An Embedded Language Approach to Teaching Hardware Compilation

Koen Claessen[1] and Gordon J. Pace[2]

[1] Chalmers University of Technology, Gothenburg, Sweden (koen@cs.chalmers.se)
[2] Vérimag Laboratory, Grenoble, France (Gordon.Pace@imag.fr)

Abstract. This paper describes a course in hardware description and synthesis (hardware compilation), taught as an introductory graduate course at Chalmers University of Technology, and as an advanced undergraduate course at the University of Malta. The functional programming language Haskell was used both to describe circuits and circuit synthesis schemes.

1 Introduction

There are two essentially different ways of describing hardware. One way is *structural* description, where the designer indicates what components should be used and how they should be connected. Designing hardware at the structural level can be rather tedious and time consuming. Sometimes, one affords to exchange speed or size of a circuit for the ability to design a circuit by describing its behaviour at a higher level of abstraction which can then be automatically *compiled* down to structural hardware. This way of describing circuits is usually called *synthesisable behavioural description.* Examples of behavioural languages that can be synthesised are Esterel [1], Occam [7], SAFL [8], and even the traditional hardware design languages such as Verilog and VHDL include a behavioural description language.

Teaching hardware synthesis techniques to students can be quite problematic. As in teaching compilation techniques, one has to choose between one of two avenues: either plough through the theory and techniques, hoping the students are sufficiently mature and motivated to explore the ideas, and risking that a number of fine points of the synthesis procedures are lost since the students would have no means of trying variations (and see why they do not work so well), or implement a simple synthesis tool as the course goes on which allows students to experiment with alternative solutions, add new constructs, etc. The latter is obviously desirable, but leads to a lot of lecture time dedicated to going through the implementation, and a lot of student time to understand the fine points in the implementation to be able to change, and enrich it.

However, using Haskell to write our compiler, together with Lava [3] to describe (and be able to simulate and verify) hardware, we found that the resulting code was short, easily understandable and compositional. Some of the features found in most modern functional programming languages (in particular pattern matching and higher-order functions) enabled us to teach the course using a real implementation, while at the same time not wasting much time on the actual implementation itself.

Functional programming has already been established as a useful teaching aid for teaching structural hardware design and structural synchronous hardware simulation [3, 6, 10, 9]. When performing structural hardware design, it can be tedious to build up components by reducing the problem structurally into a number of smaller problems. The main advantage gained when having a hardware design language as part of a programming language is that the programming language can be used as a *meta-language* for generating large, 'regular' circuits. This process is also called *structural synthesis*.

Standard hardware description languages usually have a limited meta-language used to build circuits (eg `for...generate` in VHDL), but such meta-languages tend to be very limited (eg `for...generate` can only generate linear lists of components). The main difference with embedded hardware description languages is that in principle any circuit can be generated, and parameters to circuit descriptions can be of any complexity.

What is most important in courses that teach hardware synthesis techniques is to get across how code is compiled – different techniques for optimisation and other compilation issues. The need to go down to low-level HDL code now becomes unnecessary – in fact it is more of a hindrance than a help. We thus found this to be an ideal topic to use a functional structural hardware description language to teach. Small behavioural languages can be embedded in the functional language and synthesis procedures can be described as generic circuits which are parameterised by behavioural programs! Thanks to high-level features of modern functional programming languages, description of compilation procedures are short and very comprehensible, perfect to get the concepts behind synthesis across.

2 Course Description

A one week intensive course, consisting of approximately 20 hours (including practical sessions) was given at Chalmers University of Technology. The course was primarily aimed at PhD students and advanced undergraduates. The structure of the course was: two, two hour long taught sessions per day, followed by aided practicals, where the students had to solve a number of set problems. The course attracted approximately 15–20 students. The course was also given at the University of Malta, in a reduced form, mainly aimed at advanced undergraduates, and without the practical sessions, where it attracted approximately 15 students. It was assumed that students had a strong foundation in computer science. The course could be seen as being split into five main parts:

Synchronous circuits: The course started with a brief introduction as to what synchronous ciruits are. It was assumed that the students would have seen most of this before, so it was possible to swiftly cover topics ranging from transistor circuits, gate level design, finite state machine implementation and arithmetic circuits.

Verilog: Up to this point, circuits were represented in graphical form. An overview of Verilog was given to show how circuits can be represented (structurally) in textual form, and behavioural specifications may be written for test purposes. This part finished with a first glimpse of synthesis: how certain behavioural descriptions may be converted into finite state machines and then converted into circuits, via a few simple examples.

Lava: This is where functional programming kicked in. Lava[3] is a structural hardware description language embedded in Haskell and was used to give a metalanguage to design complex hardware systems.

