Functional Programming in Hardware Design

Tomasz Wegrzanowski
Saarland University
Tomasz.Wegrzanowski@gmail.com

1 Introduction

According to the Moore’s law, hardware complexity grows exponentially, doubling every 18-24 months. While the 1993’s Intel Pentium processor had only 3.1 million transistors, the 2004’s Intel Itanium 2 has over 592 million, and the number is already getting into billions.

It does not mean that Itanium 2 is 190 times more complex than Pentium, as bigger caches and wider data paths are responsible for most of the transistor count increase, however without doubt the complexity of the ICs is growing very quickly and the hardware designers are trying to find some ways of managing that complexity.

Moore’s law is not the only reason why the hardware design is so difficult. Unlike the software, the hardware must be completely correct on the release date, as it is not possible to fix the bugs by publishing a patch later. The highly competitive environment does not permit any delays, or the hardware would be obsolete before even hitting the market. The problem is not limited to the few big firms producing CPUs, memories and other general-purpose chips. Hardware is also designed by a lot of smaller manufacturers, often in ASIC or FPGA technology.

Hardware design usually proceeds in a few well defined stages. The early stages are informal and performed with little computer assistance. The design begins as an idea, which is captured into a vague specification. Such specification is then converted to an abstract algorithm, that precisely defines the designed hardware’s functionality, but is not too specific about the implementation details. This algorithm is then refined to the word level, and then to the bit level.

The bit level algorithm usually requires little further human assistance, as the computer software is able to find a good layout of the chip, convert the bit operations to logical gates and connections between them, the gates to individual transistors, and then create the fabrication masks corresponding to those transistors and wires between them. The most time consuming part of the process are the stages where the design is precisely specified, and then refined to word and bit level.

The design serves more functions than being an input to the fabrication process. It is simulated, to explore and debug, it serves as a specification, it is also increasingly often used as an input to a theorem prover, for formal verification.
2 The mainstream of hardware design

The current mainstream of the hardware design are hardware description languages Verilog and VHDL. They are similar in expressive power, with the main difference being the syntax, with Verilog being based on C, and VHDL being based on Ada and slightly more verbose.

The languages were originally meant for precise specification and simulation of computer hardware. Because of that focus, they support various levels of abstraction, from very low level that deals with signal strengths and time bounds on gate delay, to a significantly higher level that treats word level operation like addition and multiplication as primitive. They also provide some support for analog and mixed-signal circuits.

Limited subsets of Verilog and VHDL are considered "synthesizable". If the design uses only synthesizable elements, it can be compiled to a form that is accepted by the hardware manufacturing process. Different technologies consider different constructs "synthesizable".

Verilog and VHDL are lacking in two aspects – they provide limited support for designing at a level higher than word transfer, and they have very complex semantics that makes formal verification difficult.

Many more powerful systems have been recently developed and some are presented here. Most of them operate on a higher level, describing only purely digital systems with a single clock, and for synthesis support compilation to either Verilog or VHDL.

3 Examples of functional hardware design

3.1 Lava

Lava is a Haskell library that exists in a few versions that significantly differ in their architecture, the 1998 version being based on monads, and the 2000 version on explicit circuit representation. There also exists a special version for Xilinx FPGA synthesis.

The 1998 version of Lava [1] is based on monads and type classes. The circuit is a function \( a \rightarrow m b \), where \( a \) are the input signals, \( b \) are the output signals, and \( m \) is an appropriate monad belonging to Circuit type class or one of its subclasses (Arithmetic, Sequential etc.). The circuits are composed from monadic operations and basic logic gates. Many convenient functions for composing circuits are provided and it is easy to write new ones. However, the Bit datatype is abstract, and while it is possible to set the circuit layout using any Haskell code, it is not possible to make arbitrary code operate inside the circuit.

The 2000 version [2] has completely different design. Instead of monads, Signal \( t \) family of types is used, which provides no way of inserting arbitrary types into it. Values of Signal \( t \) types can only be constructed using functions provided by the library. Internally, the signals are represented by abstract gates and references to other signals. This "impurity" is necessary to avoid recomputing the signal that is used multiple times, avoiding exponential or in case of
circular definitions even non-terminating behaviour (it is the so called “observable sharing” issue). Specifying the circuits is more convenient than in the 1998 version, as normal function composition can be used instead of monads. For comparison, here is a half adder circuit in Lava 1998 and Lava 2000.

