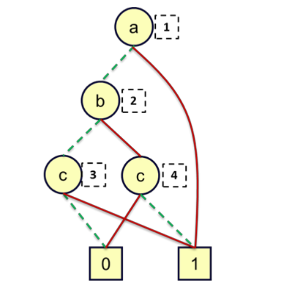
Problem Set #2

Warning: You have already made the maximum number of submissions. Additional submissions will not count for credit. You are welcome to try it as a learning exercise.

Question 1

Here is a small ROBDD for function $F(a, b, c)$. Which of these is the correct Boolean equation represented by this BDD?

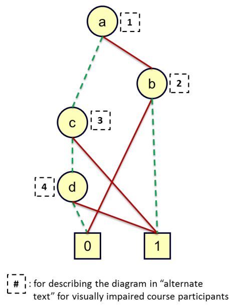


$\boxed{\#}$: for describing the diagram in “alternate text” for visually impaired course participants

- ○ $(b \oplus c)$
- ○ $(ab) \oplus c$
- ○ $a + (b \oplus c)$
- ○ a
- ○ $a + (b \overline{\oplus} c)$

Question 2

Here is another small ROBDD, for function $F(a, b, c, d)$. Which are true statements about this BDD?


a
1
b
2
c
3
d
4
0
1
#: for describing the diagram in "alternate text" for visually impaired course participants

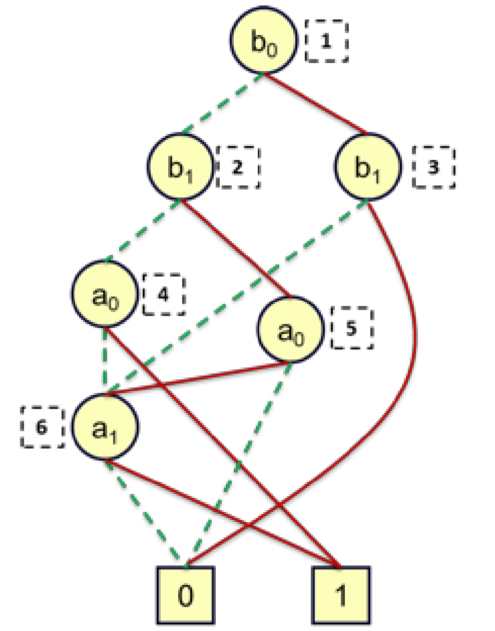
- ☐ There are 8 patterns (out of 16 total input patterns for) $abcd$ that will satisfy this BDD.
- ☐ There are 10 patterns (out of 16 total input patterns for) $abcd$ that will satisfy this BDD.
- ☐ There is no way to satisfy this BDD.
- ☐ Input pattern $abcd$ = 1101 will satisfy this BDD.
- ☐ Input pattern $abcd$ = 1010 will satisfy this BDD.

Question 3

A simple *comparator* takes two 2-bit unsigned binary numbers a_1a_0 and b_1b_0 and compares their magnitude, and sets the output $Z = 1$ just if a_1a_0 is **GREATER THAN** b_1b_0. (For example: $a_1a_0 = 11 > b_1b_0 = 01$, so $Z = 1$. But for $a_1a_0 = 01$ and $b_1b_0 = 01$, then $a_1a_0 == b_1b_0$, so $Z = 0$ for these inputs.)

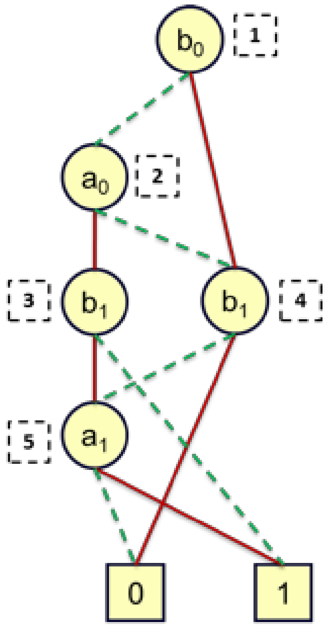
Which of these is a correct ROBDD for this function $Z(a_1, a_0, b_1, b_0)$ if we choose the variable ordering: $b_0 < a_0 < b_1 < a_1$?

