A mathematical approach towards hardware design

Gerard J.M. Smit, Jan Kuper, Christiaan P.R. Baaij

University of Twente, Enschede, The Netherlands,
G.J.M.Smit@ewi.utwente.nl

Abstract. The inadequacies of traditional design practices for embedded systems have led to a myriad proposals as to improve the state of the art and raise the abstraction level. Almost all of these proposals still start with an imperative "C"-like basis, even though the original *mathematical* specification has completely different semantics. We show a design process based on a functional language, staying in the same semantic domain as the mathematical specification. Meaning-preserving transformations on these functional description then lead to an optimized parallel design.

Keywords: Hardware Design, Functional Languages

1 What is wrong with hardware design today

Hardware description languages and design practices have not allowed the productivity of hardware engineers to keep pace with the development of chip technology. Systems described in traditional design languages such as VHDL are rather low level, which is highly cumbersome and may lead to design faults in large real-life applications. The problem with traditional design practices, especially in the (streaming) digital signal processing domain, is that semantic domains are changed *twice*, almost guaranteeing excessive amounts of verification. The algorithms for signal processing are often written down as a mathematical specification. When turning such an algorithm into a streaming application (on hardware) the traditional approach is to first translate the specification to sequential "C"-like code. As the semantics of the "C"-code do not match the original mathematical specification, extensive testing and verification has to be performed; often requiring the effort of multiple persons. Once verified, this "C"-code is then manually translated by another group of people to (parallel) RTL-style VHDL code. As the semantics of the VHDL and "C"-code do not match, the translation is cumbersome and potentially leads to design faults. As a result excessive and time-consuming verification is once again required.

2 Functional, transformational design

Our approach does not suffer from the faults incurred by the semantic domain crossing of the traditional approaches: we straightforwardly convert the original mathematical specification to a program written in the functional programming language Haskell [8]. The translation is straightforward as the semantics of the mathematical specification and Haskell match. As functions only specify true data-dependencies, full parallelism remains exposed (as opposed to “C”). The next step is to find some adequate (meaning-preserving) transformations on the specification, in particular to find specific optimizations. These transformations are written in a mathematical format, allowing us to prove the correctness of these transformations.

Next, the resulting Haskell specification is given to a compiler, called CλaSH[1] [3,9], that translates[2] the specification into VHDL. The resulting VHDL is synthesizable, so from there on standard VHDL-tooling can be used for synthesis. We remark that the choice for VHDL is of a practical nature and motivated by the availability of synthesis tools for VHDL.

Specifications written in Haskell are clear and concise. Furthermore, it is possible to use powerful abstraction mechanisms (which are not available in VHDL, or C) such as polymorphism, higher-order functions, pattern matching and partial application. These features allow a designer to describe (parameterizable) designs in a more natural and concise way than possible in traditional languages.

3 Functions and Hardware

Two basic elements of a functional program are functions and function application. These have a single obvious translation to a netlist format: 1. every function is translated to a component, 2. every function argument is translated to an input port, 3. the result value of a function is translated to an output port, and 4. function applications are translated to component instantiations. The result value can have a composite type (such as a tuple), so the fact that a function has just a single result value does not pose any limitation. The actual arguments of a function application are assigned to signals, which are then mapped to the corresponding input ports of the component. The output port of the function is also mapped to a signal, which is used as the result of the application itself. Since every function generates its own component, the hierarchy of function calls is reflected in the final netlist.

The short example below (1) gives a demonstration of the conciseness that can be achieved with CλaSH when compared to other (more traditional) HDLs. The example is a combinational multiply-accumulate circuit that works completely polymorphic, i.e., the type of the variables is not yet specified. The corresponding netlist is depicted in Figure 1.

```
mac x y acc = add (mul x y) acc                                    (1)
```

[1] CλaSH: CAES Language for Synchronous Hardware

[2] As the CλaSH compiler is still a prototype, only a subset of Haskell is supported, meaning the specification might need some modifications.

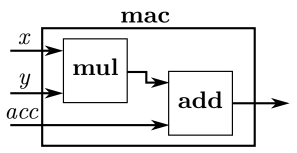

Fig. 1. Combinational Multiply-Accumulate

Thus, the function `mac` has three arguments `x`, `y`, and `acc`. The result is calculated by first applying the function `mul` to `x` and `y`, and then applying the function `add` to the result of the multiplication and `acc`.

