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

ESE535, Spring 2011 — Assignment #2 — Monday, January 31

Due: Part A: Monday, February 7, beginning of class.
Due: Part B: Monday, February 14, 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 7th. 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 schedule, place, and route a circuit netlist onto a multi-context FPGA such as [4, 5]. The assignments decompose the problem into pieces to match our coverage of material in the course. Each successive assignment will add additional constraints and concerns toward the final problem. We will general approach the problem as a time- and resource-constrained problem where we are trying to minimize the energy for the computation.

Assignment 2 Task Develop and implement an algorithm to schedule a circuit netlist, C, onto a time-multiplexed, programmable substrate:

- meeting a specified bound on the number of netlist nodes assigned to a PE
- achieving a target makespan (time-bounded scheduling)
- while minimizing the number of PEs required to schedule the netlist C.

For this week, we make the simplifying assumption that the dominant time is computation on the processing element (PE). We assume that results from one PE can be used on any other PE in the next cycle. This simplification allows us to focus on scheduling this week. In future weeks, we will make the problem more realistic by modeling delays between PEs and as a function of distance.

You have two weeks for this whole assignment. To encourage you to start early, there is a milestone (Part A) with deliverables at the end of the first week.

Part A: Identify the algorithm you will use for scheduling, the key data structures you will use, and any new auxiliary data structures of functions you will need to develop.

Part B: Complete your implementation and benchmark your results.

Architecture:

- Each PE (roughly equivalent to a CLB) can compute one Lookup-Table (LUT) evaluation in each cycle.
- Each PE can be programmed to evaluate a number (*e.g.* 8) of individual LUTs as long as they evaluate in **different** timesteps. The number of LUTs a PE can evaluate is an argument to the mapper (cluster size) and should be a parameter in your algorithm.
- Similarly, each periphery position around the central logic mesh is an I/O cluster that can hold a number of inputs or outputs as long as they produce or consume data in **different** timesteps.
- Each slot in each PE or I/O cluster has associated with it a timestep. This indicates the timestep on which the associated LUT, Input, or Output is evaluated.
- Interconnect among the PEs and I/Os is a mesh.
- The computation is performed by iterating through all defined timesteps in the schedule from 0 to the maximum timestep. On each timestep, each PE or I/O cluster evaluates the LUT, Input, or Output that is assigned to the timestep.

A properly formed mapping will:

- Assign each LUT or I/O to some timestep in some PE
- Never assign more LUTs to a PE than the specified cluster size
- Never assign more I/Os to an I/O cluster than the specified I/O cluster size
- Never assign two LUTs or I/Os to the same timestep in a PE or I/O cluster
- Assign each LUT or output to a timestep that is strictly greater than the timesteps of its predecessors.

The supplied code includes checks to validate these conditions and will print error messages when they are not met.

Code Base: A heavily used academic package that performs clustering, placement, and routing is t-vpack/vpr from the University of Toronto [2, 1, 3]. 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.

Note that VPR does **not** deal with time-multiplexed evaluation. Their CLBs hold Lookup-Tables that may evaluate simultaneously.

Please look at parts of the VPR manual (available in `~ese535/spring2011/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 a representation for the mesh and PEs and a baseline ASAP

scheduler. The baseline scheduler shows you how to use the infrastructure and give you an easy target against which to show improvement for your scheduler.

Pickup the code in `assign2.tar` from `~ese535/spring2011/assign2.tar` on `eniac`. Unpack it with `tar -xvf assign2.tar`. Run `make` to build. This should produce an executable `sched_main` which you can run. The `makefile` in the `test` subdirectory runs `sched_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 you a stub for your scheduler in the file `your_schedule.c`. Currently the routine does nothing. You should complete the routines.

Caveat: blocks of type `LATCH` or `LUT_AND_LATCH` have flip flops that will both start and end paths. For the VPR model and ours, these flip flops live at the PE where the LUT is evaluated. As such, you need to exercise some care to deal separately with the delays of these nodes based on whether you are looking at them as inputs to the flip-flop or outputs from the flip-flop. The ASAP delay calculator (`compute_level` and `asap_delay`) show examples of how to treat flops for delays.