○

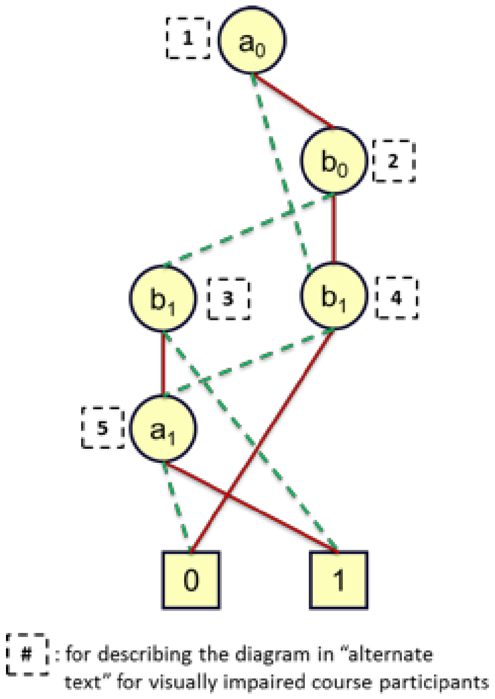


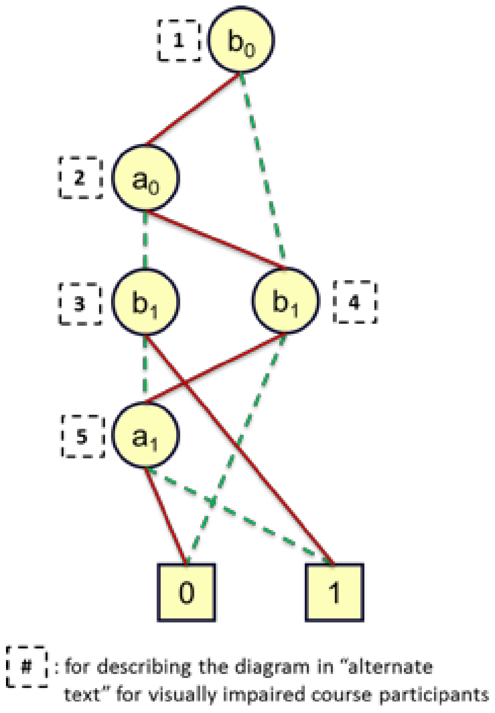
: for describing the diagram in "alternate text" for visually impaired course participants