Since (most likely) `mul` and `add` are numerical functions, the parameters are restricted to numbers, though the exact number type is still unspecified. We can create a concrete/monomorphic instance of this multiply-accumulate circuit by making a new function that is annotated with a concrete type (2), the inferred hardware of which is shown in Figure 2:

```
type Word = Signed D16
mac16 :: Word -> Word -> Word -> Word                    (2)
mac16 = mac
```

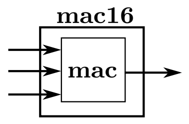

Fig. 2. 16-bit signed integer Combinational Multiply-Accumulate

Here, the type `Word` is defined as an alias for the type `Signed D16`, which indicates a signed integer type of 16 bits wide. The input of the function `mac16` now consists of three values of the type `Word` (the first three occurrences of "`Word`" in the type of `mac16`), and the result of `mac16` (the last occurrence of "`Word`" in that type expression) also is of type `Word`.

We could already have annotated the original *mac*-function with a concrete type, but defining a new function allows reuse of mac for a different number types and bit-widths. Notice that the inputs of the *mac16* function are implicitly passed to the mac function (i.e. an implicit port mapping).

To work with the concept of state in a hardware description, we make the current state an extra argument of the *hardware description function*, and the updated state an extra part of the result of that function. Such a function thus resembles the well known Mealy-machine. To indicate which argument of the function denotes the state we annotate it with the *State* keyword. In hardware these state arguments are assumed to *come from* registers, whereas the state part of the result of a function is *put into* registers. As an example we return to the multiply-accumulate circuit, and store the accumulator in a register as shown in

Figure 3. The value in this register is represented by the parameter *State s* in definition (3), whereas the updated content of the register is represented by the output *State s'*:

```
macS (State s) (x,y) = (State s',s')
  where                                                  (3)
    s' = mac16 x y s
```

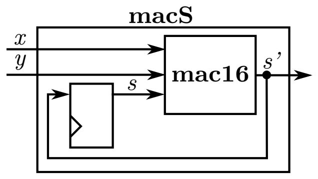

Fig. 3. 16-bit signed integer Multiply-Accumulate, with state

4 Example: FIR-filter

As an example of our transformational design method, we make a simple FIR-filter. The mathematical specification of a FIR-filter is:

$$y_t = \sum\nolimits_{i=0}^{n-1} x_{t-i} \cdot h_i \tag{4}$$

In other words, the FIR-filter takes the vector dot-product of the coefficients **h** and an equally long vector of consecutive samples of the stream **x**.

4.1 Ubiquitous functions

Before we translate the mathematical specification into Haskell, we will first elaborate on three ubiquitous functions in the functional programming world: *map*, *zipWith*, and *fold*.

Map The *map* function applies a function f on every element in a list xs, e.g.

```
map square [1,2,3] = [1,4,9]
```

In general, given a function f from type a to type b and a list xs of elements of type a, the expression

```
map f xs
```

will result in a list of values of type b. That is to say, the type of *map* is:

```
map :: (a -> b) -> [a] -> [b]
```

The respective netlist interpretation of the *map* function is depicted in Figure 4.

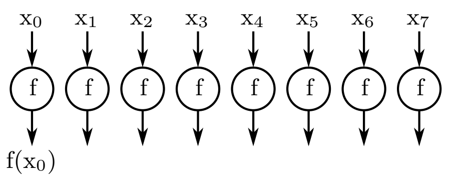

Fig. 4. Netlist interpretation of the map function

ZipWith Likewise, the *zipWith* function applies a binary function *f* pairwise on the elements of two lists, *xs* and *ys*, e.g.:

```
zipWith add [1,2,3] [4,5,6] = [5,7,9]
```

Thus, given a function from *a* and *b* to *c* and two lists of *a*'s and *b*'s,

```
zipWith f xs ys
```

will result in a list of *c*'s. The type of *zipWith* is:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

The netlist interpretation of the *zipWith* function is shown in Figure 5.

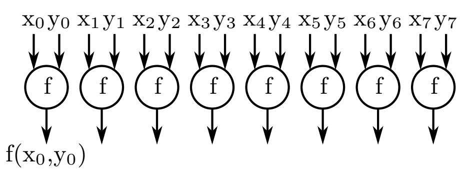

Fig. 5. Netlist interpretation of the zipWith function

Fold Finally, the *fold* function applies a binary function *f* on the elements of the list *xs* to reduce it to a single value, using the value *z* to start the reduction, e.g. (using "+" for *add*):

```
fold (+) 0 [1,2,3] = 6
```

Thus, given a function *f* from *a* and *b* to *a*, a start value of type *a*, and a list of *b*'s,

```
fold f z xs
```

will return a value of type *a*. The type of *fold* is:

```
fold :: (a -> b -> a) -> a -> [b] -> a
```

The respective netlist interpretation of the *fold* function is depicted in Figure 6.

