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

ESE535, Spring 2015 Assignment #2 Wednesday, January 21

Due: Thursday, January 29, 10PM

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. 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.

We expect everyone to abide by Penn's Code of Academic Integrity. `http://www.upenn.edu/academicintegrity/ai_codeofacademicintegrity.html` If there is any uncertainty, please ask.

Writeup Turn-in assignments on canvas. See details on course web page. No handwriting 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 map and optimize designs for minimum energy evaluation on the heterogeneous multicontext computing array introduced in [1] and developed further in [2] and [3]. These tools must cope with conventional CAD optimization challenges like partitioning, placement, and routing, along with unique opportunities and challenges for scheduling. We will start our flow with conventional LUT mapping, which we take as given,[1] and focus our development on tools for physical placement, scheduling, and routing during the term. As part of assignments 2–8, we will develop initial solutions to identified sub-problems. After assignment 8, the open-ended project in the later part of the course will allow you to select and pursue a promising direction to further optimize the heterogeneous multicontext mapping.

Model The basic model follows [2].

- Each PE has one physical 4-LUT.
- A limited number of LUTs, Inputs, or Outputs can be assigned to each PE. These will be evaluated sequentially on the physical 4-LUT.
- Local data memory in the PE holds the inputs to the 4-LUTs (Fig. 5 in [2]).
- Interconnect is a physical tree with 1:1 and 2:1 switches; each tree level uses a single switch type (Fig. 1 in [2]).
- The network is directional (Fig. 1 in [2]).

[1] ...and as we will cover in the course.

- We identify a logical growth schedule that characterizes the number of parents for each PE (at the leaf) or switch. Switches can have either 1 or 2 parents. PEs can have from 1 to 4× the number of LUTs allowed in the PE. For the sake of initial evaluation, we are not restricting the growth schedule to represent a particular Rent p value.
- The physical growth schedule has the same constraints, but will differ from the logical growth schedule. The physical growth schedule is smaller. When relevant, we will likely take $p = 0.5$ [c_{arch} (2 1)*] for the physical growth schedule.
- Sequential routing occurs in waves in which we route all the physical tree inputs from the c_{arch} physical inputs together. We will think of these routing waves as the atomic unit of clocking. Each PE is clocked once per wave.
- Evaluation must follow circuit precedence constraints. To first order, this means we route a level as one or more waves through the tree before routing the next level. If a level requires multiple waves, it may be possible to have mixed waves at the boundary between levels; we will be exploring this opportunity as part of our optimizations.

Simplification for Input and Outputs Ideally, we might have separate locations where input and outputs can be assigned. To keep things simple, we will treat inputs and outputs like LUTs and assign them to leaf clusters in the tree. This could model a case where we have area-IO and assign physical IOs to the PE clusters. However, even in that case, we would probably have a limit on the number of IOs per PE cluster. Proposing and dealing with a more realistic IO model would be a suitable extension to explore in the project portion of the course.

Opportunity and Challenges A key opportunity, and the one we will attack on the next series of assignments, will be to maximize the locality of signaling (reduce the distance that signals must travel in the tree). [2] showed that the synchronous energy can be lower than the asynchronous energy if the context factor (CF, Sec. 6.5.2) is sufficiently low. We will be exploring how to exploit freedom in placement and scheduling to minimize CF.

Cost Functions As a starter, we will consider two cost functions that capture important aspects of a mapping to the a sequential, heterogeneous, multicontext computing array. First we will consider the total energy spent driving wires. We identify a growth schedule as defined above. Assuming we have identified a growth schedule, g, the total signals that must travel along each parent-to-child (or child-to-parent) channel at height h in a tree is:

$$channel\_signals(h) = \prod_{i=0}^{i=h} g(i) \tag{1}$$

Assuming the area of a subtree is linear in the number of leaves, an assumption we guarantee by making $p_{arch} < 0.5$, the length of wires at height h in a tree is:

$$Length(h) \propto \sqrt{2^h} = 2^{h/2} \tag{2}$$

Strictly due to layout, we will round the $h/2$ up.

$$Length(h) \propto 2^{\lceil \frac{h}{2} \rceil} \tag{3}$$

The number of parent-to-child channels at level h in a tree is:

$$Channels(h) = \frac{2^H}{2^h} \tag{4}$$

where H is the total tree height. Putting this together, the total distance traveled over wires, and hence the total energy sending data on wire, is proportional to:

$$E_{wire} \propto \sum_{h=0}^{h=H} (channel\_signals(h) \times Channels(h) \times Length(h)) \tag{5}$$

$$E_{wire} \propto \sum_{h=0}^{h=H} \left(\left(\prod_{i=0}^{i=h} g(i) \right) \left(\frac{2^H}{2^h} \right) \left(2^{\lceil \frac{h}{2} \rceil} \right) \right) \tag{6}$$