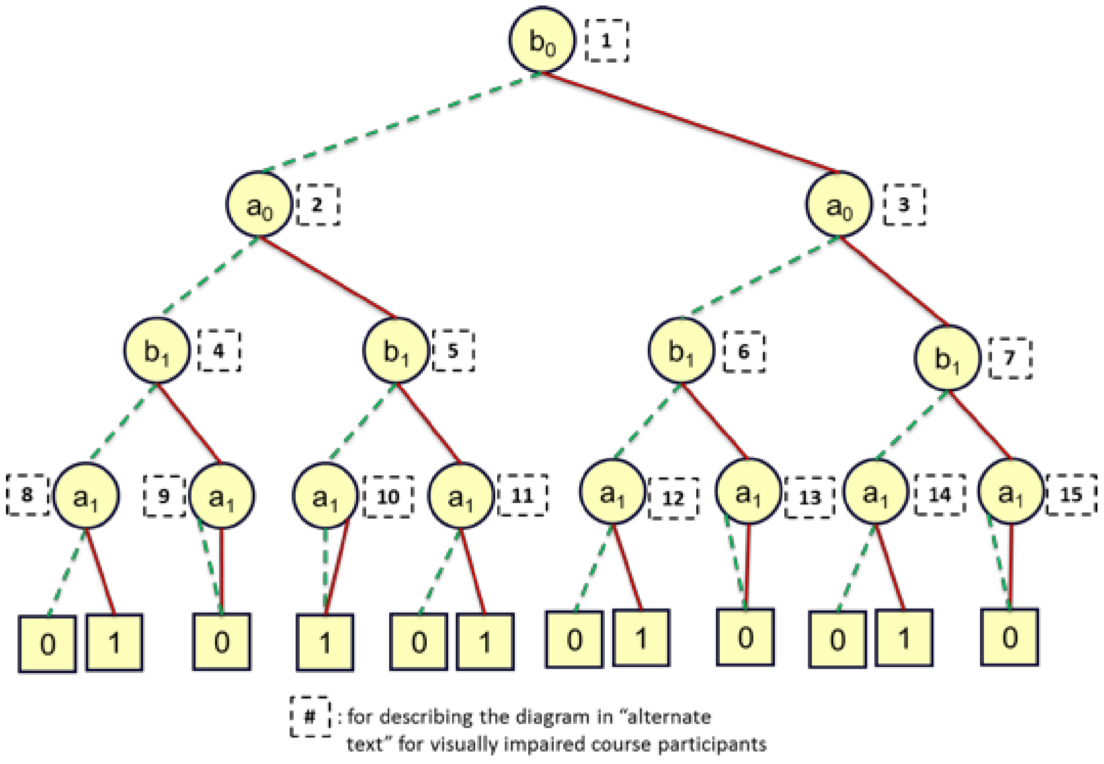
○



: for describing the diagram in "alternate text" for visually impaired course participants

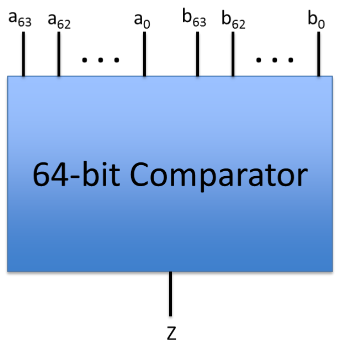

1
a_0
2
b_0
b_1
3
b_1
4
5
a_1
0
1
# : for describing the diagram in "alternate text" for visually impaired course participants


1
b_0
2
a_0
3
b_1
b_1
4
5
a_1
0
1
# : for describing the diagram in "alternate text" for visually impaired course participants

○ 

Question 4

Consider again the simple *comparator* circuit from Question 3. Recall the discussion from lecture about what sort of logic circuits have good variable orderings. Note: a comparator is essentially a *subtractor*, and the Z signal is closely related to the high-order carry bit (actually, it is technically a borrow bit, but same ideas.) Suppose instead of 2-bit numbers, we now wanted to compare large, 64-bit numbers.



Which of these is true for this new 64-bit comparator? Select all correct answers:

☐ Because this is an arithmetic circuit that looks like a single carry chain, we expect a variable order such as $a_{63} < b_{63} < a_{62} < b_{62} < a_{61} < b_{61} < \ldots a_0 < b_0$ will lead to a BDD with manageable size.

☐ Because this is an arithmetic circuit that looks like a single carry chain, we expect a variable order in which we first list all the a_i variables, and then list all the b_i variables, will lead to a BDD with manageable size.

☐ Because this is not an arithmetic circuit, we cannot really guess whether there is a good variable order or not.

☐ Because this is an arithmetic circuit that looks like a single carry chain, we expect a variable order such as $b_0 < b_1 < b_2 < \ldots < b_{63} < a_0 < a_1 < a_2 < \ldots < a_{63}$ will lead to a BDD with manageable size.

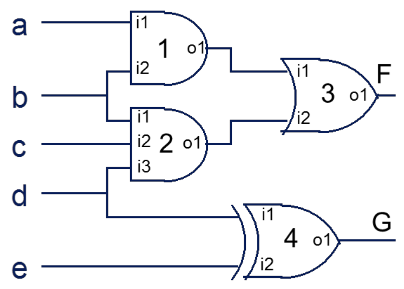
☐ Because this is an arithmetic circuit that looks like a single carry chain, we expect a variable order that alternates the a_i and b_i variables will lead to a BDD with manageable size.

Question 5

There are many good heuristics that try to find a good variable order for a BDD. In this problem, we will explore one simple such heuristic, called Dynamic Weight Assignment, from:

Shin-ichi Minato, Nagisa Ishiura and Shuzo Yajima, “Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation,” Proceedings ACM/IEEE Design Automation Conference, 1990, pp 52-57.

