系統晶片驗證 (SoC Verification)

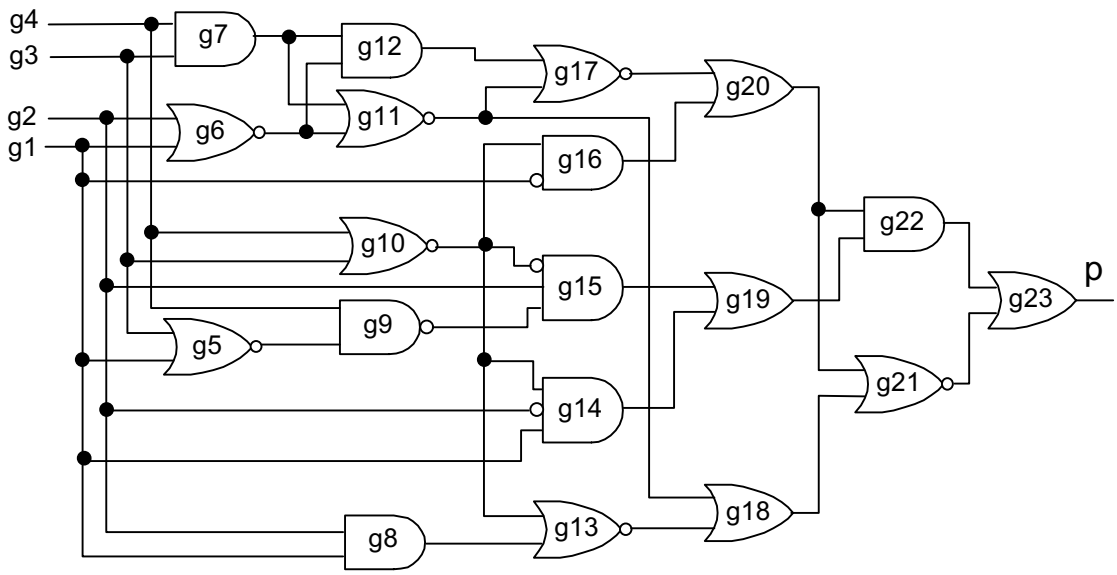
105 學年下學期 電機系電子所選修課程 943 U0250

Homework #4 [Boolean Satisfiability]

(Due: 9:00pm, Wednesday, May 10, 2017)

Note: Answer Problems 1 and 2 in plain text files as "*p1.ans*" and "*p2.ans*" (i.e. NOT Word or PDF). Remove the object files (i.e. *.o), core dump, and executable of Problem 3 (in "*p3*") before turning in. Rename homework directory as "*yourID_hw4*" (e.g. *b77503057_hw4*) and compress it by "tar zcvf *yourID_hw4.tgz yourID_hw4*".

1. In the circuit below, the bubble "○" means "inversion", and the dot "●" is for the wire connection. The ID for the gate is the "number" on its name. For example, the ID of gate "g8" is 8.



Please answer the following sub-problems with sufficient details.

(a) What are the CNF formulae for the following primitive gates using Tseitin transformation?

(i) f = inv(a);

(ii) f = and(a, b'); // b' is the inversion of b

(iii) f = nand (a, b);

(iv) f = or(a, b);

(v) f = nor(a, b);

(b) Using Tseitin transformation to generate the CNF formula for "p = 1". What are the numbers of clauses and literals of the formula?

(c) Using Plaisted-Greenbaum (PG) Encoding to generate the CNF formula for "p = 1". Note that in order to enact the polarity-cared encoding, please follow the descending order of gate IDs in examining the polarities (i.e. g23, g22,…, g1). What are the numbers of clauses and literals of the formula? Compared to (b), what are the percentages of reductions in numbers of clauses and literals?

(d) [Literal counts of a gate] The literal counts are the number of times a literal appears in the clauses. Given a variable, its positive and negative literal counts are represented as a pair "(negLitCount, posLitCount)". For example, given the following clauses,

(a + f') (b + f') (a' + b' + f)

the literal counts for a, b, and f are (1, 1), (1, 1), and (2, 1), respectively.

Please list the literal counts (as pairs) for all the gates in the circuit (including PIs) using PG encoding.

(e) [Decision order] We will determine the decision order of the gates in the circuit based on the "literal counts" of their corresponding CNF formulae (as in sub-problem (d)). The detailed rules are as follows:

(i) The "decision score" of a gate is the bigger number of its literal counts.