When we consider the levelization restriction, we only need interconnect to route one level at a time, but we will need at least one routing wave for each level. The simplest way to manage this is to use a growth schedule g_{level} that can handle the interconnect for any evaluation level. Then we perform one wave for each level. When we do that, the effort of a wave is roughly the same as Eq. 6 with g_{level} in for g.

$$E_{wave} \propto \sum_{h=0}^{h=H} \left(\left(\prod_{i=0}^{i=h} g_{level}(i) \right) \left(\frac{2^H}{2^h} \right) \left(2^{\lceil \frac{h}{2} \rceil} \right) \right) \tag{7}$$

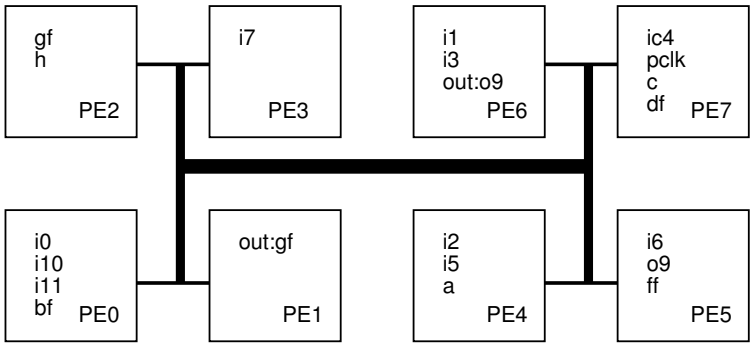
E_{wave} is more about clocking and instruction energy than wire energy, so, there is actually a different constant of proportionality for E_{wave} and E_{wire}, which we will defer identifying at this point. Now we still need to route each level, so the total cost will be:

$$E_{all\_waves} = levels \times E_{wave} \tag{8}$$

levels is the total number of levels into which the circuit is divided (critical path length or "makespan" assuming each LUT is unit delay). Then, we compute a version of the context factor:

$$CF = \frac{E_{all\_waves}}{E_{wire}} \tag{9}$$

Here we use the $\propto$'s as equality, since the denominator E_{wire} is being used like E_{wave} to capture the clocking and instruction read work. If the routing perfectly divides into levels, then $g_{level}(h) = g(h)/levels$ and $CF = 1$. In general, that won't happen, so CF tells us how much work is added due to the imperfect levelization. As noted, we will be trying to reduce CF during the term. The optional portion of this assignment explores a slightly more complex model and the opportunity it provides to reduce CF.



Warmup Exercise Consider the netlist `example.blif` provided in the `test` subdirectory of the assignment source code and the initial (poor) assignment of LUTs, Inputs, and Outputs to leaf PEs shown above (which corresponds to the placement that the provided code produces in `example.place1`).

1. Orient yourself to the BLIF netlist specification and the placement format.

   - LUTs and the following FF are merged together when possible. So you see a placement for `bf` and not `b`.
   - The `.names` directive describes each LUT. The **last** name on the line is the output, the ones before it are the inputs. (e.g., the line "`.names i2 a b`" says that `i2` and `a` are the input to a gate that produces the net `b`. The zeros and ones below the `.names` directive gives the actual logic funciton—we will learn about that format later in the course; for now you only need to know that it is some gate with not more than 4 inputs (2 inputs in this case)).

2. Calculate the value of the interconnect cost function for the given placement (Eq. 6 assuming a constant of proportionality of one).

3. Identify the minimum g_{level} growth schedule that is just large enough to support each of the required routing waves.

4. Calculate the wave energy (Eq. 7) associated with this g_{level}.

5. Using the above results (part 2 and 4), calculate the context factor for this placement (Eq. 9).

6. Identify a better placement of the LUTs and IOs. For your writeup, show which nets use each wire channel in the tree.

7. Calculate the value of the interconnect cost function for your better placement.

Assignment 2 Task Implement code to compute the interconnect cost function (Eq. 6) and context factor (Eq. 9) from a provided placement and global route for a netlist. We will use these cost functions in later assignments to assess the benefits of placement, scheduling, and routing optimizations.

`cost.c` contains the basic structure for the computation with routines for you to complete. The interconnect cost (E_{wire}) is called `interconnect_cost` in the code, and the context factor, CF, is called `context_factor`.

The code development required for this assignment is not particularly large or tricky. However, it does demand that you understand and use a relatively large code base. The main goal of this assignment is for you to begin to understand and work with this code base.

If you do complete this task quickly and would like to explore an optimization that might improve one of the cost functions, consider the following optional task.