Here is a small logic network to illustrate the idea. The heuristic works like this:



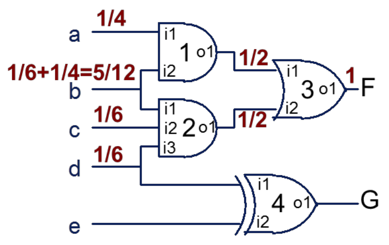
Primary inputs: a,b,c,d,e
Primary outputs: F,G

- Pick a primary output; put a weight “1” there. If there is more than 1 output, start with the one that has the deepest logic depth from the inputs.
- For each gate with weights on its output but not its input, “push” the weight back through the gate to each of the gate’s inputs, dividing the weight at the output by the number of input wires to the gate. Each input gets this new, computed equal weight.
- If there is fanout (one wire goes to >= 2 inputs) then ADD the weights to get the new

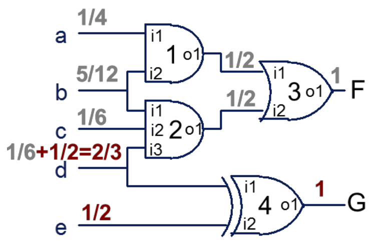
weight for this wire.

- Continue until all network's primary inputs are labeled.
- If there are more primary outputs, pick the one with next deepest logic depth and repeat.

For our example, this is what would happen:



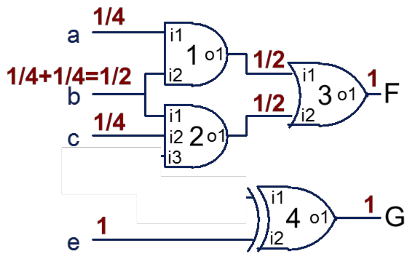
Step 1: Propagate weight back from output **F** to inputs



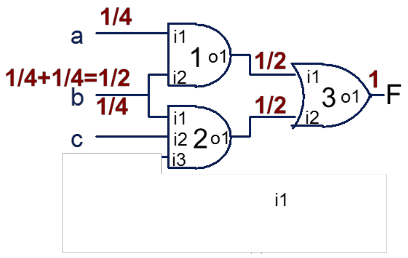
Step 2: Propagate weight back from output **G** to inputs.
Choose input **d** next in BDD order

- To choose the next variable to appear in the global order, do this: Select the input with the largest weight on it. If there is a tie, choose the input that appears first in alphabetical order; e.g., if **a** and **d** are tied, pick **a**. This becomes the first variable in your BDD ordering.
- Erase the wire with this variable from the diagram – you will not consider it further. If you erase every input wire from any gate, erase the entire gate, and its output wire.
- Continue the weight assignment procedure described above: start with a weight of 1 on the output of deepest logic depth, and push the weights backward toward the inputs.
- The input with the largest weight becomes the second variable in your BDD order; erase this wire, and continue the procedure.

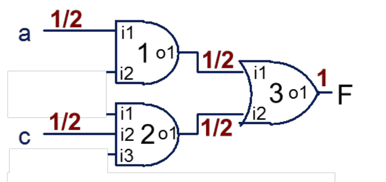
If we apply this procedure, we will select input **d** as our first in the BDD order, and erase the **d** wire to yield a new logic network, to which we can continue to apply these steps. This is how the method will work as we progress:



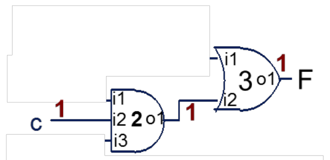
Step 3: Propagate weight back from output **F** to inputs.
Step 4: Repeat for output **G**.
Choose input **e** as next for BDD order.



Step 5: Propagate weight back from output **F** to inputs.
Choose input **b** as next for BDD order.



Step 6: Propagate weight back from output **F** to inputs.
Tie: Choose **a** as next for BDD order.

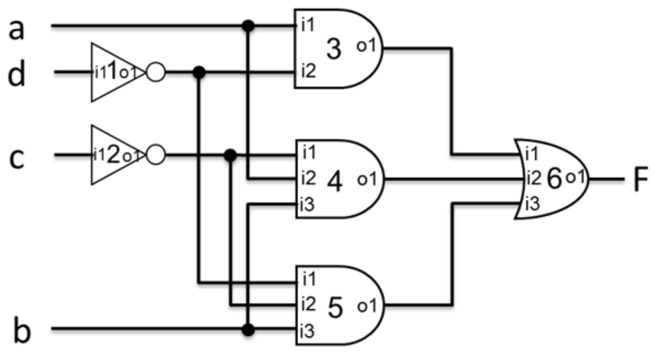


Step 7: Propagate weight back from output **F** to inputs.
Choose input **c** as next for BDD order.

The result of this heuristic yields this variable order: $d < e < b < a < c$.

The idea is that deeper logic outputs are more important for the ordering, and the gates with smaller number of inputs, and wires with fanout, are also more important (in their ability to control/influence the output). The heuristic is simple, but effective.