Hardware Synthesis: This was the main culmination of the course. A simple imperative style language was embedded in Haskell and we proceeded to write a compiler which produces Lava circuits from these programs automatically. More complex features were added to the language, and some issues of optimising the circuits were discussed.

Combining Languages: The ease by which different languages can be compiled into hardware leads to the natural question — Can one have a framework in which different parts of the circuits can be written in different languages in a straightforward, yet safe manner? This session closed the course by presenting our then ongoing research into this problem [4].

Everyday, the students had to solve a number of problems in circuit design. The problems depended on the topic covered during the day, but one of the tasks, to design a hardware stack, was repeated for each of the different design techniques presented, so that students could compare the advantages, and drawbacks of each approach.

3 Lava

Lava is a structural hardware description language embedded in the functional language Haskell [2]. The embedding is implemented as a hardware description library that consists of two parts. The first part of the library provides a number of primitive functions that can be used to build circuits. Examples of these are logical gates (`and2`, `or2`, `xor2`, etc.), registers and other state holding elements (in this paper, we will use the implicitly clocked `delay` element), and circuit combinators (serial and parallel composition, `row` and `column`).

Here is an example of a simple circuit that uses one xor gate and one delay element to implement a toggle. The delay element is initialised with the value `low`, and after that, at every point in time, produces the input from the previous point in time as output.

```
toggle :: Signal Bool -> Signal Bool
toggle inp = out
  where out' = delay low out
        out  = xor2 (inp, out')
```

The second part of the Lava library provides functions that can process circuit descriptions. Examples of these are circuit simulation (where the circuit is run on provided inputs), generation of VHDL code (where we use symbolic simulation to get a concrete representation of the circuit), and generation of input to circuit verification tools and theorem provers (again, we use symbolic simulation for this).

Here is an example of how we can use Haskell as a meta-language to describe the structure of a family of circuits. The following Lava code produces a tree structure to compute the disjunction of n-inputs ($n > 0$) with only depth $\log(n)$ combinational depth (the number of gates in the longest path between inputs and outputs) of 2-input disjunction gates:

```
orTree :: [Signal Bool] -> Signal Bool
orTree [input] = input
orTree inputs  = or2 (orTree firstHalf, orTree secondHalf)
  where (firstHalf, secondHalf) = splitAt (length inputs `div` 2) inputs
```

However, more abstract features of modern functional programming languages make this approach even more powerful. The following example shows how the binary tree structure used in the previous example can be generalised to work for any 2-input, 1-output circuit thanks to higher order functions:

```
tree :: ((a,a) -> a) -> [a] -> a
tree circ [input] = input
tree circ inputs  = circ (tree circ firstHalf, tree circ secondHalf)
  where (firstHalf, secondHalf) = splitAt (length inputs `div` 2) inputs
```

Now, `orTree` simply becomes an instance of this generic circuit generator, namely `tree or2`. This is but one instance of circuit combinators, of which one may build a sufficient library to be able to elegantly describe complex circuits (see e.g. [5]). Note that in the process, the circuit combinator has become polymorphic in its input type, which means it can be used for many more purposes than just on bits.

4 Teaching Hardware Compilation

To illustrate hardware compilation concepts, we chose to take a small imperative language (with parallel composition) and to show how it can be compiled into hardware. The language, we called MiniFlash borrows much from hardware synthesis languages such as Esterel [1], Handel [11] and, synthesisable Verilog and VHDL. We start by embedding MiniFlash syntax as an abstract datatype in Haskell.

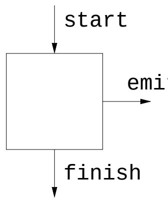
```
data MiniFlash = Skip
               | Delay
               | Emit
               | MiniFlash :>> MiniFlash      -- sequential composition
               | IfThenElse (Signal Bool) (MiniFlash, MiniFlash)
               | While (Signal Bool) MiniFlash
               | MiniFlash :|| MiniFlash      -- fork ... join
```

The language has been kept purposefully simple. A program can only output one boolean signal. This is `low` unless at least one of the running parallel programs performs an `Emit` instruction during the current clock cycle. This kind of 'absent unless specified' approach has been used by Esterel. `Skip` and `Emit` do not take up clock cycles to execute, `Delay` blocks the program for one clock cycle. Sequential composition, conditionals and loops behave as usual, while the parallel composition has a fork-join semantics — both programs are started at the same time, and the block terminates when both branches have terminated.