(optional) Assignment 2 Task Identify a level factor for each level to reduce the context factor. That is, instead of assuming that we design a single wave per level, we can imagine some levels, perhaps the most demanding ones, being divided into multiple waves. This will potentially allow us to reduce the level (wave) growth factors and, perhaps, achieve a net reduction in E_{wave}. The general formation is that each level is, itself, divided into a number of waves, $wave(l)$, such that $waves(l) \times g_{wave}$ supports the traffic of the level. The total number of waves is now:

$$Waves = \sum_{l=0}^{l=levels} waves(l) \tag{10}$$

Note that if $waves(l) = 1$ for all l, the sum is $levels$, and this reduces to the simpler formulation earlier. The energy per wave becomes:

$$E_{wave} \propto \sum_{h=0}^{h=H} \left(\left(\prod_{i=0}^{i=h} g_{wave}(i) \right) \left(\frac{2^H}{2^h} \right) \left(2^{\lceil \frac{h}{2} \rceil} \right) \right) \tag{11}$$

And the total energy across waves:

$$E_{all\_waves} = Waves \times E_{wave} \tag{12}$$

Working this optional portion will give you the satisfaction of solving a more interesting problem and showing some potential improvement and a bit of good will from the instructors.

Code Base: A heavily used academic package that performs clustering, placement, and routing is t-vpack/vpr from the University of Toronto [4, 5, 6]. 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. Since we are using a tree-based interconnection network rather than a mesh, we will diverge from VPR on many physical details.

For later assignments, you may find it useful to consult the VPR manual (available in `~ese535/spring2015/manual_430.pdf`) for descriptions of some of the original assumptions in the code. Figure 2 shows what the basic module of a LUT and FF looks like. Since we are not using a mesh, many of the physical details will not be relevant. 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. Pickup the code in `assign2.tar` from `~ese535/spring2015/assign2.tar` on `eniac`. Unpack it with `tar -xvf assign2.tar`. Run `make` to build. This should produce an executable `main` which you can run. The `makefile` in the `test` subdirectory runs `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.

A quick overview of code:

- `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.
- `main.h` — defines the type structure for `block` and `net`.
- `tree.c` – provides the structure for representing the physical tree and routing. You will certainly need to use routines provided by this. You should not need to change it. We expect there will be some additions and refinement to this code as we move through the assignments.
- `domain.c` – functions for interacting with domains. For the sake of global routing and this assignment, domains are used as sets. We will interpret them differently for later assignments when we get to detailed routing. You will need to use these routines for implementing your cost functions. You should not need to change them.
- `asap.c` — compute an ASAP levelization of the netlist. This gives you the level assignment for assignment 2. You should not need to change this or interact with it,

other than using the information it places in the `level[]` array to identify the level of each block in the netlist. Later in the project, you may need to explore different level assignments.

- `global_route.c` — performs a global route of the nets in the design. You will be using the results of this—the assignment of signals to domains—as input to your cost function calculation. This code may eventually be a useful reference for you as you write your own detail routing in later assignments.
- `check_route.c` — as written, validates the global route. This was written primarily to debug the global route. We will likely refine it for detail routes (or provide an analogous version for detail routes) later in the term.
- `dummy_tree_place.c` —randomly assigns blocks to leaf PEs. This is not a good solution, but a placeholder until you can develop better solutions in later assignments. It also provides an illustration of how to interface with the routines for placement in `tree.c`.
- `output_clustering.c` — prints out the cluster. You should not need to touch.
- `read_blif.c` — parses the BLIF input files and creates the `block` and `net` internal representation for the netlist. You should not need to touch.
- `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:

- `cost.c` – Except for the gross structure and output routines in `calculate_and_print_costs()`, you need to complete all the code here as marked.

Caveat: The code not borrowed from t-vpack/vpr was newly written or heavily revised in the past week 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 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.

We strongly recommend you use version control (e.g., svn, git, cvs) for your code, with the version control reposistory on a backed-up server, such as the `eniac` file system. Furtheremore, you will want to keep track fo the code versions used for each assignment (e.g., tag versions, create a snapshot, keep the turnin `.tar` files).

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

1. **assign2-writeup**: a single PDF with
   - Your answers to the warmup exercise
   - A table summarizing your cost function results for the 6 provided benchmarks.
   - A short description of how you calculate the requested cost functions including an overview of your code.
   - (optional) A description of the algorithm you used to assign levels to optimize the context factor.
   - (optional) extend your table to include the optimized context factor result as well as the required results above.
2. **assign2-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 on `eniac` from the source in the tar file before you upload to canvas; we will build your code and test it.

References

[1] A. DeHon, "Location, Location, Location—The Role of Spatial Locality in Asymptotic Energy Minimization," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2013, pp. 137–142.

[2] ——, "Wordwidth, Instructions, Looping, and Virtualization—The Role of Sharing in Absolute Energy Minimization," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2014, pp. 189–198.

[3] ——, "Fundamental underpinnings of reconfigurable computing architectures," *Proceedings of the IEEE*, 2015, *To Appear*.

[4] 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.

[5] 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.

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