Do this: for the small logic circuit below, apply this ordering heuristic, and tell us **the first two variables** in the computed BDD ordering. You can compute the full ordering of a,b,c,d if you like – but after two passes of the heuristic, you will see most of the interesting parts.



- Variable order: $c < d$
- Variable order: $d < a$
- Variable order: $c < a$
- Variable order: $a < b$
- Variable order: $b < c$
- Variable order: $d < b$
- Variable order: $b < d$
- Variable order: $d < c$
- Variable order: b < a
- Variable order: a < d
- Variable order: c < b
- Variable order: a < c

Question 6

Consider this function f expressed in CNF clause form:

f=\underbrace{(a_1+a_2)}_{\omega_1}

\underbrace{(a_1+a_2+a_3+\bar{a_4})}_{\omega_2}

\underbrace{(\bar{a_2}+a_4)}_{\omega_3}

\underbrace{(a_1+\bar{a_2}+\bar{a_4})}_{\omega_4}

\underbrace{(\bar{a_1}+a_2+a_4)}_{\omega_5}

\underbrace{(\bar{a_1}+a_2+a_3)}_{\omega_6}

\underbrace{(\bar{a_1}+\bar{a_2}+\bar{a_4}+\bar{a_5})}_{\omega_7}

\underbrace{(\bar{a_1}+a_2+\bar{a_3}+\bar{a_4}+\bar{a_5})}_{\omega_8}

Let us **decide** to set variable a_4 to 0. Make this assignment, then run Boolean Constraint Propagation (BCP) by hand on this CNF. Continue BCP until you cannot make any additional progress on the formula. Which of the following are true statements about this result after BCP? Select each correct statement about actions that occur doing this BCP process:

☐ The result after BCP is that the CNF is unresolved, and we need to make further variable assignments to continue progress.

☐ Clause \omega_5 becomes unit and implies that a_1=0.

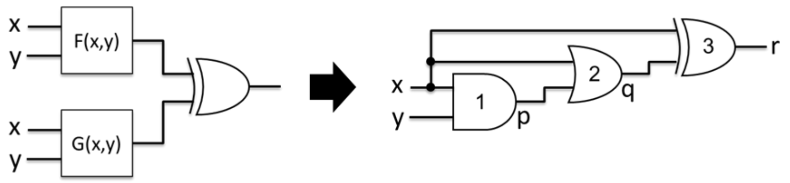
☐ Clause \omega_3 becomes unit and implies that a_2=0.

☐ The result after BCP is that the CNF has a conflict, and cannot be satisfied without undoing an earlier assignment.

☐ Clause \omega_1 becomes unit and implies that a_1=1.

Question 7

We showed in lecture how to use SAT to check if two different gate-level implementations F, G of a logic function are correctly implementing the function. We take each implementation, we connect the (assume one single) output of each block to an EXOR gate, and we run SAT on this new circuit. If the F, G circuits are *different*, the combined circuit with the EXOR will be **satisfiable**, and the SAT engine can return an input pattern that will make F\neq G (since this is the only way to make the EXOR output a 1). If F,G are the *same*, then the combined circuit will be **unsatisfiable**. Let's try this with a tiny example. Suppose F(x,y) = x. That is, we just connect the x input to the F output. And suppose G(x,y) = x+xy, which we can implement with one AND and one OR gate. The diagram below shows the EXOR construction, and the completed circuit at the gate-level.



We have labeled the internal nodes to make things easier to analyze. (Now, a little Boolean algebra shows the right answer to this question with very little work; but the point it to develop the ideas for the SAT mechanical construction for logic network comparison).

Do this: write the CNF formula that is satisfiable if and only if the gate-level circuit above (inputs x,y; intermediate wires p,q; output r) is satisfiable. Convert the formula to CNF form using the consistency functions discussed in class.

Hint: You should get 11 clauses, one of them unit. Also, we'll go ahead and tell you that the consistency function for the XOR gate is (r+\bar{q}+x)(r+q+\bar{x})(\bar{r}+q+x)(\bar{r}+\bar{q}+\bar{x})

Answer these: which of the following clauses appear in your SAT CNF formula? Select only the clauses that should appear in this CNF.