With this basic embedding we can construct a number of program constructors to ease specification:

```
forever p = While high p
wait s    = While (inv s) Delay
```

The first question is what kind of circuit we shall obtain by compiling such a program. Since, unlike a MiniFlash program, a circuit has no notion of starting and finishing, we need to add this as information to the circuit produced. We will have an input wire which is set to high when we want to start a program, and an output wire which will be set to high when the program terminates. We also have to tell the environment what value is output by the program, resulting in a circuit with the shape adjacent.




```
type Circuit = Signal Bool -> (Signal Bool, Signal Bool)
```

The compilation task would thus be a function of type:

```
compile :: MiniFlash -> Circuit
```

The compiler can be specified as a recursive function using pattern matching to separate the different operators. As an example of a base case, compiling `Delay` is straightforward — it takes one whole cycle to terminate, and always outputs `low`:

```
compile Delay start = (emit, finish)
  where finish = delay low start -- we finish after one clock cycle
        emit   = low             -- no output
```

As an example of a recursive compilation step, the case for the conditional instruction looks as follows:

```
compile (IfThenElse cond (prog1, prog2)) start = (emit, finish)
  where (emit1, finish1) = compile prog1 start1
        (emit2, finish2) = compile prog2 start2
        start1           = and2 (start, cond)
        start2           = and2 (start, inv cond)
        emit             = or2 (emit1, emit2)
        finish           = or2 (finish1, finish2)
```

Both branches are computed by preparing the corresponding start wires, and calculating the circuits recursively. We emit if one of the branches emits, and we finish when one of the branches finishes. This is correct since we know that only one branch is running at the same time. The Lava code to compile the rest of the language is given in appendix A.

Once the compiler is implemented, one can easily test it out using the simulation facilities in Lava, or by exporting to VHDL and using more sophisticated simulators.

```
risingEdge s = forever (wait (inv s) :>> wait s :>> Emit)

risingEdgeCircuit s = emit
  where start          = delay high low
        (emit, finish) = compile (risingEdge s) start

Lava> simulateCon risingEdgeCircuit [high, low, high, high]
[low, low, high, low]
```

Using this approach, the students found it easy to follow not just the circuit diagram, but also its implementation. The implementation allowed them to experiment by simulating programs, to understand obscure parts of the compilation and explore alternative definitions and the implementation of other constructs. We feel that this led to a better appreciation of the issues involved behind hardware synthesis.

5 Processing Compiled Programs

One of the benefits we get from using an embedded language is that after this compilation definition we can immediately simulate and verify programs. Here is an example of a simple Flash program. Note how we can use Haskell function declarations for describing procedures, i.e. this program can be used as a subprogram anywhere else.

```
unordered (a,b) = compile ((wait a :|| wait b) :>> emit) start0
  where start0 = delay high low
```

We can immediately simulate the circuit on inputs:

```
Lava> simulate unordered [(low,high),(low,low),(high,high)]
[(low,low),(low,low),(low,high)]
```

Also, we can define a property of the compiled program:

```
property (a,b) = emit <==> finish
  where (emit, finish) = unordered (a, b)
```

And check that it always holds using external model-checkers from within the Lava environment:

```
Lava> verify property
Proving: ... Valid.
```

A similar approach of specifying properties can be used to specify properties about the compilation process, to for example check that parallel composition is commutative.

6 Exercises

Exercises given in a new course serve, not only to judge the students, but also to judge how successful the course was in conveying ideas across to the students.

At the end of every day of lectures, the students were required to solve a number of hardware design problems. A recurring exercise was that of constructing a circuit implementing a stack, and supporting three operations: push, pop and top.

At the end of the first day, the stack was to be designed using block diagrams, and was thus, for simplicity a 4-place, 1-bit stack. Students were encouraged to find a compositional design to simplify the design. At the end of the second day, the stack was to be implemented in Verilog and tested on a simulator. Students were also asked to explore what changes would need to be made to have a bigger stack size, or wider data path. Finally, after the Lava lectures, the students were to reimplement the stack in Lava, and generalise be able to store other datatypes, and parametrise the stack description by its depth. The main motivation was to show how useful it can be to be able to generically describe hardware. The advanced students also had to verify a number of properties about their stack implementation. At first, the main problem faced by the students was that of finding a modular design to their implementation. However, by the end of the week, we found students were improving on their original design, making it more modular, and amenable to verification (for the students who were also doing the verification exercises).

As for the synthesis exercises, we gave two problems to solve. Students who were not sufficiently familiar with Haskell before the course, were to design (using pen and paper) a compilation scheme for MiniFlash augmented with an exception mechanism.

```
  data MiniFlash = ... | Throw | Catch MiniFlash
```

All managed to come up with a satisfactory solution, and we were also pleased to note that a number of them also attempted, and succeeded, in implementing their solution in Lava. We feel that this indicates that the choice of enriching the course with an actual implementation of synthesis written in Lava was a good choice.

