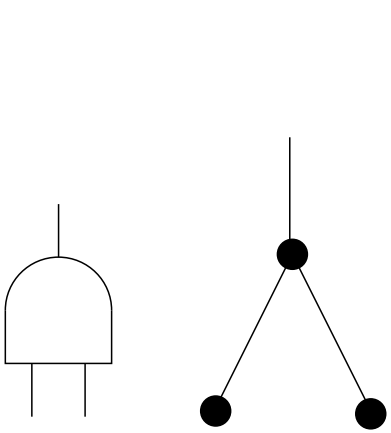
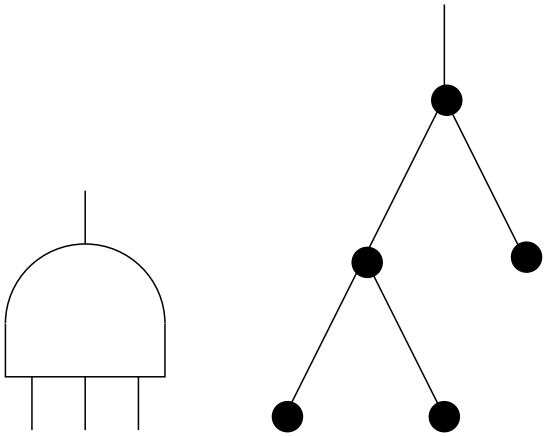
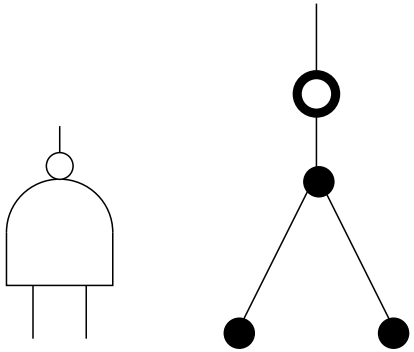
And Inverter Graphs



