Midterm Homework and CAD Project

This assignment is due at *4pm on Monday, March 28* (note the changed date!).

Grading. This entire assignment is worth approximately 21% of your final grade. It consists of three parts: (i) CAD programming problem (Handout #23a); (ii) SIS CAD tool multi-level application problem (Handout #23b); and (iii) written problems (below in this Handout #23). Part (i) is the programming project; it is worth 15% of your final course grade. Parts (ii) and (iii) together form a smaller midterm homework assignment, together they are worth 6% of your final course grade.

Working in Groups. You are *only* allowed to work in groups on problem (i): the CAD programming problem (Handout #23a). For this part, you can choose either to work in a group-of-two or solo. If you work in a group, you both get the same grade.

However, for parts (ii) and (iii) (SIS and written problems), you *must work solo* and hand in your own individual solutions.

1. **CAD Programming Mini-Project: Creating a CAD Tool for Multi-Cube Extraction.** This problem is the mini-project; it is worth 15% of your final grade. It allows you the opportunity to create and test out your own CAD tool for a key step in multi-level logic optimization: multi-cube extraction. *See Handout #23a for details.*

2. (40 points) **CAD Tool Tutorial: Introduction to Multi-Level Optimization Using SIS.** This tutorial and problem is an introduction to multi-level optimization, using the UC Berkeley "SIS" framework. This famous public-domain tool includes many of the optimization algorithms we are covering in class (and much more!). It also forms the skeleton of many of the commercial CAD tools at companies such as Synopsys, Cadence and Mentor Graphics. Over 1800 research papers cite the original 1992 SIS paper by Sentovich et al., many of which used the tool for deriving or comparing research results; type in "SIS Sentovich" under www.googlescholar.com.

   The SIS tool is no longer actively maintained, and has some peculiarities: it has some sensitivity to whether or not you write out/read in a file, since this can reorder the data structure for the circuit; it also has sensitivity to the state of tool – whether it is rebooted or not. The tutorial includes some detailed instructions so that you can generate *reproducible results* for grading. *See Handout #23b for details.*

3. (5 points) **The Algebraic Model: Basics.**
   (a) Write a *cube-free* algebraic expression.
   (b) Write a *non-cube-free* algebraic expression.
   $\Rightarrow$ For parts (c) through (d) below, assume $x = abc + bef + abf + bgh$.
   (c) Give an algebraic divisor which is a *factor* of x, and give the corresponding quotient (do this by inspection).
   (d) Give an algebraic divisor which is *not* a factor of x, and give the corresponding quotient and remainder (do this by inspection).
   (e) Given algebraic expression $y = ach + abe + abcf + acfg + bh$, (i) *list all kernels* of y; (ii) for each kernel, indicate both its *corresponding co-kernel* and *its kernel level*.

4. (15 points) **Single-Cube Extraction.** You are given the following algebraic expressions, forming a logic network:
   $x = bcdeg + adf + adeh + aceg$
   $y = acdef + cdefg + h$
   $z = bcd + acef + i$
   (a) *Deriving co-kernels and kernels:* For each local function (x, y, z), list all the kernel/co-kernel pairs. Be sure to include the trivial kernel/co-kernel if appropriate. *Note:* You can simply do this by inspection, rather than apply the algorithm, as long as you find the complete solution.
   (b) *Single-cube extraction:* Based on non-trivial co-kernel intersections, list *all* possible single cubes that can be used for single-cube extraction of these functions. (*Note:* recall that the single cube to be extracted must have 2 or more literals, and must be the result of co-kernel intersection across two or more of the local functions.)
   (c) *Resulting circuit implementation:* Pick the answer to part(b) with optimal "gain," i.e. a single cube to extract which results in the greatest overall reduction in literal count in the logic network, and show the result logic network after this extraction.