(ii) The decision value is the opposite polarity of the literal with the bigger count. If the counts for the positive and negative literals are the same, choose 0 as the decision value. For example, if the (0, 1)-literal counts of the gate 'f' and 'g' are (5, 2) and (3, 3), then their decision scores are 5 and 3, and their decision values are 1 and 0.

(iii) The decision order of the gates is determined by the decision scores, with the bigger scores in the front. If the scores of two gates are tied, check the counts of the other literals. If tied again, compare their IDs (bigger ID wins). For example, if the literal counts of gates 'f', 'g' and 'h' are (5, 2) and (3, 4), (4, 1), the decision order will be (f = 1) → (g = 0) → (h = 1).

(iv) The orders remain unchanged throughout the decision process.

Please derive the top 7 decision gates in the circuit (including PIs).

(f) [Conflict-driven learning] We will try to witness p(g23) = 1. Please follow the decision orders and values in (e) to make the decisions, perform logic implications, and construct the implication graph. Do not make decision on a gate if it has been implied in an earlier decision level.

You should encounter a conflict after a few decisions. Please perform the conflict analysis to derive the conflict sources on the first UIP cut and construct a learned gate (i.e. AND gate with constrained value '0' on its output) for it.

You can refer to the C++ code below for the procedure in identifying the first UIP cut.

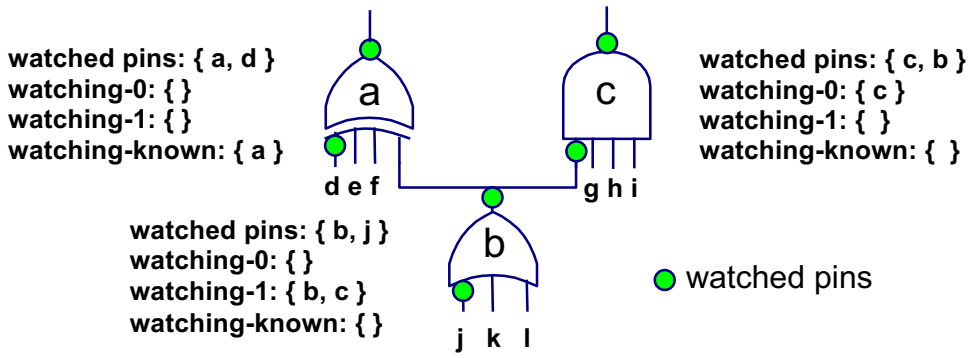
```
// ------------------------------------------------
// "imp0Src" is the list of implications that
//     imply the conflict gate '0'
// "imp1Src" is the list of implications that
//     imply the conflict gate '1'
list<ImpNode>
conflictAnalysis(const list<impNode>& imp0Src,
                 const list<impNode>& imp1Src) {
   int numMarked = 0; // num of marks in last dLevel
   ImpNode imp;
   list<ImpNode> conflictSrc;  // to be returned
   for_each_imp(imp, imp0Src)
      checkImp(imp, numMarked, conflictSrc);
   for_each_imp(imp, imp1Src)
      checkImp(imp, numMarked, conflictSrc);
   // Traverse backwards in the implication list of
   // the last decision level
   for_each_imp_rev(imp, lastDLevelImps) {
      if (!imp.isMarked()) continue;
      if (--numMarked == 0) {   // UIP found!!
         conflictSrc.push_back(imp);
         break;  // ready to return
      }
      imp.unsetMark();
      for_each_imp_src(imp_src, imp) {
         // implication sources of imp
         // (i.e. incoming edges on the imp graph)
         checkImp(imp_src, numMarked, conflictSrc);
      }
   }
   for_each_imp(imp, conflictSrc)
      imp.unsetMark();
   return conflictSrc;
}

void checkImp(ImpNode& imp, int& numMarked,
              list<impNode>& conflictSrc) {
   if (imp.isMarked()) return;
   imp.setMark();
   if (!imp.isLastDecisionLevel())
      conflictSrc.push_back(imp);
   else ++numMarked;
}
```

(g) [Witness generation] Backtrack from the learned constraint in (f) to an earlier decision level. What is the derived learned implication? At which decision level? Perform BCP on this learned implication. Will there be another conflict? If yes, perform conflict-driven learning again. If not, pick the next unassigned gate in the decision ordered list to make the next decision. Continue this process until a witness is found, or conclude that this target assignment is unsatisfiable.