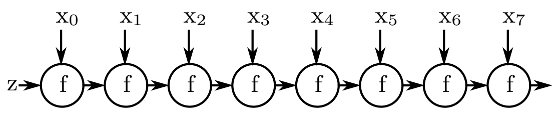

Fig. 6. Netlist interpretation of the fold function

4.2 Design

Returning to the example of the FIR-filter, we can now straightforwardly define the dot-product of two vectors by using two of the above functions, *fold* and *zipWith*:

```
dotp xs hs = fold (+) 0 (zipWith (*) xs hs)                        (5)
```

That is to say, the **x**-values are pairwise multiplied with the **h**-values, and the resulting values are added, starting with the value 0.

There is one aspect that has to be added to this definition to transform it from a dot-product to a FIR-filter: to keep a state; i.e., to update the contents of the **x**-registers and **h**-registers. To do so we add *State* keywords to the function definition:

```
fir (State (xs,hs)) x = (State (x : init xs, hs), y)
  where                                                            (6)
    y = fold (+) 0 (zipWith (*) xs hs)
```

As can be seen in this definition, the state consists of two parts: the list *xs* (i.e., the **x**-registers) and the list *hs* (i.e., the **h**-registers). The new contents of the **x**-part of the state is the new input x in front of the *init*-part of the previous contents *xs*, where the *init*-part of a list is the whole list except the last element. Effectively this means that the new input x is put into the first register, whereas the previous contents is shifted one position to the right. As before with the multiply-accumulate examples, when instantiated for vectors of length four, this definition immediately corresponds to the architecture in Figure 7.

Note that in definition (6) the **h**-part of the state remains unchanged. This offers the possibility to extract the **h**-parameters of the FIR-filter from the description of the hardware architecture and to consider them as parameters for the "pattern" of a FIR-filter in general. Consider the following alternative definition:

```
fir' hs (State xs) x = (State (x : init xs), y)
  where                                                            (7)
    y = dotp xs hs
```

Here, *fir'* is a function that yields a FIR-filter for every sequence of coefficients *hs*. Such a FIR-filter can be denoted by *fir' hs* for a given sequence *hs*.

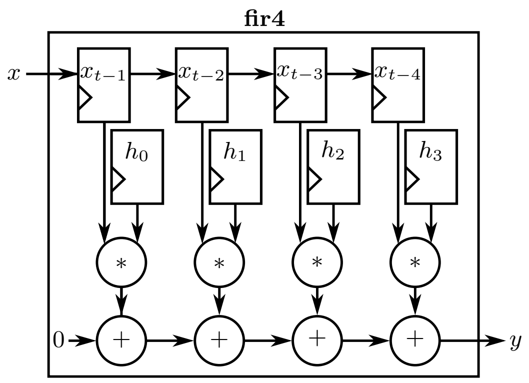

Fig. 7. Four-taps FIR-filter

5 Simulation

An advantage of our approach is that every architecture specification can be simulated immediately by a Haskell interpreter using the following function:

```
simulate f (State s) (x:xs) = y : simulate f (State s') xs
  where                                                        (8)
    (State s', y) = f (State s) x
```

In this definition *x:xs* is a sequence of inputs, of which x is the first input, and *xs* the remaining sequence of inputs. Thus, the function *simulate* takes the first input x and the present state *State s* and applies a given architecture f to them (in the *where* clause). Next, the function *simulate* outputs the value y and continues with the new state *State s'* and the remaining sequence of inputs *xs'*.

Given the above examples, the following expressions represent simulations of the according architectures:

```
simulate macS (State 0) [(1,1), (2,2), (3,3)]

simulate fir  (State (zeroes,hs)) inputs                       (9)

simulate (fir' hs) (State zeroes) inputs
```

In the above it is assumed that *zeroes* is a list of zeroes of adequate length, *hs* is a given sequence of coefficients, and *inputs* is a given sequence of input numbers. Note that the function simulate is polymorphic and works for any architecture.