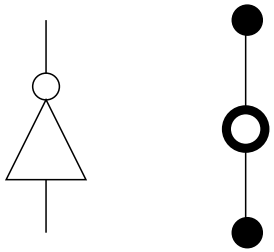
and



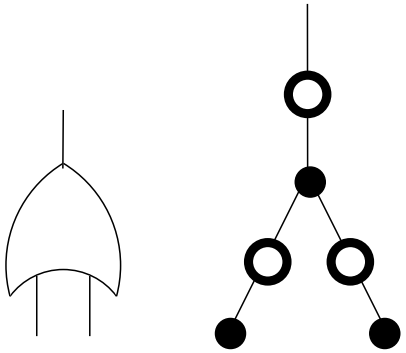
and



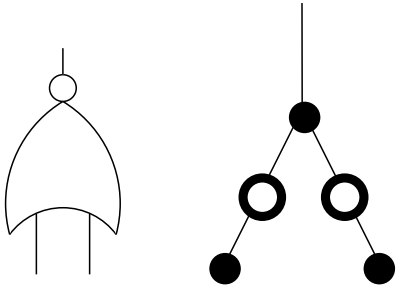
nand



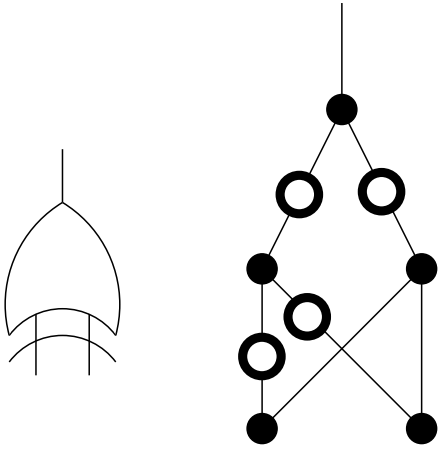
inverter



or

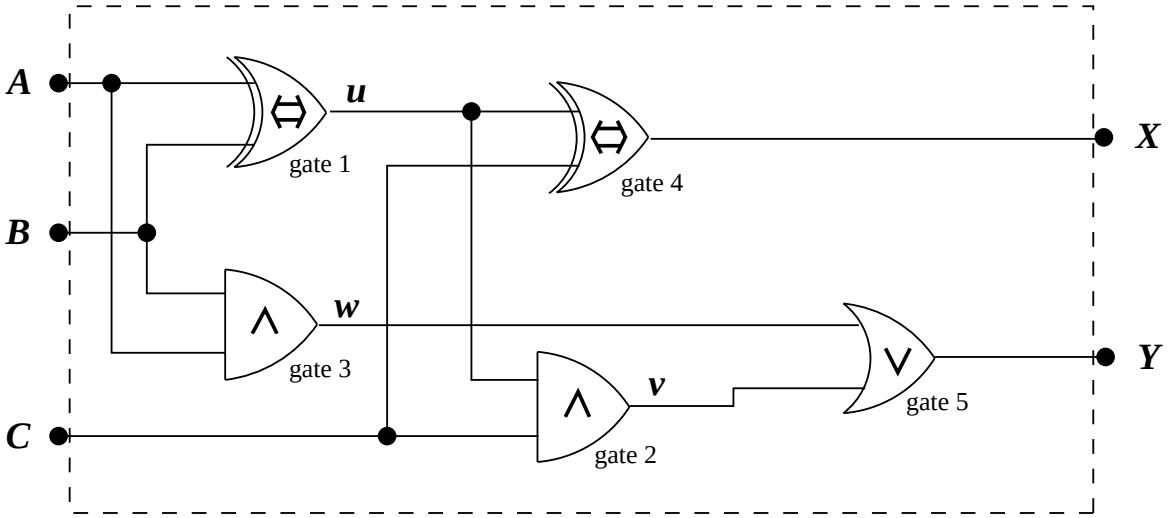


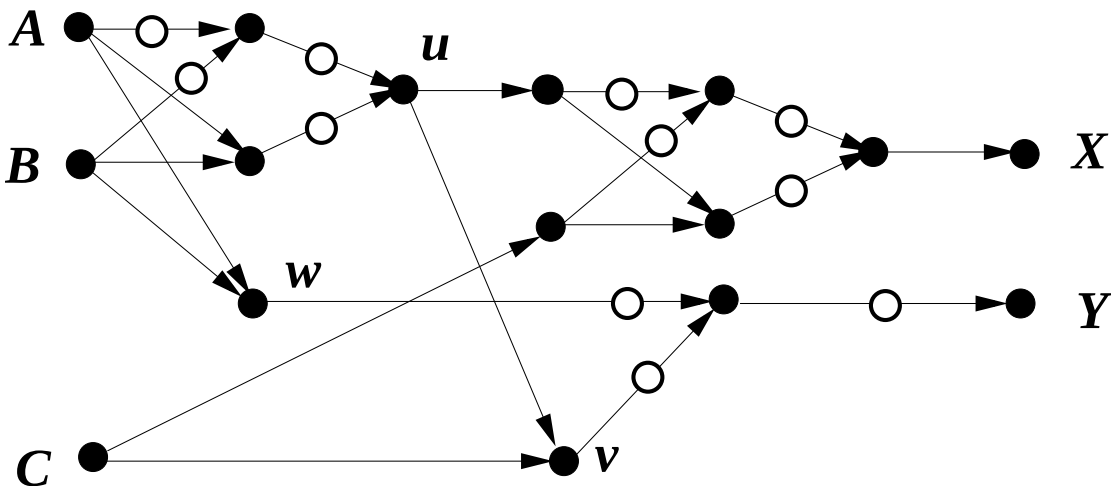
nor



xor

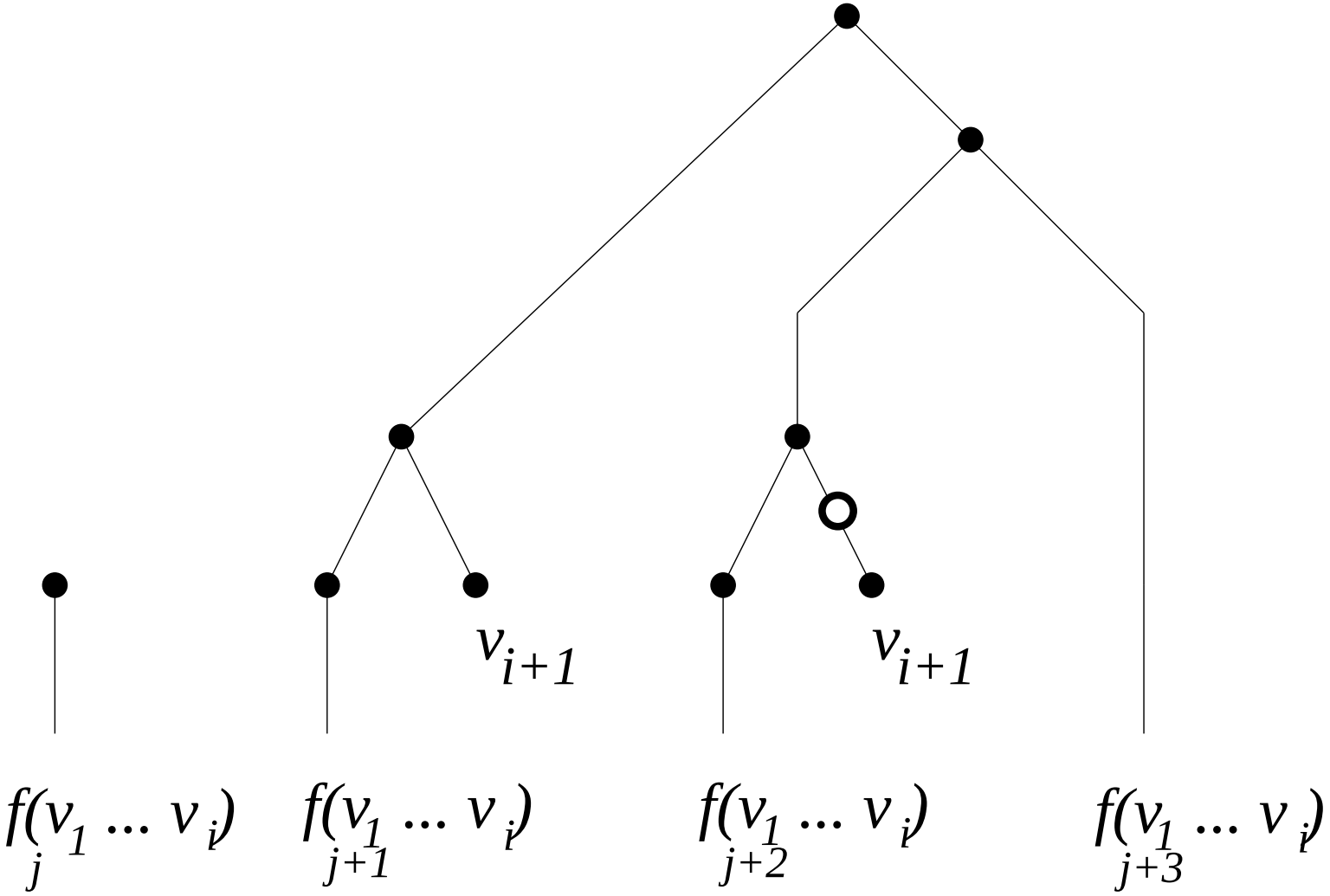
And Inverter Graphs





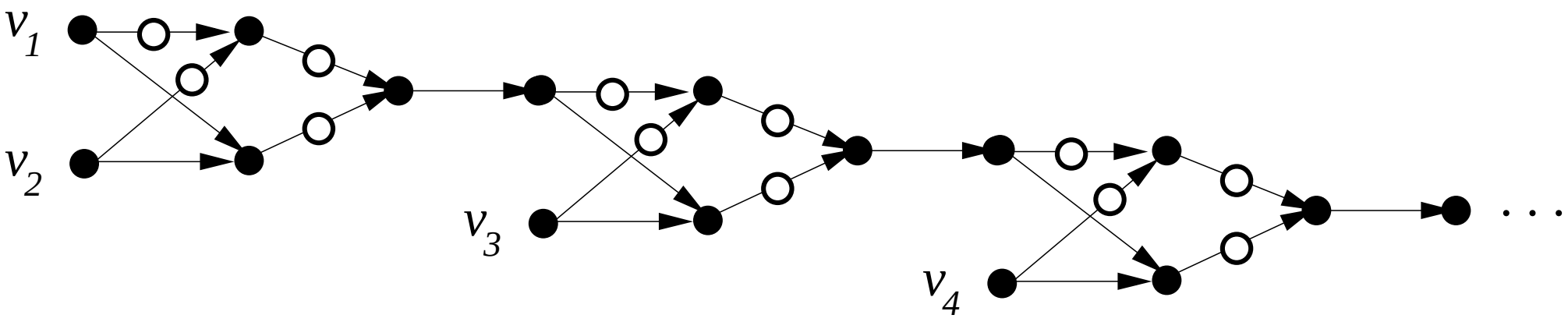
And Inverter Graphs

Can represent any Boolean function:



