Project Delta

Personal Report

Justus Matthiesen
(jm614@cam.ac.uk)

February 2009

1 Contribution to the Project

My main responsibility within the project was to provide the abstract/logical representation of digital circuits and to implement a simulation algorithm for these circuits.

My first task therefore was to investigate different algorithms and choose one which is simple enough to be implemented in the given time frame but yet has all the desired properties. It had been decided that the simulator must be capable of correctly simulating any circuit that can be built from logic gates (and a clock) and behaves deterministicly. Furthermore, it should not assume any starting conditions (e.g. all wires carry logic zero) or assign random boolean values to meta-stable outputs.

To deal with the latter condition, the decision was to use tri-state logic: The simulation extends the boolean logic values true (1) and false (0) by an additional logic state X to represent unknown values and values that cannot be inferred by boolean logic.
The former condition, however, turned out to limit the choice of algorithm quite severely. The condition implies that the simulator must be capable of simulating circuits containing loops and oscillating circuits. As there is no straight-forward way of detecting oscillating circuits, they can not be handled by the simulator as a special case. As figure 1 demonstrates, conventional boolean logic cannot

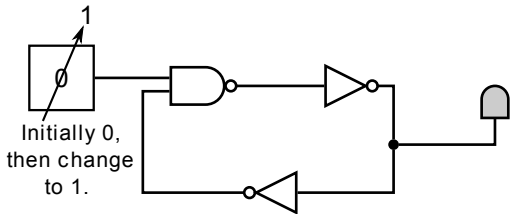

Figure 1: Ring of inverters.

model oscillating circuits and thus, the chosen algorithm must use non-zero gate delays. This ruled out several algorithms which had previously been considered: [2] describes a very memory efficient algorithm (which would have been a very good choice, if we had run the simulator on the FPGA) but it forbids loops in circuits and assumes zero gate delays. All other published algorithms I could find, like [3] or [4], that otherwise would have been suitable algorithms, suffer from similar constraints.

Consequently, the decision was to implement a simple algorithm that operates in discrete time steps and simulates on the gate level using unit-time gate delays. The idea behind the algorithm is, that we rarely need to re-evaluate the whole circuit if we store the simulation results of previous time steps. Thus, the simulator will only be called when the value carried by a wire connected to a gate input changes. In this case, the gate output is re-evaluated and if it has changed, the values of wires connecting to the gate output are updated and all gates connected to the output of this particular gate are scheduled (more specifically, added to a priority queue) for re-evaluation in the next time step [1].

[1] This is a fairly common appoach: A very similar algorithm can be found in [1].

A major disadvantage of this algorithm (or indeed any gate-level algorithm with non-zero gate delays) is that it is not possible to use a data structure with higher-order logic components that are simulated seperately. Hence, it was necessary to design a low-level circuit data structure that describes networks of gates.
From the gate-level view, logic circuits are essentially directed graphs where the vertex set is a set of gates and the edge set contains wire between gates (more specifically, from gate inputs to gate outputs). Exploiting this idea, the low-level circuit data structure has been implemented on top of a data structure representing directed graphs as provided by the well-designed graph library JGraphT[2].

However, this low-level data structure is not suitable for storing circuits that the user can build using the GUI, since such a circuit will contain higher-level components. I, therefore, had to design a second data structure based on the multi-graphs data structures provided by JGraphT[3].

Having written the simulator and data structures, I then went on implementing the various gates (AND, OR, NOT, etc.) and components (SR-Latch, D flip-flop, ...) and, together with Christopher, wrote the translation between the circuit representation the GUI uses and the data structures I have written.

Other tasks I did for the project were designing the logo, writing a build script and setting up a build server for continuous integration.

2 Evidence of Contribution

The source code of the project is openly available[4]. So I will simply refer to the location of the files that I have contributed (excluding the code Robert has written for the Verilog export feature): All files in the packages org.delta.circuit, org.delta.logic and org.delta.simulation have been written by me.

3 Contribution of other Group Members

Robert Duncan Robert wrote the interface to the hardware and the Verilog export. Consequently, he had to use most of my data structures and helped me to improve them by pointing out several issues and bugs that he found.

Christopher Wilson Christopher implemented the visual representation of the circuits. The quality of the code he wrote was of a high standard throughout the whole project.

David Weston David was responsible for the GUI framework.

[2]JGraphT. An open source Java graph library that provides mathematical graph-theory objects and algorithms, `http://www.jgrapht.org`.

[3]I suggested to use a library called JGraph (`http://www.jgraph.com`) for the visual representation and to use the (cleaner and more mathematical) JGraphT library for the data strucutures. Both libraries proved to be very useful in the course of the project.

[4]Hosted as a Google Code project: `http://code.google.com/p/delta09`.

Rubin Xu Rubin did take care of all the parts of the project which involved the FPGA board. He spent a lot of time on assesing the feasibilty of using the JOP project. Otherwise, I have little knowledge of what he did, as I was only using the interface Robert wrote which, from my perspective, just worked. So I can only assume that he did an excellent job.

References

[1] Harold Abelson, Gerald Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. McGraw-Hill Higher Education, 1996.

[2] Andrew W. Appel. Simulating digital circuits with one bit per wire. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 7(9):987–993, 1988.

[3] Peter M. Maurer. The inversion algorithm for digital simulation. In *ICCAD '94: Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 258–261, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[4] Peter M. Maurer. Logic simulation using networks of state machines. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 674–678, New York, NY, USA, 2000. ACM.