-- Half adder in Lava 1998
halfAdd :: Circuit m => (Bit,Bit) -> m (Bit,Bit)
halfAdd (a,b)=
    do carry <- and2 (a,b)
       sum <- xor2 (a,b)
    return (carry,sum)

-- Half adder in Lava 2000
halfAdd :: (Signal Bool,Signal Bool) -> (Signal Bool,Signal Bool)
halfAdd(a,b)=(sum,carry)
where sum = xor2 (a,b)
      carry = and2 (a,b)

3.2 Hawk

Hawk [3] is attempting to solve a problem of high-level design of a modern superscalar microprocessor. Conceptually, a processor is executing instructions one at a time. Physically, it has not been the case for a very long time, and at any given moment of time different parts of the processor execute different instructions. The instructions can be executed in parallel, reordered, and even executed speculatively (the processor does not know yet whether their results will be committed or rejected). Many instructions can raise exceptions that break normal execution stream, invalidating not only further instructions but also those that are currently being executed or even those that have already finished and are just waiting for the commit. Instructions that interact with the external world (the memory, the bus, various devices) may also interfere with each other.

These effects cannot be dealt with separately, and it is the interaction between them what causes microprocessor design to take so much time and effort. Even the biggest CPU manufacturers make serious mistakes here. Both the Pentium f00f bug [4] and the Cyrix coma bug [5] were result of mishandled corner cases that allowed unprivileged code to hang the CPU or make it enter an infinite loop.

Hawk is a library built on top of Haskell that provides facilities for dealing with superscalar CPU design. The most important concept is a “transaction” that encapsulates all aspects of an instruction being executed.

The signals in Hawk are values of type Signal t. Unlike with both versions of Lava, in Hawk it is possible to apply any function to a signal by using lift :: (a -> b) -> (Signal a) -> (Signal b) function.

This design makes it possible to pass arbitrarily complex through the wires, including the aforementioned transactions. Of course, it is not possible to synthesize or automatically prove such circuits.
3.3 HDCamll

Haskell is not the only functional language used in hardware design. An OCaml library HDCamll [6] provides hardware design facilities roughly comparable to Lava 2000.

The architecture of HDCamll is very straightforward – all signals have type `signal`, and there are no type classes, monads, lazy lists or other features. Like in Lava 2000, signal is represented internally by abstract gates and references to other signals. This representation is then used for simulation, generation of Verilog code and verification. Type safety is reduced, as signals of all types have the same type.

4 Common issues

4.1 Deep vs. shallow embedding

The approaches fall into two categories. Hawk and to smaller extend Lava 1998 represent the so called "shallow embedding", where the objects of the embedded language are represented as analogous objects of the host language. For example an adder circuit in Hawk is just a lifted Haskell addition function. This approach allows for easily extending and easier cooperation with the rest of the host language and other libraries.

The other approach of so called "deep embedding", taken by Lava 2000 and HDCamll, represents the objects of the embedded language explicitly. It makes it more difficult to extend the circuits by new constructs, especially to use other libraries, but on the other hand it makes it much easier to code new circuit transformations (optimization, compilation to Verilog/VHDL) and analyzes (proving, timing computations etc.).

4.2 Observable sharing

A common theme in circuit libraries coded in Haskell is the "observable sharing" problem. The circuit is a small finite graph. It is natural to represent such graph as a set of nodes, each of them linked to others. Such graph contains a lot of sharing, and fully expanded form of it would be exponentially bigger or even infinite. Unfortunately Haskell does not provide us with means of taking advantage of the sharing. Computations on such graphs are as slow as computation of fully expanded graph, that is exponentially slow or even non-terminating.

The solution used by Lava 2000 is to make sharing observable by using some "unsafe" operations. If we can observe the sharing, it is possible to do computations on the circuit graphs much more efficiently.

This problem affects only libraries written in "deep embedding" style. In "shallow embedding" we can represent the signals as infinite streams, and Haskell will take full advantage of any sharing present, not computing the signals more than once.
References