And Inverter Graphs

Can be small vs. CNF representation:



linear in AIG but exponential in CNF or in DNF

$$(v_1 \vee v_2 \vee v_3 ...) \wedge (\bar{v}_1 \vee \bar{v}_2 \vee v_3 ...) \wedge (\bar{v}_1 \vee v_2 \vee \bar{v}_3 ...) ... (\bar{v}_1 \vee \bar{v}_2 \vee \bar{v}_3 \vee \bar{v}_4 \vee v_5 ...) ...$$

And Inverter Graphs

Can be small vs. BDD representation:

Consider an integer multiplier for word size n

with outputs numbered 0 to $2n - 1$.

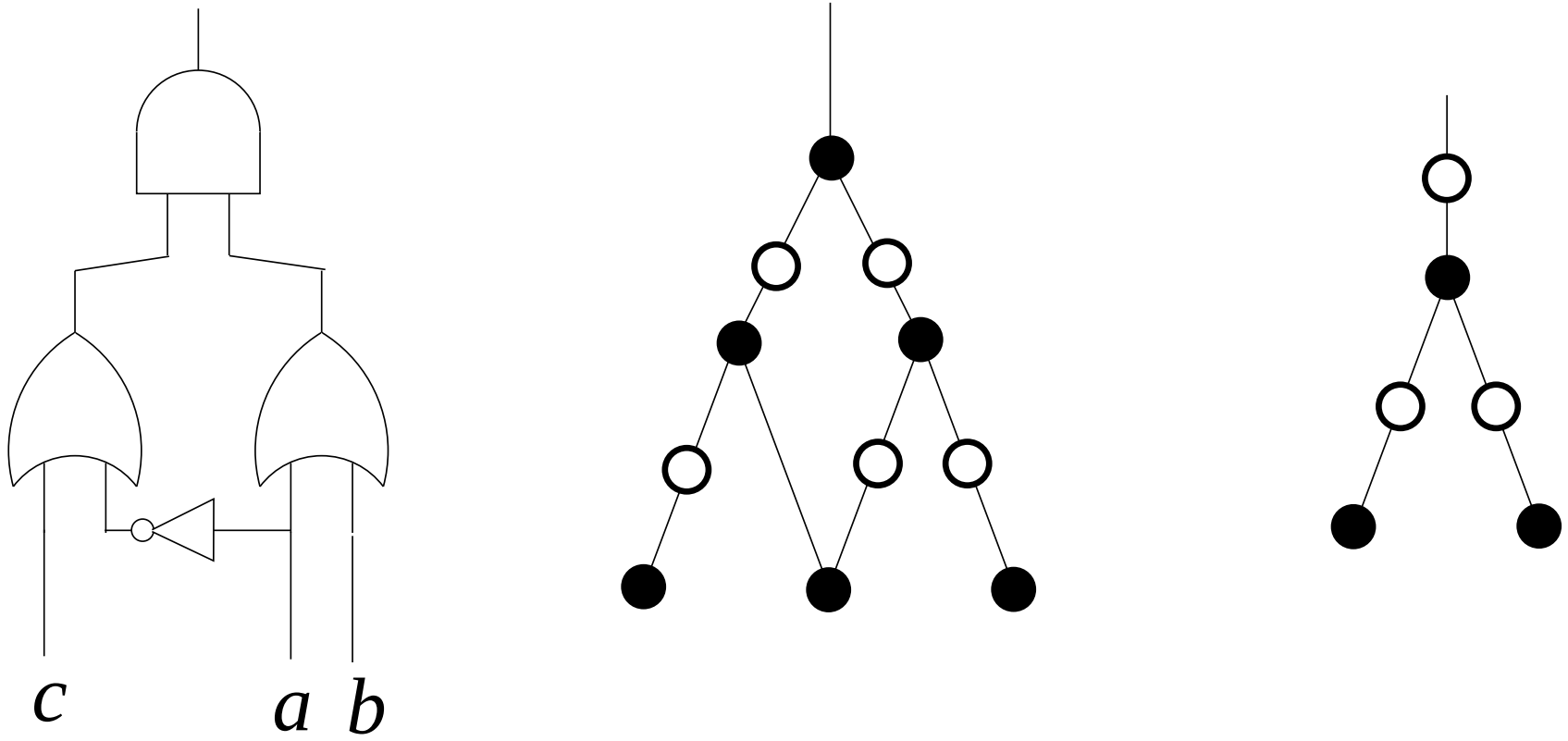
For the Boolean function representing either the output

$i - 1$ or $2n - i - 1$:

1. there is a circuit of size linear in n that implements it;
2. every BDD representing one of them has exponential size.

