University of Toronto, Faculty of Applied Science and Engineering
Department of Electrical and Computer Engineering

ECE 1387 – CAD for Digital Circuit Synthesis and Layout
Exercise #2 – FPGA Technology Mapping with the ABC Logic Synthesis Framework

Fall 2015 J. Anderson

Assignment date:	November 25, 2015
Due date:	December 4, 2015, at beginning of class.
Late penalty:	-1 per day late, with total marks available = 10

The purpose of this exercise is to gain familiarity with the ABC logic synthesis framework, and in particular, use ABC to perform FPGA technology mapping to look-up-tables (LUTs). ABC is a state-of-the-art logic synthesis tool, developed primarily by Alan Mishchenko at UC Berkeley over the past few years[1]. As mentioned in class, in ABC, the subject graph consists of 2-input AND gates and inverters – an *AND-inverter graph* (AIG).

ABC is part of the VTR framework, so if you already downloaded that for Exercise #1, you do not need to re-download it; otherwise, download VTR from the Piazza website. Enter the directory "abc_with_bb_support" and type "make" to build the executable "abc". I verified this works on ECF.

For all steps of this exercise, you should use the **20** test circuits on the course website. There are called the *MCNC benchmark circuits* and they are commonly used in FPGA CAD and architecture research. I suggest you write your own scripts to automate the steps of this exercise for the 20 circuits. The test circuits are in BLIF (Berkeley Logic Interchange Format). Take a look at one of the BLIF inputs: the circuit is represented as a directed acyclic graph, where the nodes represent logic functions. BLIF gives each node's logic function as a truth table. Don't-cares in BLIF truth tables are represented using a hyphen(-).

For technology mapping, you will use the "priority cuts" FPGA technology mapper[2]. The `if` command executes the priority cuts algorithm. This algorithm is very similar to the DAGON algorithm used in class. Namely, it applies dynamic programming to map the AIG circuit into LUTs (instead of standard cells asin DAGON).

What to do and what to hand in?

1. Start ABC by typing `abc`. Read each circuit into ABC, perform technology independent optimization, and map the circuit to 4-input LUTs. ABC commands for the circuit `alu4.blif`:

```
read_blif alu4.blif
resyn2    // technology indep. optimization (you need to have abc.rc in your directory)
print_stats   // prints details about the AIG
```

[1] http://www.eecs.berkeley.edu/~alanmi/abc/
[2] Alan Mishchenko, Sungmin Cho, Satrajit Chatterjee, Robert K. Brayton: Combinational and sequential mapping with priority cuts. ICCAD 2007: 354-361

`if -K 4` // technology mapping to 4-input LUTs
`print_stats` // prints details about the mapping solution
`write_blif alu4.mapped.blif` // writes the mapping in BLIF format

The output from the `print_stats` command before mapping gives the number of 2-input AND gates in the AIG; the number of inverters is not reported as inversion is managed as an attribute on edges of the AIG. The output from the `print_stats` command after mapping gives the number of LUTs (`nd`) in the mapping solution and the number of LUT levels on the critical path (`lev`) – i.e., the mapped circuit depth. You can compare the output from the two calls to `print_stats` to gauge the number of 2-input AND gates each LUT implements, on average. For each test circuit, report the number of LUTs in the mapping, and its depth. Take a look at the mapped BLIF and you will see that each node uses at most 4-inputs, and can therefore be realized in a 4-input LUT.

2. Until about 2007 or so, Xilinx and Altera FPGAs used 4-input LUTs. Modern FPGAs (65*n*m and below) use 6-input LUTs. The reason for this architectural change is speed: with 6-LUTs, fewer levels of LUTs are needed. Repeat step #1 above, except this time, map the circuits into 6-input LUTs (`if -k 6`). Report the number of LUTs and the depth for each circuit. By how much was depth reduced (on average) when 6-LUTs were used versus 4-LUTs?
3. By default, the priority cuts algorithm in ABC optimizes circuit depth. With the `-a` option, the mapper can optimize silicon area (# of LUTs). Repeat step #2 above (6-LUT mapping), optimizing for area (`if -K 6 -a`). By how much was area reduced, on average, versus the results in step #2? How much was depth affected?
4. As mentioned in class, technology *independent* optimization (before tech mapping) can affect the mapping results. In `abc.rc` you will find the definition of the `resyn2` script (it calls other ABC commands like AIG balancing (`b`) and re-writing (`rw`)). Experiment with using other scripts, namely, `resyn`, `resyn2a` and `resyn3`, and repeating step #2 above. Give a table showing the average and standard deviation of LUT count and depth, for all 4 scripts considered (including `resyn2`).
5. Modify the mapper's cost function in ABC to produce mapping solutions that represent an *intermediate* point between the mapping solutions produced in #2 (minimum depth) and #3 (minimum area). In particular, go into the `src/map/if` directory in the ABC distribution and edit `ifCut.c.` Alter the `If_ManSortCompare` function, which compares the relative value of two cuts. A cut is represented a C `struct` (`If_Cut_t`), wherein the `Area` field represents the # of LUTs in the fanin cone, the `Delay` field represents the depth, the `nLeaves` field represents the number of inputs to the cut, and the `AveRefs` represents the average fanout of the inputs to the cut in the mapping. (I suggest you combine these already-implemented parameters in your new cost function.) Once you have modified the cost function, recompile ABC. Repeat step #2 above. In your write-up describe the cost functions you tried and the give details on the one you found to be most successful.

Aside: The main mapping routines are in `ifMap.c`. The function `If_ObjPerformMappingAnd(…)` generates all of the feasible cuts for a node by combining cuts from its fanins, and also selects the “best cut” using the dynamic programming approach we talked about in class. `If_ManPerformMappingRound(…)` is the top-level function that walks through all of the nodes in topological order and calls `If_ObjPerformMappingAnd(…)` for each one.