5. (15 points) **Multiple-Cube Extraction.** You are given the following algebraic expressions, forming a logic network:
   $x = cd + dgi + acgi + bgi$
   $y = acgh + dgh + e$
   $z = bd + abc + bf + h$
   (a) *Deriving co-kernels and kernels:* For each local function (x, y, z), list all the kernel/co-kernel pairs. Be sure to include the trivial kernel/co-kernel if appropriate. *Note:* You can simply do this by inspection, rather than apply the algorithm, as long as you find the complete solution.
   (b) *Multi-cube extraction:* Based on non-trivial kernel intersections, i.e. using Brayton/McMullen's Theorem, list *all* possible multiple-cube expressions that can be used for multi-cube extraction of these functions. (*Note:* recall that the multi-cube expression to be extracted must have 2 or more cubes, and must be the result of kernel intersection across two or more of the local functions.)
   (c) *Resulting circuit implementation:* Pick the answer to part(b) with optimal "gain," i.e. a multi-cube expression to extract which results in the greatest overall reduction in literal count in the logic network, and show the result logic network after this extraction.

6. (25 points) **Unate Recursive Paradigm: Function "Similarity" Evaluation.**

In this problem, you are to use your experience with unate recursive algorithms to design a new algorithm: *to evaluate the "similarity" of two Boolean functions, f and g.* As in previous problems, this one gives you an opportunity to explore the power of this divide-and-conquer strategy for quite varied applications.

Overview. Given two single-output Boolean functions f and g, it is sometimes useful to determine how "similar" the functions are, especially when incremental design changes are made. For example, suppose a combinational function block was implemented for function f, and a designer made modifications to the block, where the new combinational function is g. We define the **similarity** *s(f,g)* **of functions f and g** as the *percentage of all minterms where the two functions have the same values.* From an engineering perspective, the similarity metric $s(f,g)$ is the percentage of all possible input vectors under which the two combinational blocks will produce the same functional result.

Discussion. In this problem, your goal is to follow some of the recursive strategies you have learned, as well as modify or extend them, to handle this new problem. In this problem, your starting point will now be *two covers*, F and G, not one cover. In addition, you will be returning a result which is a similarity percentage, rather than a cover.

Assumption: Assume that both f and g are fully-specified functions, i.e. have no don't cares (DC-set is empty). Also, assume both functions have identical support (i.e. both f and g are functions of the same input variables).

What To Do: Your algorithm will take covers F and G, and eventually return the similarity result, $s(f,g)$, which is the final percentage of all minterms where f and g have identical function values. You are to clearly and precisely outline your new algorithm.

In particular, your answer should be concrete and clear, addressing the following issues.

(i) Give a short overview (1-2 paragraphs) of your proposed approach.

(ii) How does algorithm handle two covers, instead of one? (how is splitting performed, how are results assembled?)

(iii) What termination rules do you propose? (these can be simple, but must still allow early termination). For each termination condition, what result is returned? You should try to find a varied and powerful set of termination conditions, including simple basic ones, as well as more sophisticated ones. However, in each case, the rule must be simple and fast to apply.

Note: your conditions should be easy to apply, and should improve the search rather than making it more complicated or slower. Follow the guidelines of the existing algorithms, and see if you can also come up with new conditions, too.

(iv) What form of Shannon decomposition will you use? Given your experience in tautology checking and fast complementation, write out the new Shannon decomposition equation for returning the resulting cover H (i.e. similarity function result), given the two initial covers F and G, and the requirement of returning a similarity percentage at each step. How is the similarity percentage formed and returned at each step of the recursion?

(v) Can you exploit unateness properties in (a) forming termination rules, or in (b) directing and simplifying the search? (By analogy, for tautology check, note that the unateness condition resulted in Rule U1 for (a), and for Rule U2 for (b); see if you can come up with analogous conditions for (a) and (b) for this problem.)

(vi) What criteria do you propose to select a good splitting variable, given that the algorithm handles two covers instead of one?

Note: More points will be awarded for creative and especially-effective solutions, but a solid and complete but somewhat suboptimal solution will still get a high percentage of the total points.