And Inverter Graphs

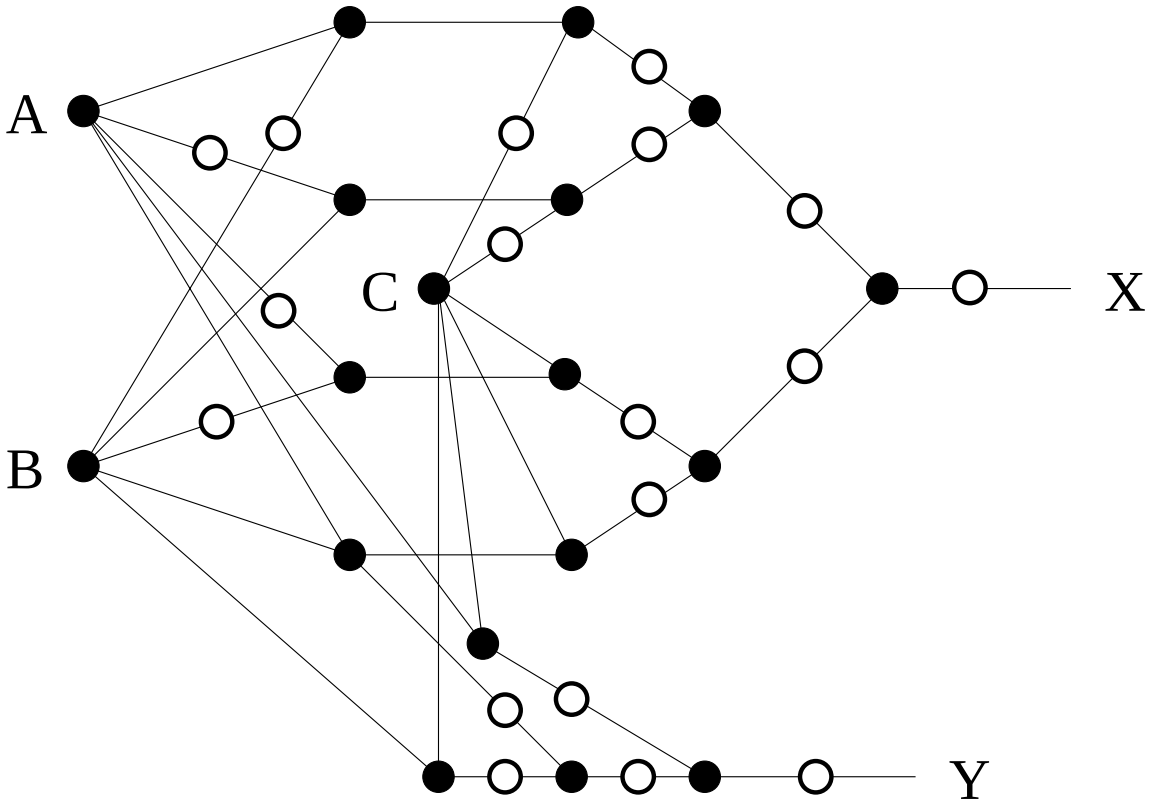
Can be smaller than the circuit they represent:



And Inverter Graphs

AIGs can represent specifications, for example 1-bit adder:

```
(equal x2 (or (and a2 (not b2) (not c1))
              (and (not a2) b2 (not c1))
              (and (not a2) (not b2) c1)
              (and a2 b2 c1)))
(equal c2 (or (and a2 b2) (and a2 c1) (and b2 c1)))
```



Equivalence Checking

Formally prove that representations of circuit designs or specifications exhibit exactly the same behavior (over time)

Three principle kinds of circuit tests:

Synchronous: show that two design specifications are functionally equivalent if they result in exactly the same sequence of output signals for any valid sequence of input signals.

Microprocessor: show that the functions specified for the instruction set architecture will result in exactly the same update of the content of memory that a given RTL implementation does.

System Design Flow: show that a Transaction Level Model matches its corresponding RTL

Equivalence Checking

Reminder:

Transaction Level Model:

High-level approach to modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture. Busses or FIFOs are modeled as channels, and are presented to modules using the SystemC language. Transaction requests take place by calling interface functions of these channel models, which encapsulate low-level details of the information exchange. At the transaction level, the emphasis is on the functionality of the data transfers - what data are transferred to what locations. Transfer protocol is less important. Allows experimentation with different bus architectures, etc. without having to recode models that interact with any of the buses.

Equivalence Checking

Reminder:

SystemC:

```
#include "systemc.h"

SC_MODULE(adder)            // module (class) declaration
{
  sc_in<int> a, b;          // ports
  sc_out<int> sum;

  void do_add()             // process
  {
    sum = a + b;
  }

  SC_CTOR(adder)            // constructor
  {
    SC_METHOD(do_add);      // register do_add to kernel
    sensitive << a << b;    // sensitivity list of do_add
  }
};
```

Equivalence Checking

Reminder:

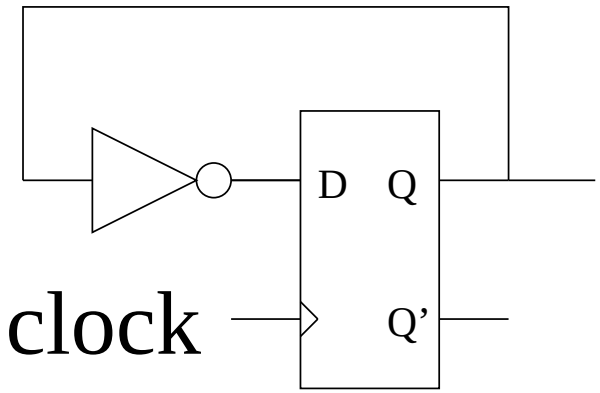
Register Transfer Level:

Describes the behavior of a synchronous digital circuit in terms of the flow of signals, or transfer of data, between hardware registers, and the logical operations performed on those signals.

RTL is used in hardware description languages like VHDL.

Example in VHDL for this circuit:

```
process(clk)
begin
   if rising_edge(clk) then
      Q <= not Q;
   end if;
end process;
```



Equivalence Checking

Reminder:

Synchronous circuits:

Step 1: Design at RTL using Verilog or VHDL

Step 2: Verify and/or validate the design

Step 3: Convert the design to a netlist using logic synthesis tool

Step 4: Optimize the design

Step 5: Add circuitry for testing

Step 6: The Problem - all that mucking may have changed functionality

Step 6: A Solution - test - but who says the tests are comple

Step 6: Better Solution - equivalence check!!

```
-- ========================================
-- full_adder_bl.vhd
--
-- full_adder Module - VHDL Gate Level Description
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
-- ========================================

ENTITY full_adder IS
  PORT (
     a  : IN BIT;
     b  : IN BIT;
     ci : IN BIT;
     s  : OUT BIT;
     co : OUT BIT
  );
END full_adder;
-- ========================================

ARCHITECTURE gate_level OF full_adder IS
  COMPONENT an02d1 PORT (a1, a2:     IN BIT; z: OUT BIT); END COMPONENT;
  COMPONENT xo02d1 PORT (a1, a2:     IN BIT; z: OUT BIT); END COMPONENT;
  COMPONENT or03d1 PORT (a1, a2, a3: IN BIT; z: OUT BIT); END COMPONENT;
  -- Intermediate nets
  SIGNAL net1,net2, net3, net4 : BIT;
BEGIN
  -- Sum Output Bit
  U1 : xo02d1 PORT MAP (a, b, net1);
  U2 : xo02d1 PORT MAP (ci, net1, s);
  -- Carry Output Bit
  U3 : an02d1 PORT MAP (a, b, net2);
  U4 : an02d1 PORT MAP (a, ci, net3);
  U5 : an02d1 PORT MAP (b, ci, net4);
  U6 : or03d1 PORT MAP (net2, net3, net4, co);
END gate_level;
```

