Exercise 3

Symbolic Model Checking

1) Give the initial predicate and the transition predicate for the following Verilog module:

```
module simple(clock, flip1, flip2, var1, var2);
   input clock, flip1, flip2;
   output var1, var2;
   reg      var1, var2;

   initial begin
      var1 = 0;
      var2 = 0;
   end

   always @(posedge clock) begin
      if (flip1) begin
          var1 = ~var1;
      end
      if (flip2) begin
          var2 = ~var2;
      end
   end
endmodule
```

Solution 1)
Initial predicate (refers only to register variables: `var1,var2`)
`var1=0 /\ var2=0`

Transition predicate (refers only to primed and unprimed register variables: `var1,var1’,var2,var2’`)

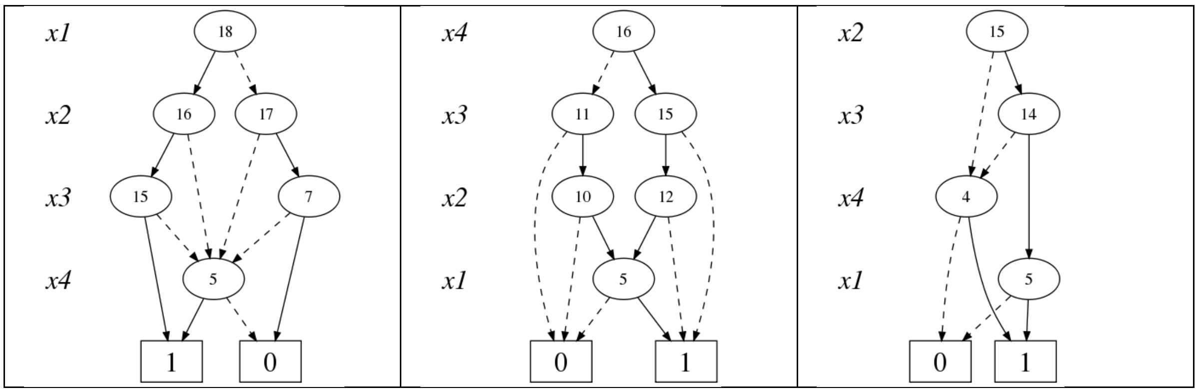
```
∃ flip1, flip2.
  ( flip1 -> (var1’<->!var1)
/\(!flip1 -> (var1’<-> var1)
/\( flip2 -> (var2’<->!var2)
/\(!flip2 -> (var2’<-> var2)
=((var1’<-> var1) /\ (var2’<-> var2)) \/
 ((var1’<->!var1) /\ (var2’<-> var2)) \/
 ((var1’<-> var1) /\ (var2’<->!var2)) \/
 ((var1’<->!var1) /\ (var2’<->!var2))
```

Note that `∃a,b.f(a,b,c) = f(false,false,c) \/ f(false,true,c) \/ f(true,false,c) \/ f(true,true,c).`

2) Consider the Boolean expression

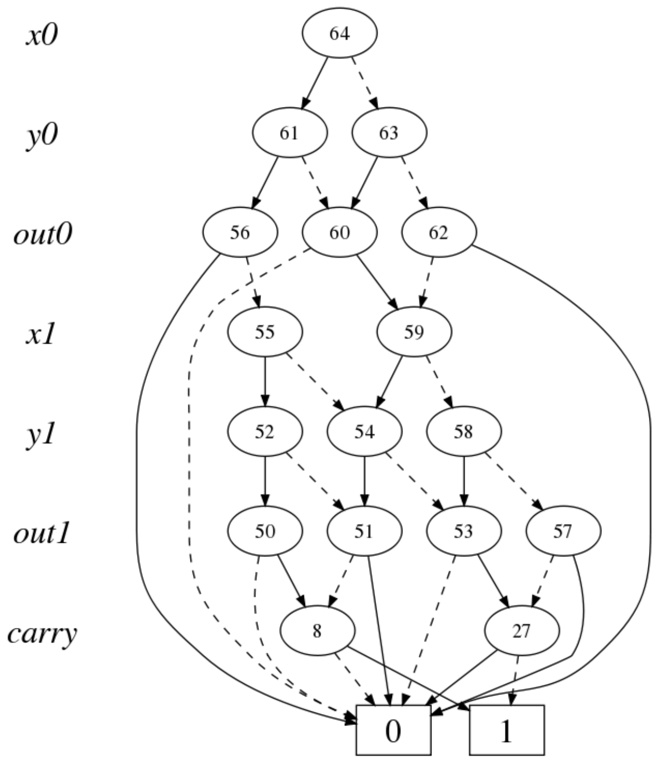
$(x_1 \wedge x_2 \wedge x_3) \vee (\neg x_2 \wedge x_4) \vee (\neg x_3 \wedge x_4)$