Note that the `num_nets` on the `s_block` includes the clock. So, when you are dealing with a `LATCH` or `LUT_AND_LATCH`, the actual data inputs are [1..`num_nets`-2].

A quick overview of code:

- `sched_main.c` — contains the `main` function that drives the overall optimizer; it also contains the command-line option parsing. If you need to change the arguments to `your_schedule`, you will need to modify them here. You may also need to modify this to enable various debugging options. However, note that we will likely provide you an updated `sched_main.c` for later assignments, so be prepared to merge your changes and ours.
- `your_schedule.c` — this is where you add your code for your scheduler.
- `globals.h` — defines global data structures: notably the `block` and `net` datastructure which represent the netlist.
- `sched_main.h` — defines the type structure for `block` and `net`.
- `mesh.c`, `mesh.h` — routines for working with the physical mesh, including placing and moving blocks, legality checks, and printing out the placement and clustering. We will likely provide you an updated `mesh.c` for later assignments, so if you make any changes here, be prepared to merge your changes into the updated version.
- `output_clustering.c` — prints out the cluster. You should not need to touch.
- `read_blif.c` — prints out the cluster. 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. There is an example of use in `asap.c`
- `util.c` — various utilities. You should not need to touch. You may or may not want to use some of these utilities.

- `asap.c` — routines to perform both ASAP level calculation (`asap_delay`) and ASAP scheduling (`asap_schedule`) onto the array.
- `check_precedence.c` — routine that checks to see if the current schedule is complete and obeys all precedence constraints. This, too, while likely be updated for future assignments.

Caveat: The code not borrowed from t-vpack/vpr (mesh code, asap, queue, sched_main, check_precedence) 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 baseline for comparison for your solutions for subsequent assignments. So, you will want to keep your solution to each piece around for comparison. It is not an issue this week, but this is advanced notice so you are prepared.

Part A turnin: A single PDF that includes:

1. Read `asap_schedule` (including the support routines it calls) in `asap.c`. Answer the following questions (2-3 sentences each).
   (a) Why does the scheduler need to reset the (x,y) coordinates each time the ASAP level changes?
   (b) How could we avoid reseting them to (0,0), and how would this impact the efficiency of the scheduler?
2. Provide a high-level, English text description of your scheduling strategy (1–2 paragraphs may be sufficient). Use references if appropriate.
3. Define any cost functions used by your algorithm.
4. Provide psuedocode for your algorithm.
5. Identify and define the key data structures; describe their purpose and use in the mapping algorithm.
6. Identify any additional helper functions you plan to develop.
7. Identify any challenges or concerns you still anticipate for your solution.

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; this means the make file should be updated to build the application with any additional source files you may have added)
  - 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

- **assign2b-writeup**: a single PDF with

1. Description of your algorithm and the code that supports it; we expect this may evolve from your Part A turnin.
   (a) A high-level, English text description of your strategy. Use references if appropriate.
   (b) Define any cost functions used by your algorithm.
   (c) Provide psuedocode for your algorithm.
2. Completed version of this comparison table with your results:

Design	**ASAP delay**	PE cnt 1.0×ASAP delay			PE cnt 1.2×ASAP Delay			PE cnt 2.0×ASAP Delay		
		ASAP	Yours	Δ%	ASAP	Yours	Δ%	ASAP	Yours	Δ%
example	3	5			5			5		
e64	5	117			117			117		
s1423	15	54			54			54		
alu4-em4	8	505			505			505		
frisc-em4	23	409			409			409		
s298-em4	15	763			763			763		

Δ% is the percent improvement of your algorithm over the ASAP baseline:

$$\Delta\% \text{ improve} = \frac{ASAP\_PE\_Count - Your\_PE\_Count}{ASAP\_PE\_Count}$$

References

[1] Vaughn 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.

[2] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement, and routing tool for FPGA research. In Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner, editors, *Proceedings of the International Conference on Field-Programmable Logic and Applications*, number 1304 in LNCS, pages 213–222. Springer, August 1997.

[3] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, Massachusetts, 02061 USA, 1999.

[4] André DeHon. DPGA Utilization and Application. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 115–121, February 1996.

[5] Tom. R. Halfhill. Tabula's time machine. *Microprocessor Report*, March 29 2010.