- ☐ (\bar{p}+x)
- ☐ (r)
- ☐ (\bar{p}+x+y)
- ☐ (p+\bar{x}+\bar{y})
- ☐ (q+\bar{p})
- ☐ (\bar{q}+x+p)
- ☐ (\bar{p}+y)
- ☐ (\bar{r})

- ☐ (q+\bar{x}+\bar{p})
- ☐ (q+\bar{x})

Question 8

BDDs operating on general Boolean logic, and SAT solvers operating on CNF clause lists, are both techniques we can use to work with complex logic. But they each have different capabilities. Which of these are true statements about each respective technology? Select the correct statements in this list.

- ☐ SAT solvers and BDD packages will always work for any Boolean function.
- ☐ We can use SAT to find a satisfying assignment of a Boolean equation.
- ☐ We can use SAT to find ALL satisfying assignments of a Boolean equation.
- ☐ A BDD package allows us to perform many arbitrary manipulations on a set of Boolean equations: AND, OR, NOT, EXOR, EXNOR, QUANTIFY, etc.
- ☐ Suppose we have two gate level hardware implementations of some function, call them F and G. We want to check if the hardware for F produces identical outputs as the hardware for G. We can do this using BDDs, but we cannot do this using a SAT solver.
- ☐ A SAT solver allows us to perform many arbitrary manipulations on a set of Boolean equations: AND, OR, NOT, EXOR, EXNOR, QUANTIFY, etc.
- ☐ We can use a BDD to find a satisfying assignment of a Boolean equation.
- ☐ Suppose we have two gate level hardware implementations of some function, call them F and G. We want to check if the hardware for F produces identical outputs as the hardware for G. We can do this using a SAT solver, but we cannot do this using BDDs.
- ☐ SAT solvers and BDD packages may not always work for any Boolean function.
- ☐ We can use a BDD to find ALL satisfying assignments of a Boolean.

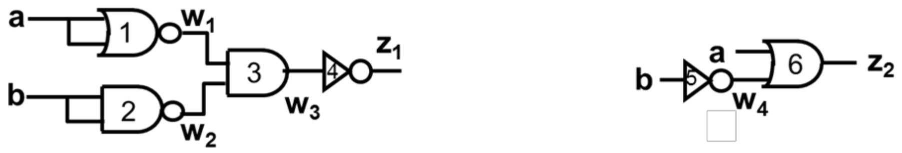
Question 9

Equivalence Checking with BDDs.

We can use a BDD to determine if two gate-level networks are implementing the same function or not. And, if they are not the same, we can use the BDD to find an assignment of the variables that will make the network outputs different. In this problem, we will let you play with a real BDD package: **kbdd** from Prof. Randy Bryant's research group at Carnegie Mellon University (http://www.cs.cmu.edu/~bryant/)

To explain how to use **kbdd**, let us compare these two very simple logic networks:

Equivalent to??



If you go look in the TOOLS section of our course website, you will find two helpful tutorials about kbdd: a written document that explains the tool and its capabilities, and also a short video tutorial about to use kbdd in our Coursera MOOC environment. Please go look at those before you proceed further in this problem.

Assuming you have read/viewed these tutorial materials, the kbdd script for checking equivalence of z1 and z2 is shown below. Note that we have embedded some comments in the script to help you read it more easily.

```
=================================================================================
# input vars
boolean a b
#
# define left-hand circuit z1
eval w1 !(a+a)
eval w2 !(b&b)
eval w3 (w1&w2)
eval z1 !w3
#
# define the right-hand circuit z2
eval w4 !b
eval z2 (a+w4)
# verify if two functions are equivalent
verify z1 z2
# XOR z1 and z2 to find assignments of the variables
# that will make the network outputs different
eval diff z1 ^ z2
satisfy diff
```

# that's it

quit

===

You will see two results.

(1)

“0 verify failed”

It means z1 and z2 are NOT equivalent.

