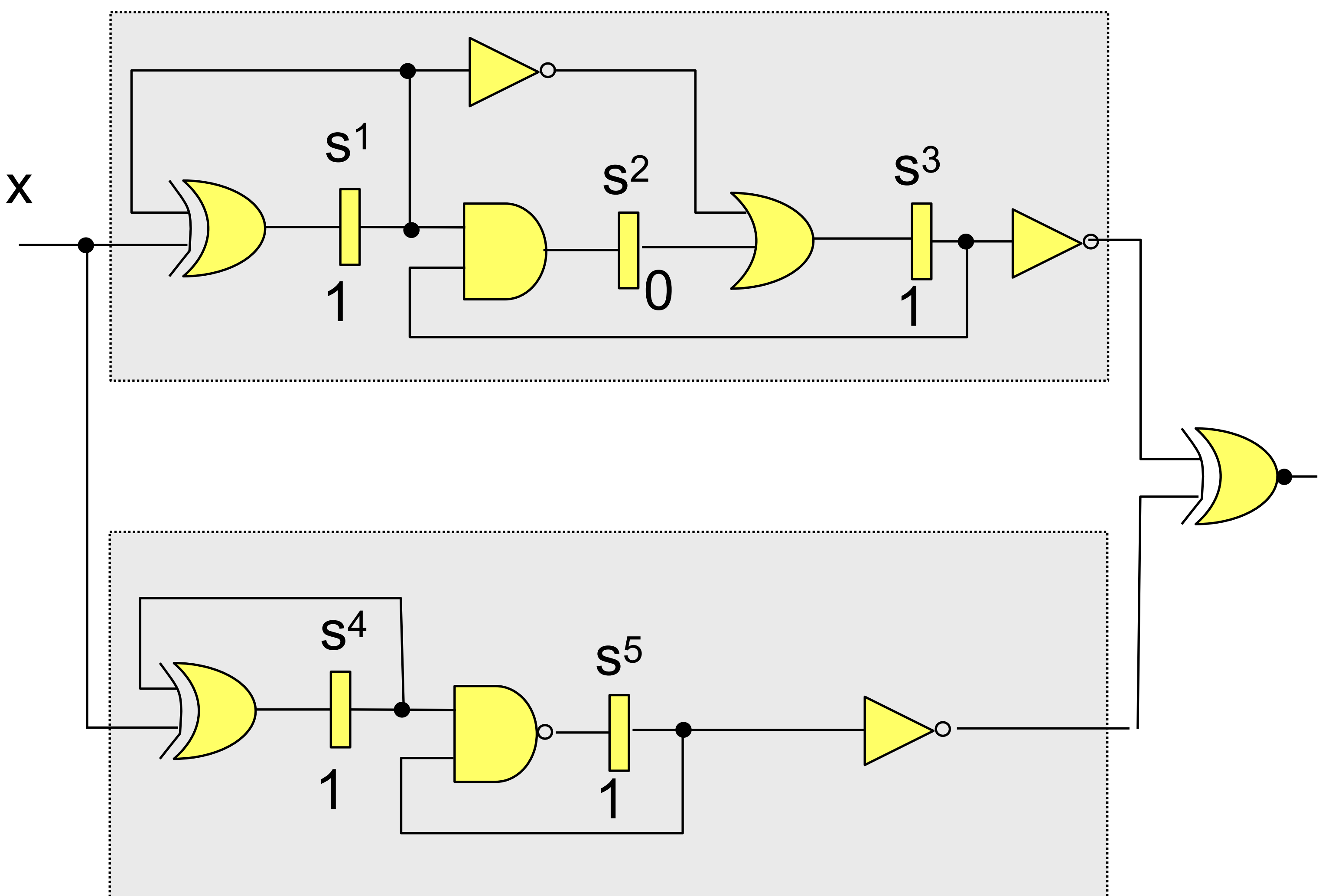
Solved by removing duplicate states in induction step.