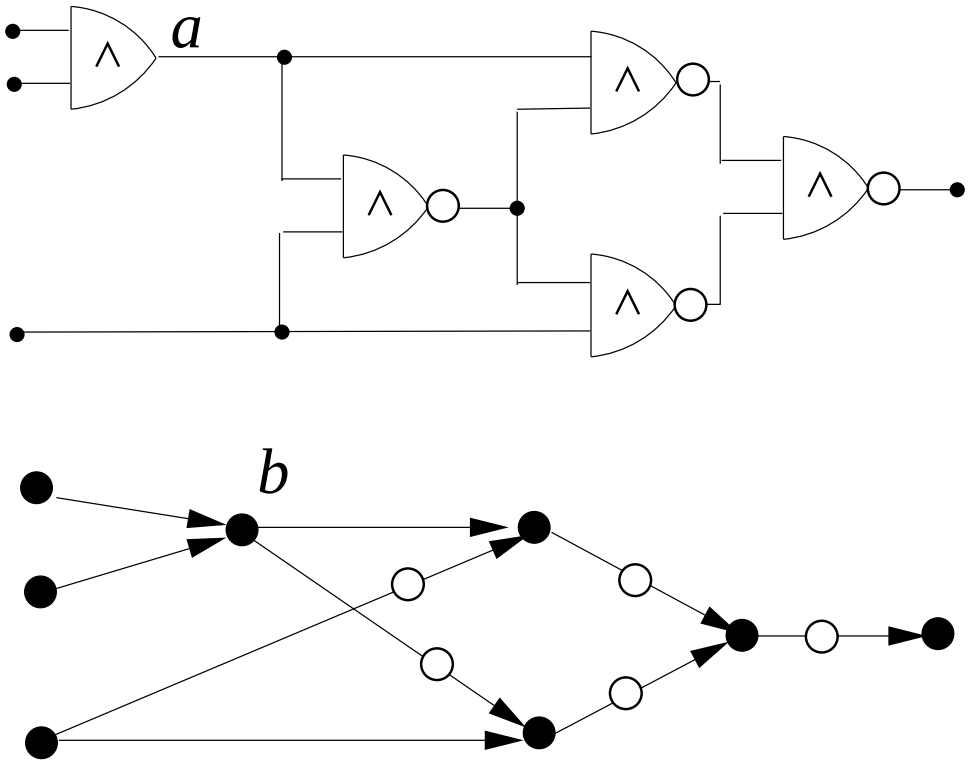
Netlist in ACL2

```
(defun f74181-netlist (c~ a0 a1 a2 a3 b0 b1 b2 b3 m s0 s1 s2 s3)
  (let* ((w0  (q-not m))
         (w1  (q-not b0))
         (w2  (q-not b1))
         (w3  (q-not b2))
         (w4  (q-not b3))
         (w5  a0)
         (w6  (q-and b0 s0))
         (w7  (q-and s1 w1))
         (w8  (q-and w1 (q-and s2 a0)))
         (w9  (q-and a0 (q-and s3 b0)))
         (w10 a1)
         (w11 (q-and b1 s0))
         (w12 (q-and s1 w2))
         (w13 (q-and w2 (q-and s2 a1)))
         (w14 (q-and a1 (q-and s3 b1)))
         (w15 a2)
     ...
         (w47 (q-not (q-and w26 (q-and w28 (q-and w30 w32)))))
         (w48 (q-not (q-and c~ (q-and w26 (q-and w28 (q-and w30 w32))))))
     ...
         (cout~ w62)
         (g~  w57))
    (list f0 f1 f2 f3 cout~ p~ g~ a=b)))
```

Netlist in ACL2 - a variant

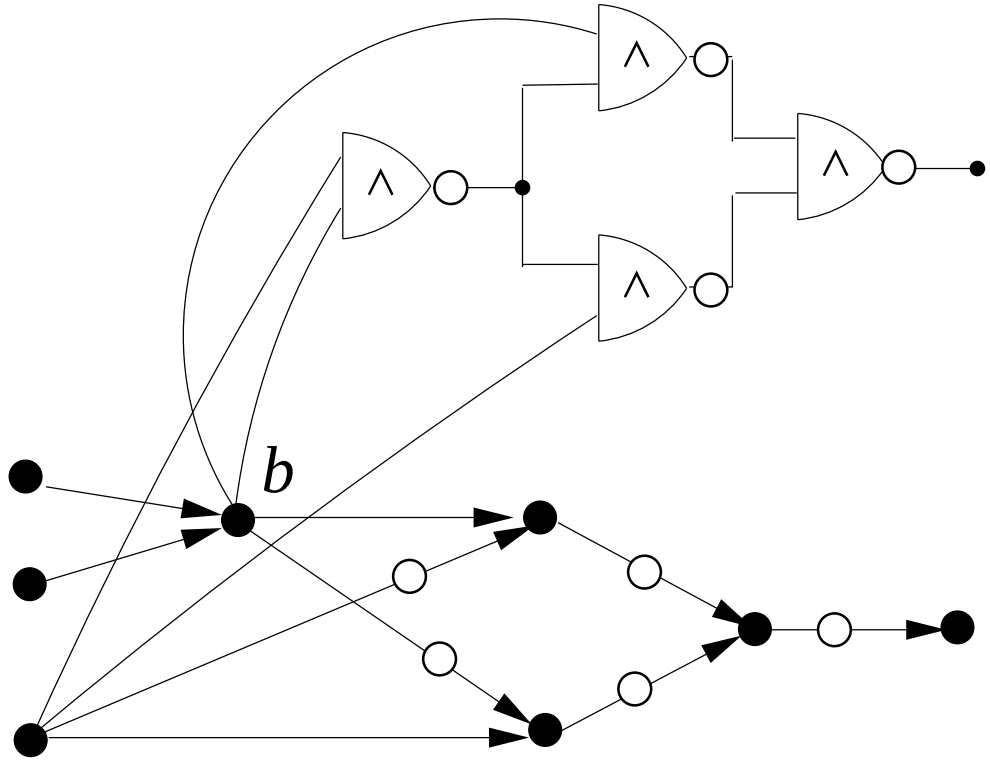
```
(defm *full-adder-module*
    `(:i (c a b) :o (sum co)
         :occs
         ((:u o0 :o (sum1 c1) :op ,*half-adder-module* :i (a b))
          (:u o1 :o (sum  c2) :op ,*half-adder-module* :i (sum1 c))
          (:u o2 :o (co)      :op ,*or2*       :i (c1 c2)))))

(defm *half-adder-module*
  `(:i (a b) :o (sum carry)
       :occs
       ((:u o0 :o (sum)   :op ,*xor2* :i (a b))
        (:u o1 :o (carry) :op ,*and2* :i (a b)))))

 ...
```