Choose a variable ordering for the variables x_1, x_2, x_3, x_4, and draw the resulting BDD. Can you reduce the size of the BDD by reordering the variables?



Intuitively, a good variable order puts variables that appear in the same clause close to each other because the truth-value of the clause depends on all of them.
Furthermore, it is good to decide variables that appear in a lot of clauses first because they have "more" influence than others since more clauses depend on their values. Another rule of thumb is to first decide on variables that appear in small clauses, e.g., which only two literals.
In the example, first note that x2 and x3 are completely symmetric and that x1 appears only in one clause. So we need to choose between {x2,x3} < x4 < x1 or x4 < {x2,x3} < x1, where {x2,x3} means the order of x2 and x3 does not matter. You can see from the two pictures above on the right, that {x2,x3}<x4<x1 is the better one.

3) Let X be the set $\{x_0, x_1, y_0, y_1, out_0, out_1, carry\}$. Choose an appropriate ordering of the variables, and construct the BDD for the requirement that the output out_1out_0, together with the carry bit carry, is the sum of the inputs x_1x_0 and y_1y_0. Is your choice of ordering optimal?



About the variables order, see rules-of-thumb in the previous example.

4) Given a BDD B over a set X of variables, a variable x, given an algorithm to construct a BDD representing the function ∃x.B, so x is existentially quantified over B. What is the running time of your algorithm in terms of the number of vertices of the input B.

Bonus: Instead of pseudo-code, extend miniBDD http://www.cprover.org/miniBDD/ with a function with the signature: `BDD exists(unsigned) const;`
that when applied to BDD B with the variable id x returns a BDD that corresponds to ∃x.B.

```
//source code by Iris Safaka and Dumitru Ceara
BDD restrict_bdd(const BDD &x, unsigned label, bool value)
{
  assert(x.node!=NULL);

  miniBDD_mgr *mgr=x.node->mgr;
  BDD u;


  if (x.is_constant()) {
    cerr << "constant " + x.var() << endl;
    u = BDD(x);
  }  else if (x.node->var == label) {
    cerr << "x found " + x.var() << endl;
    value ? u = BDD(x.high()) : u = BDD(x.low());
  } else {
    u = mgr->mk(x.var(),
          restrict_bdd(x.low(), label, value),
          restrict_bdd(x.high(), label, value));
  }

  return u;
}

BDD exists(const BDD &x, unsigned label)
{
  return apply(or_fkt,
            restrict_bdd(x, label, true),
            restrict_bdd(x, label, false));
}
```

For Running time: `restrict_bdd` needs to traversing the BDD by looking at each node, so if the BDD has n nodes, the running time of restrict_bdd is O(n).
`exists` first calls `restrict_bdd` twice (O(n) + O(n) = O(n)) and then it applies the `or_fct` between two BDDs of size n. This can take n^2 time, since in the worst case we have to consider every pair of BDD nodes (see example on the next page). So, we get O(n) + O(n^2) = O(n^2).

Example that shows how expansive exists/apply can be:

Consider the BDD starting at node 14. It represents the function
$f(x0,x1,x2,x3,x4)=(!x0 \wedge (x1 \oplus x3)) \vee (x0 \wedge (x2 \oplus x4)$, where $\oplus$ is the exclusive or.
The BDD starting at node 21 represents
$\exists x0.f(x0,x1,x2,x3,x4)= f(false,x1,x2,x3,x4) \vee f(true,x1,x2,x3,x4) =(x1 \oplus x3) \vee (x2 \oplus x4)$
If you go through the $\vee$-computation of the BDD for $(x1 \oplus x3)$ (Node 8) with the BDD for $(x2 \oplus x4)$ (Node 10), you will see that every pair of nodes is considered.