6 Parallelization

Automatic parallelization of given C-code is an extensive topic of research. However, as yet there is no method that is able to split any given C-program in parallel threads, all methods presuppose severe restrictions on the given program [4,15]. When starting from a mathematical specification, or, as indicated above, from an equivalent Haskell specification, the situation is different. First of all, a functional specification is inherently parallel since such a specification is *function* based and not *statement* based. That means that it is possible to check a given program for specific mathematical properties in order to apply transformational rules to parallelize a given piece of code. For example, consider the following specification of the *fold* function (*op* is a binary operation, *a* is the starting value, and *xs* is a list of values):

```
fold op a xs                                                    (10)
```

As seen above, this expression reduces the list *xs* by applying the operation *op*, and starting from the value *a*. Clearly, when *op* is associative, and *a* is the unit element of *op*, then one may split the list *xs* in sub-lists, reduce each sublist separately and reduce the list of results. This leads to the following transformation rule:

Suppose *xss* is a list of lists which together form the list *xs*, i.e., the concatenation of all elements of *xss* forms *xs*. Then we have the following equality (see also Figure 8):

```
fold op a xss = fold op a (map (fold op a) xss)                 (11)
```

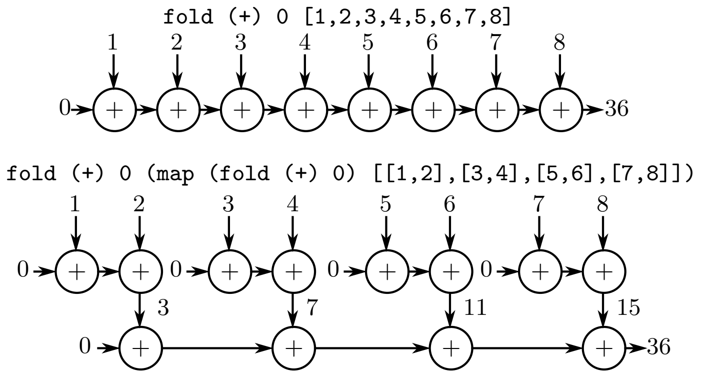
fold (+) 0 [1,2,3,4,5,6,7,8]
1
2
3
4
5
6
7
8
0
+
36
fold (+) 0 (map (fold (+) 0) [[1,2],[3,4],[5,6],[7,8]])
3
7
11
15
36

Fig. 8. Equivalence of fold and fold-map-fold

Note that the obligation of a designer is a well isolated proof obligation: prove that *op* is associative, and that *a* is the unit element of *op*. In a similar way, but using a different transformation, we show that is also possible to split up e.g. a FIR-filter:

```
fir'' (us,(x,z)) (hs,xs) = (us ++ (x : init xs), (last xs, y))
  where
    y = fold (+) z (zipWith (*) xs hs)
                                                                   (12)
firSplit n hs xs x = fold fir'' ([],(x,0)) xs
  where
    xs = zip (split n hs) (split n xs)
```

We apply a structural transformation (of which the details are beyond the scope of this text) to the original *fir*-filter, to give it an extra input and output for the purpose of connecting it to other FIR-filters. We can then *fold* this new *fir"* function over a list of tuples (containing the n-way split coefficients and previous x values), resulting in the *firSplit* function.

The netlist interpretation of 4-way split 16-taps instantiation of the above FIR-Filter description is depicted in Figure 9.

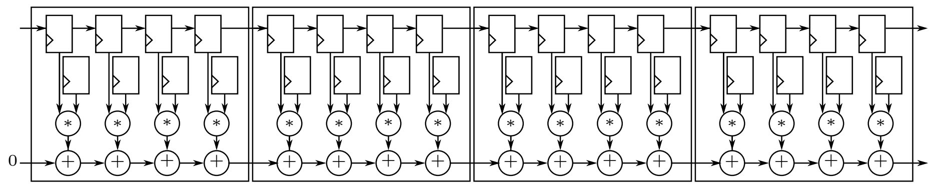

Fig. 9. 16-taps FIR-filter: four times four-taps FIR-filter

7 Related Work

In an attempt to raise the abstraction level of hardware descriptions, a great number of approaches based on functional languages have been proposed in the past [2,5,6,7,10,11,13,14]. The idea of using functional languages for hardware descriptions actually started in the early 1980s [6,14], a time which also saw the birth of VHDL. In an attempt to reduce the effort involved with prototyping a new language, such as creating all the required tooling like parsers and type-checkers, many functional HDLs [2,5,7,11] are embedded as a domain specific language (DSL) within the functional language Haskell. The CλaSH system differentiates itself from these embedded DSLs by using (a subset of) the Haskell language *itself* for the purpose of describing hardware.