Equivalence Checking



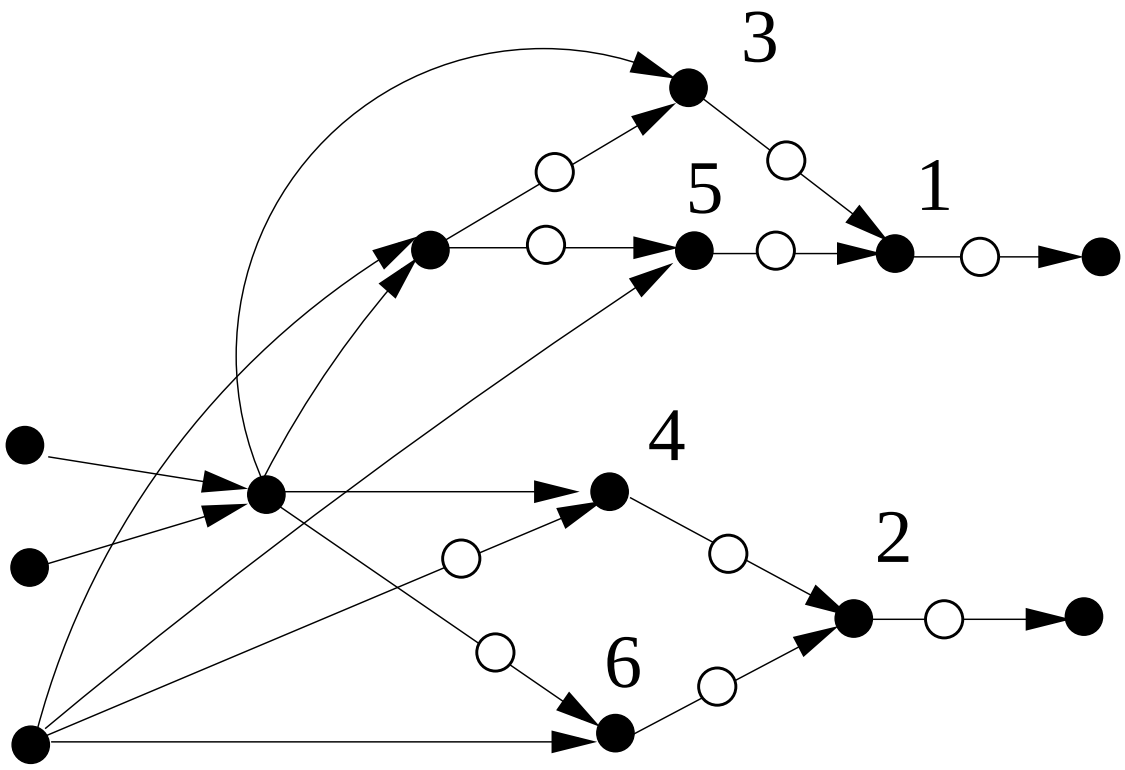
The beginning of the equivalency check - one circuit has been transformed to an AIG.

Equivalence Checking



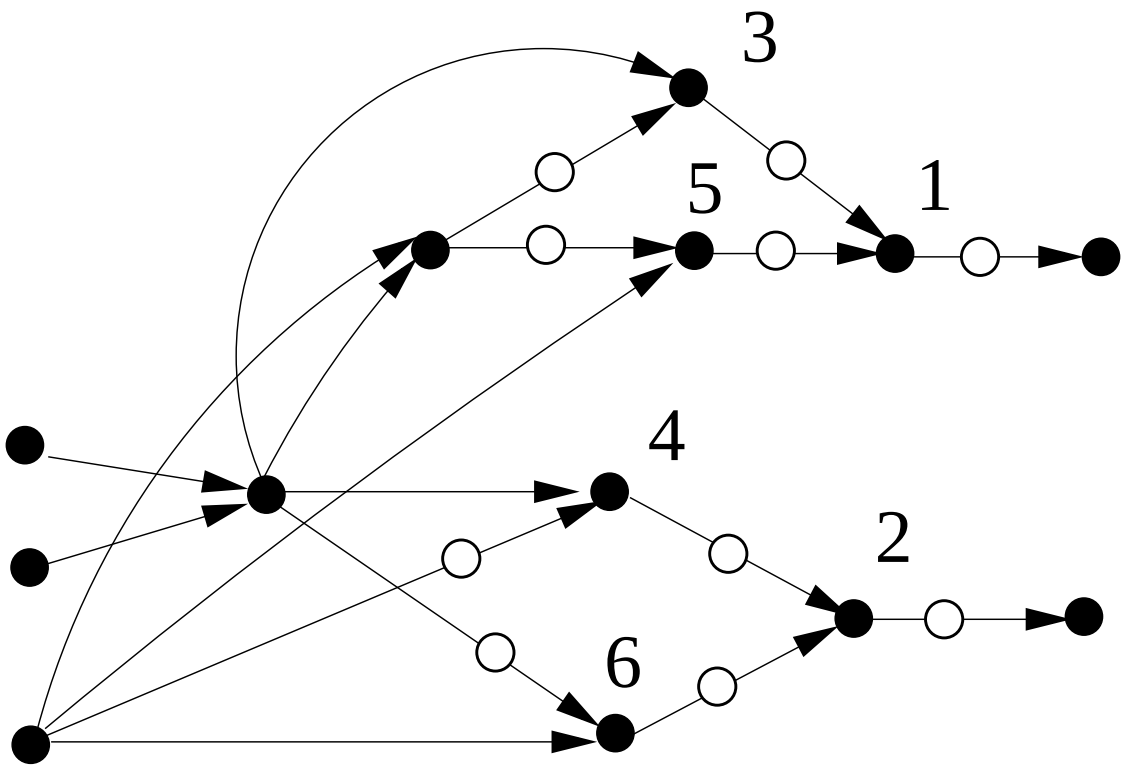
Vertex a of the circuit has been merged with vertex b of the AIG.