Product machine - state traversal



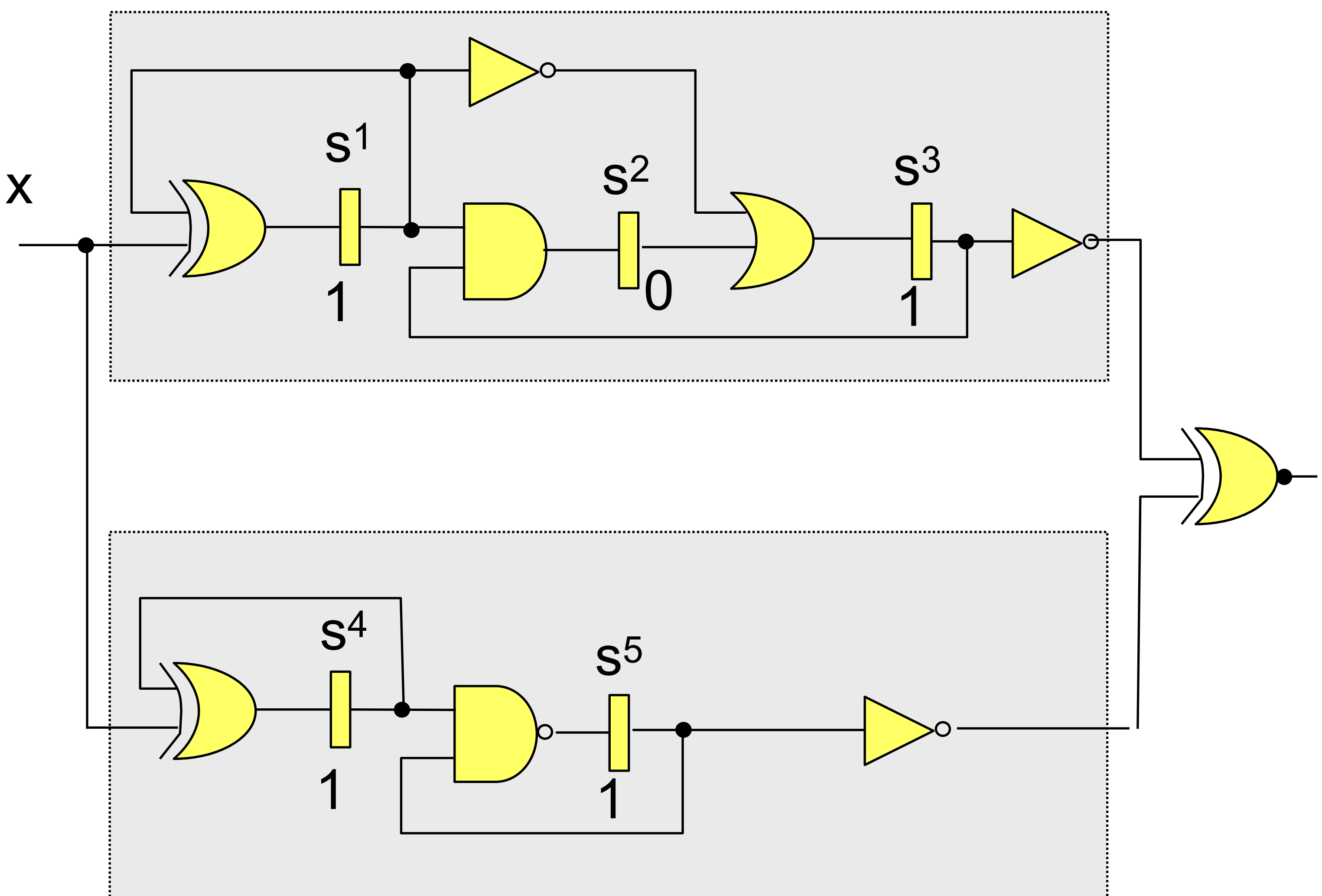
As an exercise, compute forward and backward traversal on this example.

Product machine with induction



As an exercise,
check the depth needed
to prove equivalence
using induction.

Another question



For which initial states are the machines equivalent? What which ones are they different?

When they are different, can you give a distinguishing sequence?

Summary

We looked at combinational and sequential equivalence.

We looked at reachability analysis: forward, backward, symbolic, and k-induction.

We looked at different representations for Boolean functions: AIGs, BDDs.