Bluespec [12] is a high-level synthesis language that features guarded atomic transactions and allows for the automated derivation of control structures based on these atomic transactions. Bluespec, like CλaSH, supports polymorphic typing and function-valued arguments. Bluespec's syntax and language features *had* their basis in Haskell. However, in order to appeal to the users of the traditional HDLs, Bluespec has adapted imperative features and a syntax that resembles Verilog. As a result, Bluespec is (unnecessarily) verbose when compared to CλaSH.

The merits of polymorphic typing and function-valued arguments are now also recognized in the traditional HDLs, exemplified by the new VHDL-2008 standard [1]. VHDL-2008 support for generics has been extended to types and subprograms, allowing a designer to describe components with polymorphic ports and function-valued arguments. Note that the types and subprograms still require an explicit generic map, while the CλaSH compiler automatically infers types, and automatically propagates function-valued arguments. There are also no (generally available) VHDL synthesis tools that currently support the VHDL-2008 standard.

8 Concluding remarks

In the above we showed that Haskell is an adequate language to specify hardware and that it is close to an original mathematical specification. We also indicated that it is well possible to have transformational rules to derive a final architecture from a given specification.

The current state of the research of our team is that several non-trivial examples are specified using Haskell and translated into synthesizable VHDL using CλaSH. These examples include a floating point reduction circuit with a pipelined addition component, and a dataflow processor. Currently, we are investigating the aspects of a comparison between Haskell and traditional hardware specification languages such as VHDL and SystemC. Further issues are the definition of the exact subset of Haskell that can be synthesized to hardware, extending the system CλaSH with a graphical visualization of a specified architecture, and formalizing the internal rewrite mechanism.

References

1. VHDL Language Reference Manual (2008)
2. Axelsson, E., Claessen, K., Sheeran, M.: Wired: Wire-Aware Circuit Design. In: Proceedings of Conference on Correct Hardware Design and Verification Methods (CHARME). Lecture Notes in Computer Science, vol. 3725, pp. 5–19. Springer Verlag (2005)
3. Baaij, C., Kooijman, M., Kuper, J., Boeijink, A., Gerards, M.: CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In: Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, Nice, France. pp. 714–721. IEEE Computer Society Press, Los Alamitos, USA (September 2010)
4. Bijlsma, T., Bekooij, M.J.G., Jansen, P.G., Smit, G.J.M.: Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In: Falk, H. (ed.) Proceedings of the 11th international workshop on Software & compilers for embedded systems (SCOPES), Munich, Germany. ACM International Conference Proceeding Series, vol. 296, pp. 33–42. ACM Press, New York (March 2008)
5. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in Haskell. In: ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming. pp. 174–184. ACM, New York, NY, USA (1998)

6. Cardelli, L., Plotkin, G.: An Algebraic Approach to VLSI Design. In: Proceedings of the VLSI 81 International Conference. pp. 173–182 (1981)
7. Gill, A., Bull, T., Kimmell, G., Perrins, E., Komp, E., Werling, B.: Introducing kansas lava. In: 21st International Symposium on Implementation and Application of Functional Languages. LNCS 6041 (November 2009)
8. Jones, S.P. (ed.): Haskell 98 language and libraries, Journal of Functional Programming, vol. 13 (2003)
9. Kuper, J., Baaij, C., Kooijman, M., Gerards, M.: Exercises in architecture specification using CλaSH. In: Proceedings of the Forum of Specification & Design Languages 2010: FDL 2010, Southampton, United Kindgom (September 2010)
10. Li, Y., Leeser, M.: HML, a novel hardware description language and its translation to VHDL. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 8(1), 1–8 (Feb 2000)
11. Matthews, J., Cook, B., Launchbury, J.: Microprocessor specification in Hawk. In: Proceedings of 1998 International Conference on Computer Languages. pp. 90–101 (May 1998)
12. Nikhil, R.S.: Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In: Philippe Coussy and Adam Morawiec (ed.) High-Level Synthesis - From Algorithm to Digital Circuit, pp. 129–146. Springer Netherlands (2008)
13. Sander, I., Jantsch, A.: System Modeling and Transformational Design Refinement in ForSyDe. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 23(1), 17–32 (January 2004)
14. Sheeran, M.: μFP, a language for VLSI design. In: LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming. pp. 104–112. ACM, New York, NY, USA (1984)
15. Verdoolaege, S., Nikolov, H., Stefanov, T.: pn: A Tool for Improved Derivation of Process Networks. EURASIP Journal on Embedded Systems 2007 (Article ID 75947)