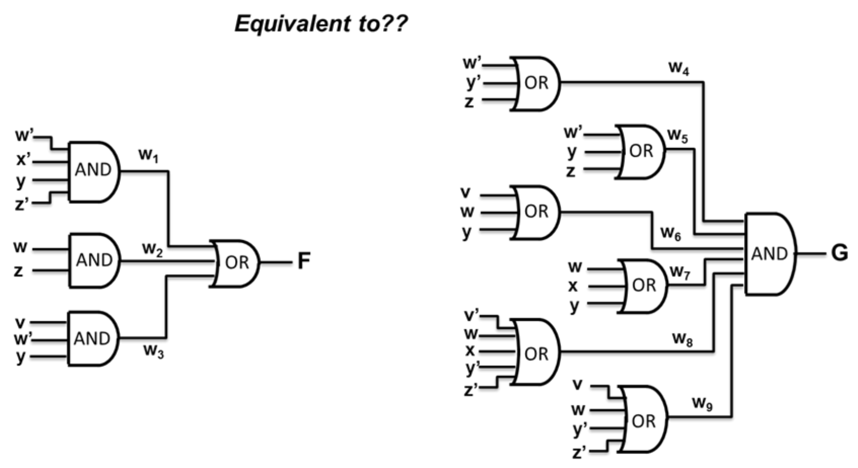
(2)

“Variables: a

0”

It means inputs <a=0, b=1> and <a=0, b=0> will make two logic networks have different values.

Now it’s your turn! Please compare the two logic networks F and G, each functions of 5 variables (v,w,x,y,z) shown below using **kbdd**. Then, look at the tool’s outputs and answer these questions: which are the true statements?



☐ Assignment (v, w, x, y, z) = (00110) can tell F and G are not equivalent.

☐ Assignment (v, w, x, y, z) = (10011) can tell F and G are not equivalent.

☐ F and G are equivalent.

☐ Assignment (v, w, x, y, z) = (10100) can tell F and G are not equivalent.

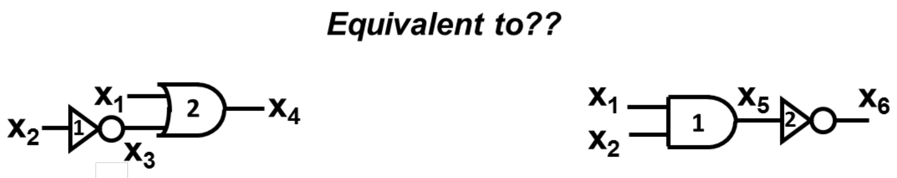
☐ Assignment (v, w, x, y, z) = (10110) can tell F and G are not equivalent.

Question 10

Equivalence Checking with SAT.

We can also use a SAT solver to determine if two gate-level networks are implementing the same function or not. And, if they are not the same, the SAT solver can find an assignment of the variables that will make the network outputs different. In this problem, we will let you play with a real SAT solver package: **MiniSat**, from Niklas Eén, Niklas Sörensson (http://minisat.se).

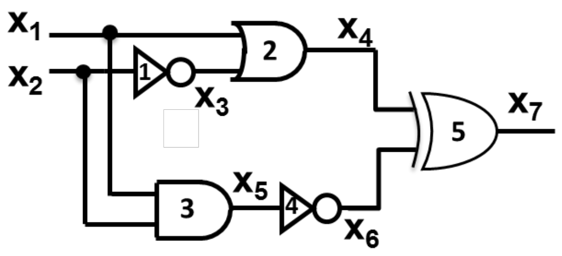
To explain how to use MiniSat, let us compare these two very simple logic networks. (For your convenience, we label each signal, both primary inputs/outputs, and internal wires, with the name “x” and a subscript i; this will help you easily connect them to the corresponding numerical variable i in MiniSat)



If you go look in the TOOLS section of our course website, you will find two helpful tutorials about **MiniSat**: a written document that explains the tool and its capabilities, and also a short video tutorial about to use MiniSat in our Coursera MOOC environment. Please go look at those before you proceed further in this problem.

Assuming you have read/viewed these tutorial materials, the **MiniSat** input file for checking equivalence of x_4 and x_6 is developed below.

First, we connect the two circuits’ outputs with an EXOR gate; we will use **MiniSat** to check if this new logic network can be satisfied (x_7=1) or not. If it is *satisfiable*, we will know the original two circuits are not equivalent, because there exists one satisfying assignment such that <x_4 = 1, x_6=0> or <x_4=0, x_6=1>.



Here are SAT clauses converted from this little logic network:

(x_2+x_3)(\neg x_2+\neg x_3)(\neg x_1+\neg x_2+x_5) (x_1+\neg x_5) (x_2+ \neg x_5)

(x_1+x_3+\neg x_4) (\neg x_1+x_4) (\neg x_3+x_4)\cdot (x_5+x_6)(\neg x_5+\neg x_6)

(x_4+\neg x_6+x_7) (x_4+x_6+\neg x_7) (\neg x_4+x_6+x_7) (\neg x_4+\neg x_6+\neg x_7)

(x_7)

The input file for the MiniSat solver is as follows.