2. In this problem, we will use the provided "updateWatch()" routine to update the watched pins and watching gate lists for gates 'a', 'b', and 'c'. The candidates of the watched pins include the gate itself and all of its fanins. We use a list "_wCandidates" to store these watch candidates, where the index 0 is the gate itself and index i (i > 0) is the i-th fanin of the gate. The initial watch indices for

these three gates are 0 and 1, that is, the gate itself and the first (leftist) fanin, respectively.




```
// common method for AND, OR, XOR
// called when any of the gate's watched pins gets watched value
// newIdx: watched pin index to be updated
// return ImpStatus: { IMP_DONE, IMP_NEW, IMP_CONFLICT }
// if new watch found, newIdx will be updated
ImpStatus
Gate::updateWatch(int& newIdx, const int otherIdx)
{
   assert(isWatchedValue(newIdx));
   assert(newIdx != otherIdx);
   int origIdx = newIdx;
   for (++newIdx; newIdx < _wCandidates.size(); ++newIdx) {
      if (!isWatchedValue(newIdx)) {
         if (newIdx != otherIdx)
            return IMP_DONE;  // found new watch
      }
   }
   for (newIdx = 0; newIdx < origIdx; ++newIdx) {
      if (!isWatchedValue(newIdx)) {
         if (newIdx != otherIdx)
            return IMP_DONE;  // found new watch
      }
   }
   // NOT found
   newIdx = origIdx; // restored
   if (!isWatchedValue(otherIdx))
      return genIndexImp(otherIdx);  // return IMP_DONE or IMP_NEW
   return IMP_CONFLICT;
}
```

In the following sub-problems, for gates 'a', 'b', and 'c', please derive their updated watched pins and watching lists with respect to the specified implications. In addition, if any implication is generated by "updateWatch()", please also specify.

(a) f ← 0 (b) b ← 1 (c) e ← 0 (d) a ← 1

(e) l ← 1 (f) j ← 0 (g) g ← 0 (h) h ← 1

Please note that whenever a new implication is generated (direct or indirect), we will perform direct implications based on this gate's direct implication graph first, and at the same time, schedule new "updateWatch()" if necessary.

3. The "seat assignment" problem is that, given n men (denoted as m_0, m_1,…, m_{n-1}), n seats (denoted as s_0, s_1,…, s_{n-1}), and some assignment constraints, we (as the hosts) would like to figure out an assignment for the n men on n seats that satisfies all the constraints. There are 5 types of constraints:

   (i) Assign(m_i, s_j): man m_i must be seated on seat s_j.

   (ii) AssignNot(m_i, s_j): man m_i cannot be seated on seat s_j.

   (iii) LessThan(m_i, m_j): man m_i must be seated on a seat with the number less than that of man m_j.

   (iv) Adjacent(m_i, m_j): man m_i must be seated adjacent to man m_j.

   (v) AdjacentNot(m_i, m_j): man m_i cannot be seated adjacent to man m_j.

   You will be given an input file to describe the constraints. An exemplar input file is as follows:

```
10
LessThan(8, 3)
Adjacent(4, 6)
AssignNot(1, 3)
```

   , where the number in the first line specifies the number of men (as well as seats), and each line in the sequel defines a constraint.

   You are asked to write a program that can: (1) read in the constraints from the input file, (2) convert the constraints to CNF format, (3) call SAT solver to solve the problem, and (4) print out the result on the screen.

   Name the executable as "*seatAss*", and it should take exactly one argument as the input file name (e.g. "*seatAss ex1*"). The output should be one of the following two formats:

```
Satisfiable assignment:
0(3), 1(2), 2(0), 3(1)
```

   or

```
No satisfiable assignment can be found.
```

   , where for the satisfiable assignment, the assignment `0(3)` means man m_3 is seated on seat s_0.

   The SAT package (the classic miniSat-1.14) and a wrapper class (defined in "*sat.h*") are provided in the directory "*p3*". Please refer to the testing program "*test/satTest.cpp*" for the exemplar usage of the SAT solver. Note that you should call the member functions in "*sat.h*" to generate the proof instance and call the solver. DO NOT write out the CNF formula to a file and call the SAT solver separately. Name your main program "*seatAss.cpp*" under the "*p3*" directory and provide a "*Makefile*" to compile your program. We will test your program with various constraint files.