Students familiar with Haskell were asked to implement assignment variables in the language. The exercise was split into three parts: implementing one assignment variable (assignment statements taking one clock cycle to perform), multiple assignment variables, and finally, for the most adventurous, add assignment statements which take 'no time' to perform.

```
data MiniFlash = ...
               | Declare (Variable -> Miniflash)
```

```
| Variable := Signal Bool
| Read Variable (Signal Bool -> MiniFlash)
```

Again, all students solved the problem, although only two tried and managed to (partially) solve the last exercise. Since it was meant to be a very hard challenge, we were pleased that two students actually managed to find a partial solution.

7 Experience and Conclusions

We feel that we managed to teach quite advanced concepts in both hardware compilation and functional programming to students who had no background in hardware design or in functional programming!

From a hardware point of view, the embedded functional language approach gave us a quick and easy way to describe compilation processes precisely and get an 'implementation for free'. The resulting descriptions were easy to understand and easily extended by the students.

From a functional programming point of view, we managed to convincingly show that embedded languages are a good idea. Also, we quite naturally introduced concepts such as polymorphism and higher-order functions, which are both motivated by the desire for an economical description of the hardware design or compilation process.

References

1. Gérard Berry. The Esterel primer. Available from `http://www.esterel.org`, 2000.
2. K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, Phuket, Thailand, 1999. ACM Sigplan.
3. K. Claessen and M. Sheeran. A tutorial on Lava: A hardware description and verification system. Available from `http://www.cs.chalmers.se/~koen/Lava`, 2000.
4. Koen Claessen and Gordon Pace. An embedded language framework for hardware compilation. In *DCC'02*, ETAPS, Grenoble, France, 2002.
5. Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *CHARME*. Springer, 2001.
6. Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems*. Marstrand, Sweden, 1998.
7. David May. Compiling Occam into silicon. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, University of Texas at Austin Year of Programming Series, chapter 3, pages 87–106. Addison-Wesley, 1990.
8. Alan Mycroft and Richard Sharp. A statically allocated parallel functional language. In *Automata, Languages and Programming*, pages 37–48, 2000.
9. J. O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming Glasgow*, Springer-Verlag Workshops in Computing, pages 178–194, 1993.
10. J. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Functional Programming Languagues in Education*, volume 1125 of *Lecture Notes In Computer Science*, pages 221–234. Springer Verlag, 1996.

11. Ian Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.
12. Ian Page and Wayne Luk. Compiling Occam into field-programmable gate arrays. In Wayne Luk and Will Moore, editors, *FPGAs*, pages 271–283. Abingdon EE&CS books, 1991.
13. Niklaus Wirth. Hardware compilation: Translating programs into circuits. *Computer*, 31(6):25–31, 1998.

A Compiling MiniFlash

The purpose of this appendix is to give a complete definition of the compilation of MiniFlash. Although the examples given in the paper should suffice to convey the idea behind how hardware compilation was presented in the course of the lectures, this section completes the missing definitions there.

Programs in MiniFlash are represented as object of type `MiniFlash`.

```
data MiniFlash =
        Skip
      | Delay
      | Emit
      | MiniFlash :>> MiniFlash      -- sequential composition
      | IfThenElse (Signal Bool) (MiniFlash, MiniFlash)
      | While (Signal Bool) MiniFlash
      | MiniFlash :|| MiniFlash      -- fork ... join
```

Upon compilation, we will produce a circuit, which takes an input signal (which triggers off the program), and produces two output signals – the output of the program, and a signal indicating whether the program has just terminated.

```
type Circuit = Signal Bool -> (Signal Bool, Signal Bool)
```

The compilation task is thus a function of type:

```
compile :: MiniFlash -> Circuit
```

The compiler can be easily written using pattern matching over the datatype `MiniFlash`.

