University of Pennsylvania
Department of Electrical and Systems Engineering
Electronic Design Automation

ESE535, Spring 2013

Assignment #2

Monday, January 28

Due: Part A: Monday, February 4, beginning of class.
Due: Part B: Monday, February 11, beginning of class.

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 algorithmic and testing approaches **away from** computers before February 4th. You may give tutorial assistance on using OS, compiler, and debugging tools. All code development should be done independently. You may **not** share code or show each other code solutions. All writeups must be the work of the individual.

Writeup Turn-in assignments on blackboard. See details on course web page. No hand-writing or hand-drawn figures. See details below on what you need to turn in and the format.

Project Overview We will be developing the tools to match, cover, and place a circuit netlist onto a partially defective FPGA. Specifically, we will be dealing with a class of partial failures of the Look-Up Tables (LUTs) used to implement programmable gates. The assignments decompose the problem into pieces to roughly match our coverage of material in the course. Each successive assignment will progress toward the final problem. This extends work on mapping to high variation FPGAs we published in [1]; however, in the FPGA2012 paper we only addressed interconnect. This specifically expands work Nikil Mehta started in his PhD thesis [2]. Section 3.1.7 sets up the opportunity and Section 5.4 describes his preliminary work. Nikil has identified an interesting problem, and I believe there is an opportunity to do better with a more comprehensive solution, which we will try to develop during this course.

Model We model each physical 4-LUT as a tree of 15 2:1 multiplexers (See Figure 1). As Nikil has established, the common failure mode under variation is not that the multiplexer does the completely wrong thing, but that it cannot dynamically drive its output to different values. If it happens that the multiplexer only needs to ever drive a 0 or a 1, such as when both of its data inputs are always 0 or always 1, then it will still work. However, if it needs to dynamically drive either a 0 or a 1, such as when its data inputs are 1 and 0 or 0 and 1, a failed mux will not be able to do that. Consequently, we classify each multiplexer as either **fully** functional (can pass anything) or **constant-output** functional (can only always drive 0 or always drive 1). Our goal in mapping is to assign the logic functions to the physical LUTs so that the constant-output multiplexers never see differing inputs.

Opportunity The first opportunity we have is to select the physical 4-LUT to which a logical function is mapped. An FPGA will have hundreds (millions) of 4-LUTs and the

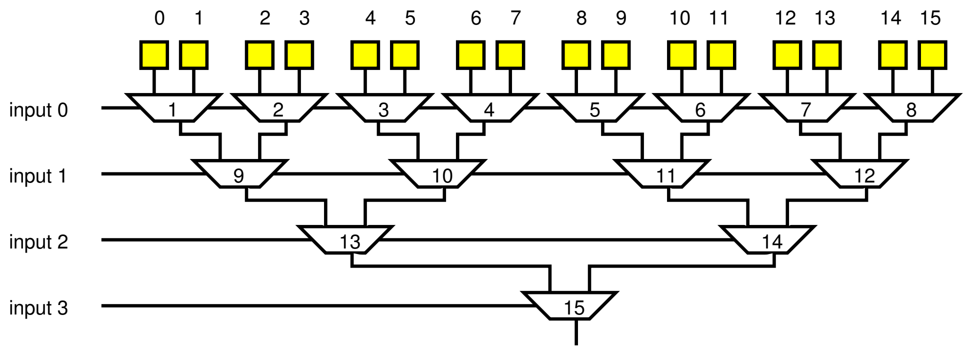

Figure 1: Physical 4-LUT Implementation

defects are randomly distributed. So, we get to choose which 4-LUTs to use. However, we can do better. We can also invert the polarity of the inputs and possibly permute the inputs. Both of these will change how the logical LUT function is mapped onto the physical LUT. It is understanding the impact of this local transformation that we want to characterize in this first assignment. In later assignments we will deal with the issue of mapping to different physical LUTs if we cannot use these local changes to make a logical function compatible with a partially-defective physical 4-LUT.

Warmup Exercise Consider the function: `(A xor B) * C * D`.