Equivalence Checking



The completed AIG.

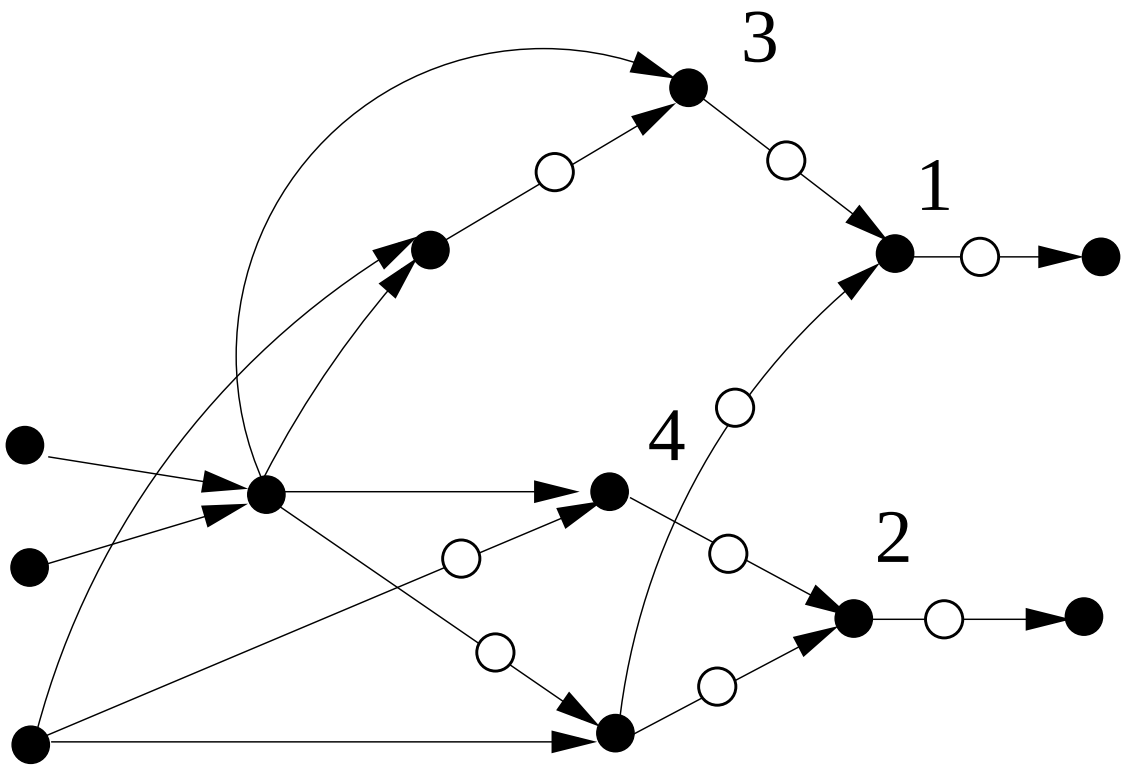
Equivalence Checking



inp/out	**vectors**							
top	0	0	0	0	1	1	1	1
middle	0	0	1	1	0	0	1	1
bottom	0	1	0	1	0	1	0	1
5	0	1	0	1	0	1	0	0
6	0	1	0	1	0	1	0	0

Check nodes 5 and 6.

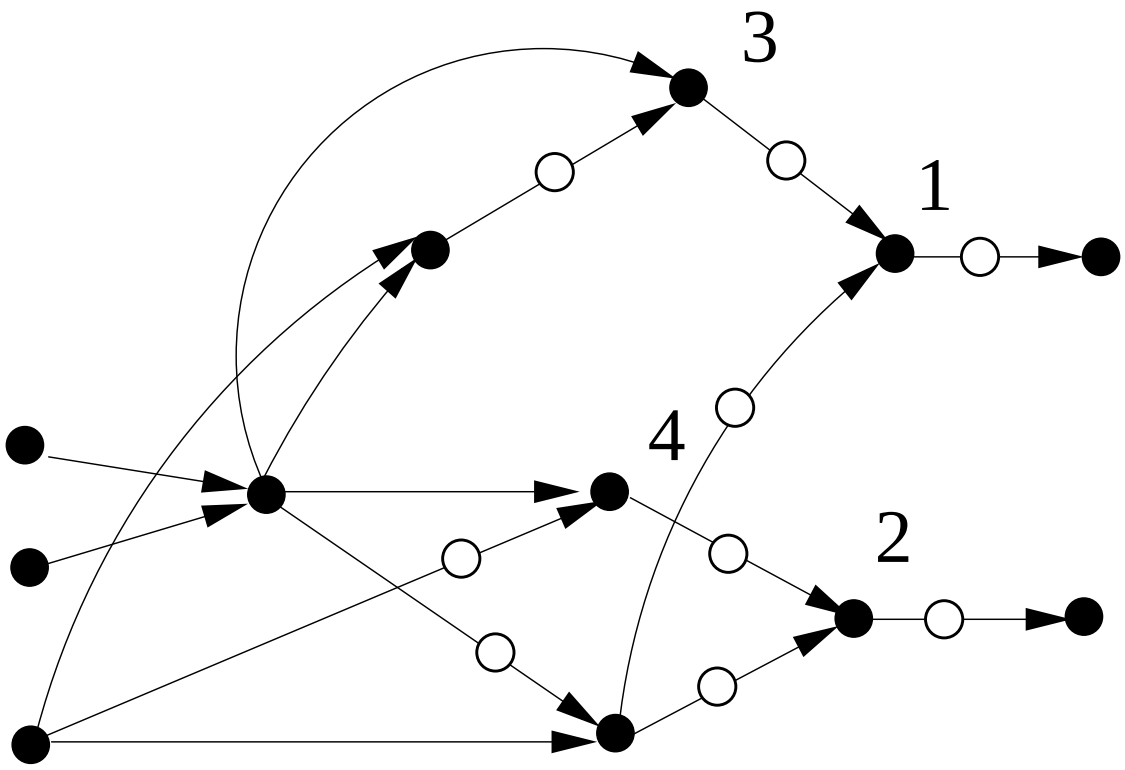
Equivalence Checking



inp/out	**vectors**							
top	0	0	0	0	1	1	1	1
middle	0	0	1	1	0	0	1	1
bottom	0	1	0	1	0	1	0	1
5	0	1	0	1	0	1	0	0
6	0	1	0	1	0	1	0	0

Reduce the graph.

