EECE6086 - HW 3

Max Thrun — Samir Silbak

April 18, 2014

1 Objective

The objective of this lab is to implement an algorithm based on the unate recursive paradigm (URP) which uses a heuristic called `BINATE_SELECT` to choose a variable in the recursive Shannon expansion. Therefore, given a cover `F`, we must determine if the cover is a `tautology` or not. If cover, `F` is not found to be a `tautology` then we must perform an algorithm (the complement) to find the missing covers in order to make the cover a tautology.

2 Implementation Details

2.1 Tautology Checking

2.1.1 Unate Reduction

In order to check if the given cover is a tautology we check for a few special cases:

- If a cube (minterm) in the given function (tautology) has all don't cares, it is found to be a tautology,
- If a column is all unate, it is found to be not a tautology,
- If the total number of minterms is less than 2^n, then the cover is not a tautology.

If our tautology does not cover any of the special cases as described above then we go ahead and perform unate reduction. During this process we must rearrange the cover so that the input matrix is in this form:

$$F = \begin{bmatrix} U & F1 \\ D & F2 \end{bmatrix}$$

Where `U` are the unate columns and `D` is a matrix of all don't cares. Once we have reduced all the unate columns (if there were any) then we checked to see if we had any cubes with all don't cares. We then try to find the most binate column. We do this because we would like to find the variable that is most dependent, meaning that all other variables depend heavily on this variable. This makes it easier to eliminate cubes and terminate the program faster. After selecting the most binate variable, we are now ready to cofactor (Shannon expansion). We do this by following this simple boolean equation: $F = x * F_x + \bar{x} * \bar{F}_{\bar{x}}$

2.2 Data Structures

2.2.1 Tautology

2.2.2 Complement

3 Work Division

4 Usage

Building is accomplished via a `Makefile` which generates an executable names `main`. Additional build opions are shown below.

```
$ make              # regular build
$ make clang        # builds using clang++ instead of g++
$ make debug        # enable debug printfs
$ make clean        # removes all binaries, object files, and benchmark results
$ make benchmarks   # runs all the benchmarks in the 'benchmark' directory
$ make pngs         # converts all the benchmark .svg files to .png
$ make jpgs         # converts all the .png files to reduced resolution .jpg files
```

The program accepts a single argument which is a file containing the netlist formatted as specified in the assignment. All log and debug output is printed to `stdout` and the output .mag and .svg files are named the same as the input file.

```
$ ./main benchmark/1 > benchmark/1.log
$ ls benchmark/1.*
benchmarks/1.log  benchmarks/1.mag benchmarks/1.svg
```

5 Results

5.0.3 Performance

Benchmark	Tautology	Execution Time(s)	Memory	Algorithm Used
Cover1_8_10	NO			Heuristic Method
Cover2_8_100	NO			Flags Method
Cover3_8_250	YES			Flags Method
Cover4_15_1000	NO			Heuristic Method
Cover5_15_10000	YES			Flags Method
Cover6_15_30000	YES			Flags Method
Cover7_20_10000	NO			Heuristic Method
Cover8_20_100000	YES			Flags Method
Cover9_20_1000000	YES			Flags Method
Cover_25_100000	NO			Heuristic Method
Cover_25_1000000	YES			Flags Method
Cover_25_10000000	YES			Flags Method
Cover_30_1000000	NO			Heuristic Method
Cover_30_10000000				
Cover_30_100000000				

Table 1: Execution time and memory for each run of cc and tc

Note: If the cover was found to be not a tautology, the complement algorithm was run, therefore column two conveys whether cc or tc was run. For example, in Row 1, we see that the cover is not a tautology, therefore the execution time and memory used takes in account for both cc and tc, whereas for `Cover3_8_250`, it was found to be a tautology so the data reflects only for the tc program.

6 Retrospective

Overall, we are pretty satisfied with both the execution speed and quality of our results. All of our execution times with the exception of benchmark 7 are under half a second. We also feel that our memory usage is minimal as we only store one vector of actual cell objects and all other data structures just contain pointers back to the original objects. We also make heavy use of pass by reference to avoid needlessly copying big data structures into functions.

In terms of placement and routing quality there is always room for improvement. It's easy to visually look at almost any result and see ways that it could be improved but how to translate these improvements into algorithms is not always so obvious. We believe, however, that we were are able to handle a lot of specific cases (like moving the feed-throughs to the other side of the cell) which in the end resulted in superior layouts. For the force directed placement algorithm it is hard to tell if the results you get are truly the best, especially when the circuit is so large you cannot visually comprehend it. We feel like we did our best analyzing it empirically and tuned is to best suit our benchmark files. One of the main problems with the force directed algorithm is that it tends to pull all the cells into the center of the layout. We tried to fix this by redistributing the unconnected cells evenly through the layout and then re-force directing the rows but it is still not always ideal. One interesting method to pursue would be using a combination of min-cut / quadratic placement and then force directing the sub divisions (or vice versa). If we were to min-cut horizontally we might be able to reduce some nets that currently span multiple rows.

7 Appendix