1. Assuming that A maps to input 0, B to input 1, C to input 2, and D to input 3, what is the value (0 or 1) of each of the 16 configuration memory bits?
2. Assuming that A maps to input 0, B to input 1, C to input 2, and D to input 3, identify the set of multiplexers that must be **fully functional** and the set that can be **constant-output**.
3. Identify how the set of tolerable **constant-output** multiplexers changes if you invert the polarity on input 2. That is, input 2 now receives /C rather than C, but you want to implement the same function.
4. Identify how the set of tolerable **constant-output** multiplexers changes if you now assign B to input 3 and D to input 1, leaving A and C assigned to inputs 0 and 2.
5. Considering polarity changes and input permutation, what mapping transforms may change the number of tolerable **constant-output** multiplexers and what transforms will not?
6. How many different input permutations are there?
7. Considering both polarity selection and input permutations, how many different ways are there to map a logical 4-input gate to a physical 4-LUT?

Assignment 2 Task Develop and analyze a cost function to identify how hard a logical function (a gate) will be to map to a partially defective physical LUT.

- As a "ground-truth" metric, we will compute hardness as the number of partially defective LUTs in the set of all possible defect sets for a 4-LUT that are compatible with the function. The fewer that are compatible, the harder the logical function will be to map.
- You want to find a cost function you can calculate just based on the logical function that is predictive of this "ground-truth" metric. That is, you do not want to compare compatibility to all possible defects in order to assess the hardness of mapping a logic function. This cost function should be inexpensive to compute.

You have two weeks for this whole assignment, but it is broken into two parts with a component that needs to be turned in each week.

Part A: Implement the "ground-truth" characterization and a simple cost function we give you.

CostFunA(fun): max over all permutations of inputs, $perm$

$$cpair(fun, perm) = \sum_{j=0}^{7} (c_{2j} == c_{2j+1}) \tag{1}$$

where c_j is the j-th configuration bit in the 4-LUT for the permuted function. That is, $cpair$ is a value form 0 to 8 that adds a one for every identical pairing of configuration bits.

We don't think this is the best cost function. Rather, we think it is an easy starting point that you can definitely improve upon in Part B.

Part B: Determine and implement a better cost function.

Code Base: A heavily used academic package that performs clustering, placement, and routing is t-vpack/vpr from the University of Toronto [3, 4, 5]. We are using code from the t-vpack/vpr distribution as a basis for our work (reading the initial netlist, representing the netlist in C, writing out the final cluster and placement). Using this code base, we avoid having to rewrite these I/O and representation routines, allowing us to focus on the optimization.

This assignment only leverages the BLIF reading capabilities of the code base. We will use more of the functionality in later assignments.

For later assignments, you may want to consult the VPR manual (available in `~ese535/spring2013/manual_430.pdf`) for descriptions of the mesh architecture and placement coordinate system. Particularly Figure 2 shows what the basic module of a LUT and FF looks like. Figure 10 shows the coordinate system. The manual also defines the netlist format. Since we provide code to read and write this formats, you do not have to implement it, but you will likely find it useful for debugging to be able to look at these files and make sense of them.

We are providing an infrastructure in C. In addition to providing the base t-vpack/vpr netlist capabilities, we are also providing support for representing LUT programming bits.

Pickup the code in `assign2.tar` from `~ese535/spring2013/assign2.tar` on `eniac`. Unpack it with `tar -xvf assign2.tar`. Run `make` to build. This should produce an executable `defect_main` which you can run. The `makefile` in the `test` subdirectory runs `defect_main` on the various cases needed for this assignment and provides an example of how to use it. Please use the architecture and target parameters in the `makefile` for producing your results for this assignment.

For this assignment, we provide the basic code outline, but you will need to complete various functions as identified below. In later assignments you will have more responsibility for code structure and decomposition.

Representations: We will represent a permuted LUT function as an integer. We will similarly represent defects as an integer where a bit value of 0 represents **fully** functional and 1 represents **constant-output**. Position are as shown in Figure 1. This means position 0 is unused.

A quick overview of code:

- `defect_main.c` — contains the `main` function that drives the overall optimizer; it also contains the command-line option parsing. You may need to modify this to enable various debugging options. However, note that we will likely provide you an updated main functions for later assignments, so be prepared to merge your changes and ours.
- `globals.h` — defines global data structures: notably the `block` and `net` datastructure that represent the netlist.
- `defect_main.h` — defines the type structure for `block` and `net`.
- `output_clustering.c` — prints out the cluster. You should not need to touch.
- `read_blif.c` — prints out the cluster. Also creates initial lut encoding (.lut field in s_block) You should not need to touch.

