

California Institute of Technology  
Department of Computer Science  
Electronic Design Automation

CS137a, Winter 2004

Assignment #3&amp;#4

Monday, February 9

**Assignment #3 Due:** Friday, February 20, 11:59pm.

**Assignment #4 Due:** Friday, March 3, 11:59pm.

**Resources** You are free to use any books, articles, notes, or papers as references. Provide citations in your writeup as appropriate.

**Collaboration** You may discuss algorithm and software engineering strategies with your classmates. You are to provide your own implementation. Except for code provided you, you should write all the code yourself. If your solution derived from significant discussion with others, please acknowledge them appropriately in your writeup.

**Writeup** Writeup should be in an electronically readable format (HTML or PDF preferred). State any assumptions you need to make.

## Problem

Find a two-level implementation of a multi-output function with extra outputs added to allow the detection of any single error. The two-level implementation should use as few product terms as possible.

The error-model is as described in class. Any product term or output can have the wrong value on a cycle. Your logic should either produce the correct output (when the error is not observable) or produce an incorrect codeword (parity inconsistency will catch error).

## Subproblem

We are specifically suggesting you attack this by using multiple parity groups. So, a more focused statement of the problem is: decompose the outputs into parity groups, and select even or odd parity for each group, so as to minimize the total number of product terms.

The two simple extremes are:

1. Single Parity – compute a single parity output; remember there can be no sharing of product terms between output bits in order to maintain the single-fault detection property.
2. Duplicate Logic – compute the function twice; one could think of this as providing a parity bit for each output. Recall that we are allowed to share logic between **different** parity groups, so this allows complete sharing of product terms within each copy of the logic.

We hypothesize that there are better solutions between these two extremes which balance the cost of additional parity logic with the savings achievable from product term sharing.

Notes:

- You must preserve the output polarity of the original outputs of the network.
- You may choose even or odd parity for each of your parity groups. This allows you, for instance, to implement the complement of a signal in the “copy” logic in the duplicate case above if the complement is cheaper to compute than the original output.
- We are not asking you to consider the cost of the checking logic or the cost of the additional outputs in selecting your minimum solution. We presume that whatever general search strategy you devise could easily be adapted to these other cost models.

## Tools and Infrastructure

You may use ESPRESSO as a subroutine to optimize 2-level logic for you. You may re-implement portions of the ESPRESSO algorithm if you feel that is necessary. We provide a Java-based class infrastructure for representing and manipulating two level logic. You can pickup this infrastructure in the form of a tar file from: `/cs/courses/cs137/2004/assignment3_4/package.tar`

To unpack and get started:

- copy package.tar to your working directory
- unpack (e.g. `tar -xvf package.tar`)
- note that source code lives under `src/` and 8 test PLAs are provided in `test/`
- build executable class files using: `ant`
  - You can find an `ant` executable in: `/cs/research/ic/software/apache-ant-1.5.3-1/bin/`
  - Note the `dist` subdirectory created during your build.
  - You should be able to run `duplicate` once you've built the executables.
  - Typically you would run: `.../dist/scripts/icrun.sh duplicate -phase true -O /tmp/outdir test/add4.pla`
- build the Javadoc using: `ant doc`  
Documentation lives in the `docs` subdirectory. Point your browser at `.../docs/index.html`.

We will support this infrastructure on the CS linux systems. You are not required to use this infrastructure, but we will not support you if you use something else.

- It should be moderately easy to adapt this to work under cygwin. It will probably require some standard changes to `EspressoWrapper` to properly invoke a windows `espresso` executable.
- It would be possible to modify the `espresso` code directly, but we suspect that will be more trouble than using the Java wrappers we provide.

## Assignment #3: Due February 20th

1. Adapt the code we've provided to compute the single-parity cover. (This should entirely be an issue of using the infrastructure we've provided and understanding what the solution needs to look like. There should be no algorithmic development required to assemble this solution. We've even provided code to actually compute the parity function.)
2. Generate a driver that partitions the original set of outputs into two output groups and provide a parity for each group. (For this particular assignment, we don't care about the quality of the results. You can partition randomly or using some oblivious scheme. We simply want you to be over the hurdle of using the infrastructure to create partitions.)
3. Based on things you've learned in this course, make a list of a set of techniques and attacks that might be applicable to this problem. Just given a sentence or two on each technique.
4. Sketch psuedocode for the algorithm you plan to use to solve the problem. (You may change your mind later. We want one, well thought out solution.)

## Assignment #4: Due March 3rd

1. Develop an implementation for your two-level PLA covering, single-error detecting algorithm.
2. Writeup should include:
  - (a) Basic problem formulation (including optimization criteria and cost model)
  - (b) Outline of solution, including algorithm description (could be same as on Assignment #3, but we suspect it will evolve as you get feedback from implementation)
  - (c) Instructions on how to run your optimizer (arguments, options, etc.)
  - (d) Results and comparison (benchmark results on all provided test cases, etc.)
    - you at least want to compare your best results with the single parity results (from Assignment #3) and the duplication result (from the provided `duplicate` implementation).
    - breakdown the overhead of solutions into sharing costs and parity costs
  - (e) Lessons and recommendations including:
    - important issues and lessons in the design and implementation of the algorithm;
    - discussion of the optimality of the solutions produced by this algorithm and implementation;
    - discussion of the next set of experiments and studies which would be beneficial to further improve this algorithm and implementation.