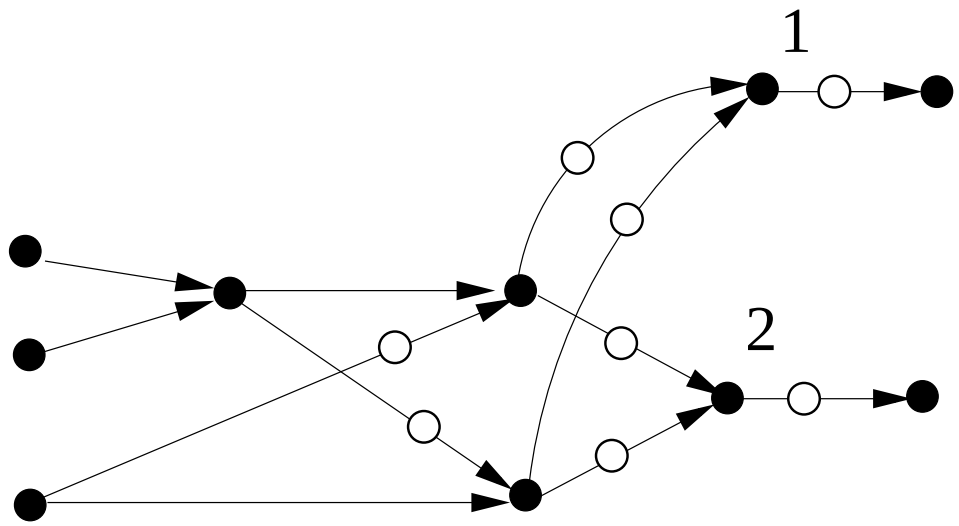
Equivalence Checking



inp/out	**vectors**							
top	0	0	0	0	1	1	1	1
middle	0	0	1	1	0	0	1	1
bottom	0	1	0	1	0	1	0	1
3	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	1	0

Check nodes 3 and 4.

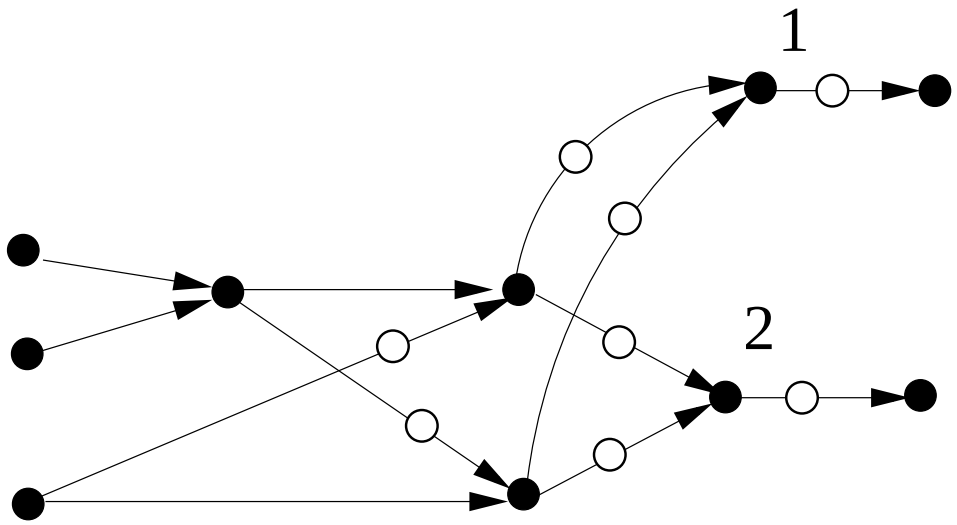
Equivalence Checking



inp/out	**vectors**							
top	0	0	0	0	1	1	1	1
middle	0	0	1	1	0	0	1	1
bottom	0	1	0	1	0	1	0	1
3	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	1	0

Reduce the graph.

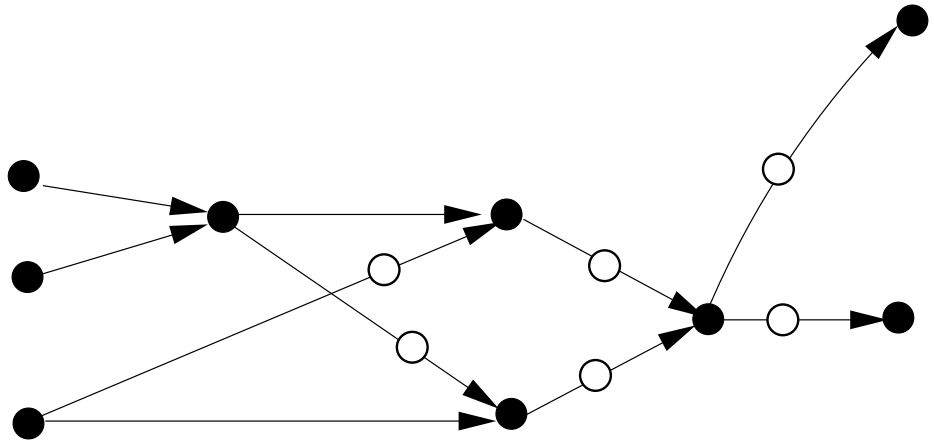
Equivalence Checking



inp/out	**vectors**							
top	0	0	0	0	1	1	1	1
middle	0	0	1	1	0	0	1	1
bottom	0	1	0	1	0	1	0	1
1	1	0	1	0	1	0	1	1
2	1	0	1	0	1	0	1	1

Check nodes 1 and 2.

Equivalence Checking



inp/out	**vectors**							
top	0	0	0	0	1	1	1	1
middle	0	0	1	1	0	0	1	1
bottom	0	1	0	1	0	1	0	1
1	1	0	1	0	1	0	1	1
2	1	0	1	0	1	0	1	1

Reduce the graph.