- `defect.c` – provides the compatibility check between a LUT function and a partially defective physical LUT. Also includes code to compute defect compatibility for all LUTs in the design.
- `ff_pack.c` — packs LUTs and FFs. You should not need to touch.
- `heapsort.c` — a sort implementation. You should not need to touch. You may find it useful to use this. There is an example of use in `asap.c`
- `queue.c` — a queue implementation. You should not need to touch. You may find it useful to use this.
- `util.c` — various utilities. You should not need to touch. You may or may not want to use some of these utilities.

You need to complete code in:

- `transform.c` — perform permutations and input polarity inversions, explore full set of transforms.
- `cost.c` – implement CostFunA , then your cost function for Part B.

Statistics: For this assignment we will want to visualize and compute the correlation between the "ground truth" hardness calculation and our cost functions. To do that, the code will produce an output file with comma-separated values (CSV) to be read by a statistics package, R. We provide an R script (`correlation.R`) to produce a correlation plot and calculate the correlation coefficient.

Caveat: The code not borrowed from t-vpack/vpr (defect.c, read_blif.c, defect_main.c) was newly written or heavily revised for this assignment. While we have tried to test it, like any recently developed code it may contain bugs. Let us know if you have any problems. Similarly, we may need to provide updated source as we fix bugs or add additional functionality.

We strongly recommend you become familiar with a debugger (`gdb` if you don't already have a favorite). Since this is C code, it is quite likely you will need to debug memory errors. It is much easier to do this with the proper tools.

We will likely ask you to use your solution from earlier assignments (like this one) as a component of or as a baseline for comparison for your solutions for subsequent assignments. So, you will want to keep your solution to each piece around for comparison.

Part A turnin: You will need to upload two files. We have created separate assignments on blackboard so that you only need to submit a single file to each assignment

1. **assign2a-warmup**: a single PDF with
    - Your answers to the warmup exercise
    - Correlation plot of "ground truth" hardness versus CostFunA estimates for either the design s1423 or frisc-em4.
    - A table summarizing Correlation Coefficient between the "ground truth" and CostFunA for all 6 sample blif files.
    - Description of what is weak about CostFunA – not just that it could be better correlated; discuss what the cost function is not capturing that may be important and/or why it is doing a poor job of characterizing the hardness

2. **assign2a-code**: a single tar file with your code (no binary files, but in an archive like the provided support so it can be unpacked and built)
   - run `make clean` in both the code and test directories
   - use `make assign2.tar` to create the tar file
   - test that you can unpack your `assign2.tar` and build and run tests from there before you upload to blackboard; we will build your code and test it.

Part B turnin: You will need to upload two files. We have created separate assignments on blackboard so that you only need to submit a single file to each assignment

- **assign2b-code**: a single tar file with your code (no binary files, but in an archive like the provided support so it can be unpacked and built – same as above)
- **assign2b-writeup**: a single PDF with
  1. Definition of your cost function
  2. Description of why this cost function is good and how you arrived at it
  3. Correlation plot of "ground truth" hardness versus your cost function for either the design s1423 or frisc-em4.
  4. A table summarizing Correlation Coefficient between the "ground truth" and CostFunA (same as Part A) and between "ground truth" and your cost function (new for part B) for all 6 sample blif files.

References

[1] N. Mehta, R. Rubin, and A. DeHon, "Limit Study of Energy & Delay Benefits of Component-Specific Routing," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2012, pp. 97–106.

[2] N. Mehta, "An ultra-low energy, variation tolerant fpga architecture using component-specific mapping," Ph.D. dissertation, California Institute of Technology, 2013. [Online]. Available: http://resolver.caltech.edu/CaltechTHESIS:10072012-230900231

[3] V. Betz and J. Rose, "VPR: A new packing, placement, and routing tool for FPGA research," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, ser. LNCS, W. Luk, P. Y. K. Cheung, and M. Glesner, Eds., no. 1304. Springer, August 1997, pp. 213–222.

[4] V. Betz, "VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs," http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html, March 27 1999, version 4.30.

[5] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, Massachusetts, 02061 USA: Kluwer Academic Publishers, 1999.