Compiling `Skip` and `Emit` is straightforward:

```
compile Skip start = (emit, finish)
  where
    finish = start -- we finish as soon as we start
    emit   = low   -- we never push the output high

compile Emit start = (emit, finish)
  where
    finish = start -- we finish as soon as we start
    emit   = start -- we push the output high when started
```

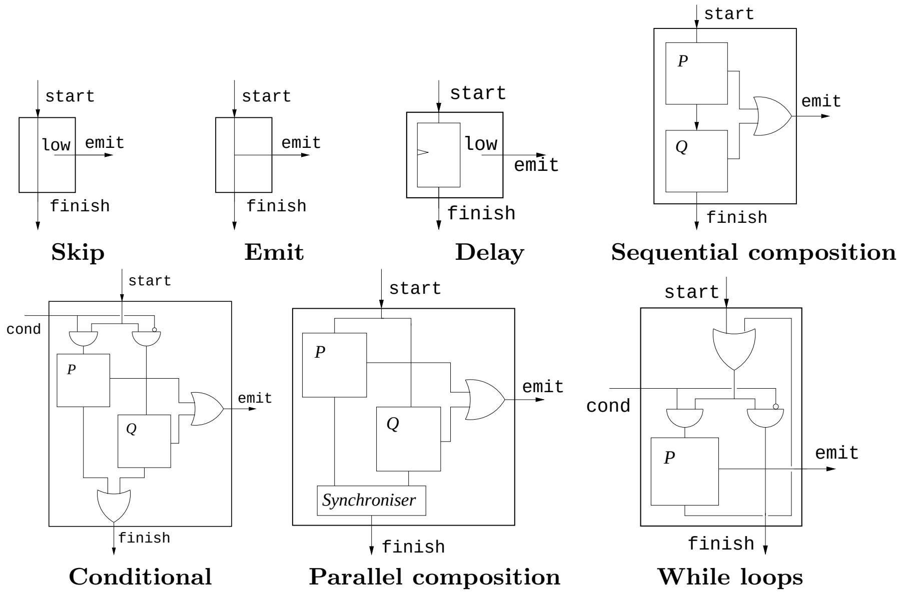


Fig. 1. Compiling MiniFlash

Compiling `Delay` is identical to `Skip`, except that the instruction takes one whole cycle to terminate:

```
compile Delay start = (emit, finish)
  where
    finish = delay low start -- we finish one clock cycle after the start
    emit   = low   -- we never push the output high
```

The compilation shemata of the compound constructs, can be best understood by studying the circuit diagrams in figure 1. The Haskell/Lava code is a direct translation of the figures.

Consider sequential composition:

```
flash (prog1 :>> prog2) start = (emit, finish)
  where
    (emit1, middle) = compile prog1 start
    (emit2, finish) = compile prog2 middle
    emit = or2(emit1, emit2)
```

The case for the conditional instruction looks as follows:

```
compile (IfThenElse cond (prog1, prog2)) start = (emit, finish)
  where
    (emit1, finish1) = compile prog1 start1
    (emit2, finish2) = compile prog2 start2
```

```
    start1 = and2 (start, cond)
    start2 = and2 (start, inv cond)

    emit  = or2 (emit1, emit2)
    finish = or2 (finish1, finish2)
```

Both branches are computed by preparing the corresponding start wires, and calculating the circuits recursively. We emit if one of the branches emits, and we finish when one of the branches finishes. This is correct since we know that only one branch is running at the same time.

The case for the while loop is one of the more interesting cases, due to the feedback loop passing the finish wire of the subcircuit back to the start wire (via a disjunction gate).

```
compile (While cond prog) start = (emit, finish)
  where
    (emit, finish') = compile prog start'
    restart = or2 (start, finish')
    start'  = and2 (restart, cond)
    finish  = and2 (restart, inv cond)
```

We might (re)start the body of the while loop, if the whole loop is started or if the body has just finished. In that case, depending on the condition, we restart the body or we finish. Note that we have created a loop since `finish'` depends on `start'` depends on `restart` depends on `finish'`. In fact, the circuit will only work correctly if the subprogram takes time to execute.

Parallel composition is another interesting case. At face value, it is similar to the conditional statement except that (i) we start both branches at once, and (ii) the program finishes as soon as *both* subprograms have terminated.

```
compile (prog1 :|| prog2) start = (emit, finish)
  where
    (emit1, finish1) = compile prog1 start
    (emit2, finish2) = compile prog2 start
    emit  = or2 (emit1, emit2)
    finish = synchroniser (finish1, finish2)
```

We start both processes as soon as the parallel composition is started. We emit when one of the processes emits. But when do we finish? We use a little circuit, called *synchroniser*, which keeps track of both processes, and generates a high on the finish signal exactly when both processes have finished.

```
synchroniser (finish1, finish2) = finish
  where
    both   = and2 (finish1, finish2)
    one    = xor2 (finish1, finish2)
    wait   = delay low (xor2 (one, wait))
    finish = or2 (both, and2 (wait, one))
```

The wire `both` is high when both processes are finishing at the same time. The wire `one` is high when exactly one process is finishing. The wire `wait` is high when one process has finished but not the other.

Various paper can be found in the literature with an in depth study of the compilation of high-level imperative-style languages. See eg [12, 13, 1].