Wired - a Language for Describing Non-Functional Properties of Digital Circuits

Emil Axelsson, Koen Claessen and Mary Sheeran
Chalmers University of Technology

Abstract

Increasingly, designers need to estimate non-functional properties such as area, power consumption and timing, even when working at a high level of abstraction, early in the design. In deep sub-micron processes, it is the routing wires that account for most of the power consumption and signal delays. So, information about the wires is vital for controlling non-functional properties. To deal with more and more complex constructions, current design methods and languages strive towards higher and higher levels of abstraction, and provide only very limited possibilities for low-level control. Often, detailed information about wire properties is only available in the very last design stages - after placement and routing.

We propose a language, Wired, that aims to bridge this gap in abstraction levels. The main idea is construction with *combinators*, an approach previously used in Ruby and Lava [3, 1]. Regular circuits, such as arrays and trees, are described with generic higher-order *connection patterns* [1]. The key to the usefulness of this style is that the connection patterns have both *functional* and *geometric* interpretations. This allows us to construct circuits at high-level, without loosing control over lower levels. The descriptions have a recursive structure, with the leaves being primitive building blocks.

What distinguishes Wired from previous work is: (1) There is a clear distinction between circuits with geometry (hard circuits), and those without (soft circuits). The hard circuits have a more strict geometrical interpretation than in Lava. (2) Wires are circuits with size, and are not implicit. In order to take wire properties into account, wires shouldn't be treated as anything but normal circuits. (3) Circuits are 3-dimensional objects. This is necessary in order to describe circuits with several metal layers, or the 3-dimensional circuits that may soon appear [6]. However, in our first prototype, used to explore our ideas, we only model 2-dimensional circuits.

In 2D Wired, a hard circuit is a rectangle, whose four sides are called ports. Each port is a sequence of unit-size contact segments, where each segment can be either a connection or an insulator. Note that circuit (and wire) size is encoded in this way. The contact sequence is referred to as the *type* of the port, and the port can only be connected to other ports with the same type. This ensures that no unintentional connection/insulator mismatches are made. When two circuits are composed horizontally, using the *beside* combinator, for example, the top and bottom ports of the resulting circuit are structured into a pair of ports, giving them a type that can only be connected to other pairs. To convert between different types, a special thin circuit, called *port map*, is used. It can, however, not change the location of the contacts - only rearrange the structure of lists and tuples into which they are gathered. Having ports with structure makes it possible to describe connection patterns using recursion over port lists. This results in one type of generic circuits. We also have primitive wires with generic length. This genericity is also reflected in the port types of such a circuit, and we can use the type system to instantiate the generic circuits by placing them in a context of circuits with known types.

Soft circuits describe circuit function only. We are still debating whether to use relational or functional descriptions here, and at present we simply use ordinary Lava descriptions. A hard primitive is made by giving ports to a soft circuit (soft-to-hard conversion), and mapping the soft circuit's signals unto the connections in the ports. We think of *nailing down* the wires. A description of a hard circuit can also be converted back into a soft circuit (hard-to-soft conversion). This is useful when, for example, we want to connect two hard circuits and the wiring is too hard to describe. A construction could contain a mix of hard and soft parts, and we hope to have an interaction, where Wired takes care of the hard parts, and place-and-route tools implement the soft parts.

This is work in progress. We are currently studying the effect of different placement alternatives on wire length in a high-speed multiplier [2]. We plan also to tackle layout aware synthesis of arithmetic circuits (see [5, 4]). This will involve combining Wired with the idea of *clever circuits* [7].

Acknowledgement

This research is funded by the Semiconductor Research Corporation, in an Intel-custom research project called Expressing and Estimating Non-Functional Properties of Circuits (TASK ID 1041.001).

1. REFERENCES

[1] K. Claessen, M. Sheeran and S. Singh: The design and verification of a sorter core. Proceedings of the 11th Advanced Working Conference on Correct Hardware Design and Verification Methods, vol. 2144 of LNCS, Springer-Verlag, 2001.

[2] H. Eriksson, P. Larsson-Edefors, and W. P. Marnane: "A Regular Parallel Multiplier Which Utilizes Multiple Carry-Propagate Adders". Proceedings of IEEE International Symposium on Circuits and Systems, 2001.

[3] G. Jones and M. Sheeran: Circuit design in Ruby. In Formal Methods for VLSI Design, J. Staunstrup ed., North-Holland, 1990.

[4] Junhyung Um and Taewhan Kim: Layout-Aware Synthesis of Arithmetic Circuits. Proceedings Design Automation Conference, 2002.

[5] C. Martel, V. Oklobdzija, R. Ravi, and P. Stelling: Design strategies for optimal multiplier circuits. Proceedings 12th Symposium on Computer Arithmetic, 12:42–49, 1995.

[6] Matrix Semiconductor Inc., http://www.matrixsemi.com

[7] M. Sheeran: Finding regularity: describing and analysing circuits that are not quite regular. Proceedings 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, vol. 2860 of LNCS, Springer-Verlag, 2003.