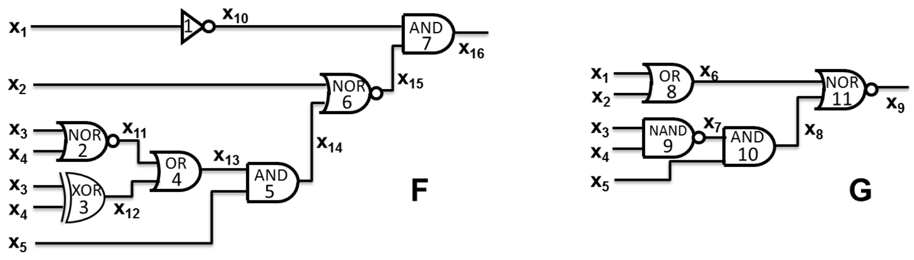
```
==========================================================================
c There are 7 variables and 15 clauses
p cnf 7 15
2 3 0
-2 -3 0
-1 -2 5 0
1 -5 0
2 -5 0
1 3 -4 0
-1 4 0
-3 4 0
5 6 0
-5 -6 0
4 -6 7 0
4 6 -7 0
-4 6 7 0
-4 -6 -7 0
7 0
==========================================================================
```

You will see two sorts of results: (1) some “internal information” about what actions **MiniSat** took during its DPLL search (see our tutorial document for a discussion of these); (2) a result about the *satisfiability* of the circuit, and maybe a solution which is a satisfying assignment. In this case, the second part produces this:

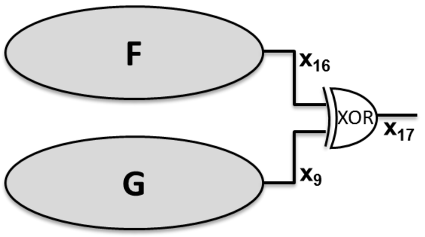
“SAT
1 2 -3 4 5 -6 7 0”

It means x_1=1 and x_2=1 will lead to x_7=1.

Now it's your turn! Please use **MiniSat** to compare these two logic networks F and G, which are each functions of 5 variables x_1, x_2, x_3, x_4, and x_5.



You need to EXOR the outputs of F and G to see if the EXOR gate can be satisfied or not. So, this is the network to submit to **MiniSat**:



X_{17} = 1 Satisfied ?

Since transforming gate networks into SAT clauses is very tedious (and, indeed, we write computer programs to do this!) we are giving you most of the partial input file for this problem, to make things easier. Please complete this file and run **MiniSat** to check if x_{17} can be satisfied. To help ensure you get the right answer, we will also tell you here that the total number of clauses should be 38.

==

c There are 17 variables and 38 clauses

c The clauses for F

p cnf 17 38

1 10 0

-1 -10 0

3 4 11 0

-3 -11 0

-4 -11 0

3 -4 12 0

3 4 -12 0

-3 4 12 0

-3 -4 -12 0

11 12 -13 0

-11 13 0

-12 13 0

-5 -13 14 0

5 -14 0

13 -14 0

2 14 15 0

-2 -15 0

-14 -15 0

-10 -15 16 0

10 -16 0

15 -16 0

c Please fill out the clauses for G

…

…

…

c The clauses for comparing XOR gate

16 -9 17 0

16 9 -17 0

-16 9 17 0

-16 -9 -17 0

17 0

==

Based on your results from **MiniSat**, which of the following are the true statements?

☐ Assignment (x_1, x_2, x_3, x_4, x_5) = (00110) can tell F and G are not equivalent.

☐ Assignment (x_1, x_2, x_3, x_4, x_5) = (00101) can tell F and G are not equivalent.

☐ F and G are equivalent.

☐ Assignment (x_1, x_2, x_3, x_4, x_5) = (00100) can tell F and G are not equivalent.

☐ Assignment (x_1, x_2, x_3, x_4, x_5) = (00111) can tell F and G are not equivalent.

☐ In accordance with the Honor Code, I certify that my answers here are my own work.

Submit Answers Save Answers