Designing a dataflow processor using CλaSH

Anja Niedermeier, Rinse Wester, Kenneth Rovers, Christiaan Baaij, Jan Kuper, Gerard Smit
University of Twente, Department of Computer Science
Enschede, The Netherlands

***Abstract*—In this paper we show how a simple dataflow processor can be fully implemented using CλaSH, a high level HDL based on the functional programming language Haskell. The processor was described using Haskell, the CλaSH compiler was then used to translate the design into a fully synthesisable VHDL code. The VHDL code was synthesised with 90 nm TSMC libraries and placed and routed. Simulation of the final netlist showed correct behaviour. We conclude that Haskell and CλaSH are well-suited to define hardware on a very high level of abstraction which is close to the mathematical description of the desired architecture. By using CλaSH, the designer does not have to care about internal implementation details like when designing with VHDL. The complete processor was described in 300 lines of code, some snippets are shown as illustration.**

I. Introduction

As current embedded systems become increasingly complex, new ways of modelling these systems are being investigated. Different levels of abstraction can be used to keep the system manageable. In this paper, CλaSH [1] is used as a language for specifying hardware on a high traction levels. CλaSH is a recently developed hardware description system which compiles specifications written in the functional programming language Haskell into VHDL. Haskell offers higher abstraction mechanisms than existing hardware description languages, thus the designer does not have the burden of specifying several implementation details.

This paper presents the design process of a complete architecture using CλaSH. The designed architecture is a simple dataflow processor which is taken as example as its functionality can be easily modelled in a functional programming language as the principles are very close to mathematics. The processor is designed based on previously presented dataflow processors. The focus of this paper is thus not the designed architecture itself but more a presentation of a design method which has a short design time and is very close to the initial concepts of the architecture.

II. Dataflow processors

Dataflow processors can directly execute dataflow graphs[2] which are mathematical representations of programs in which nodes represent operations and edges (called arcs) represent the dependencies between these operations. Arcs carry the data to other nodes as tokens. A node can only execute when all required data is available (the firing rule). During firing the node consumes all tokens on the input and produces result-tokens on the output. Several nodes may fire at the same time which may also trigger firing of other nodes.

Dataflow machines do not use a central program counter as in von Neumann architectures, but use the firing rules of dataflow nodes to trigger the execution of operations. The first machine capable of executing dataflow graphs is the static dataflow machine developed at MIT [3]. The dataflow graph resides in a special memory containing instruction cells which implement the firing rule. When a cell fires, the operands and instruction are sent to an execution unit which executes the instructions and sends the result back to the memory.

The Monsoon[4] is the first implementation with an explicit token store (ETS) to provide a more efficient token storage. In an ETS, every node in the dataflow graph is assigned a unique memory location. When a token is sent to a node it is checked if there is already a token present at the corresponding address in the token store. If not, the token is stored at that address. If yes, a match occured, i.e. the firing rule of the node is satisfied and the execution is triggered. A *presence bit* is used to indicate whether an address in the token store is occupied.

III. Designing hardware using Haskell and the CλaSH compiler

This section gives a short introduction to designing hardware using Haskell and CλaSH, the CAES[1] Language for Synchronous Hardware. The CλaSH compiler was recently developed at the CAES group at the University of Twente, it translates a Haskell description of a design to fully synthesisable VHDL. It is directly integrated into ghc [5], an open source Haskell compiler. A detailed description of the working principle of CλaSH can be found in [1], several design examples in [6]. The idea behind CλaSH is that electronic circuits can be seen as a mathematical function: For a certain set of inputs, a determined output is produced. An electronic circuit can thus intuitively be modelled in a functional programming language.

With Haskell as well as with other functional languages, it is possible to achieve a very concise description of the desired architecture. Before CλaSH, other approaches were presented to describe hardware with help of Haskell, like Lava [7], which is an HDL embedded in Haskell and ForSyDe [8], which uses Haskell for system modelling. In contrast to CλaSH, they do not directly use a subset of Haskell but use Haskell to define their syntax. That has the disadvantage that many of Haskell's features like control structures (e.g. guards, if-else) or polymorphism are not supported whereas they are fully supported in CλaSH.

The CλaSH compiler produces fully synthesisable VHDL code from a given Haskell description which is compliant to the CλaSH restrictions which are described in [1] (e.g. no dynamic lists but vectors, a state of a function is marked with the `State` keyword). Higher order Haskell functions like `map` or `zip` are fully supported as are control structures like guards

[1]Computer Architecture for Embedded Systems (University of Twente)

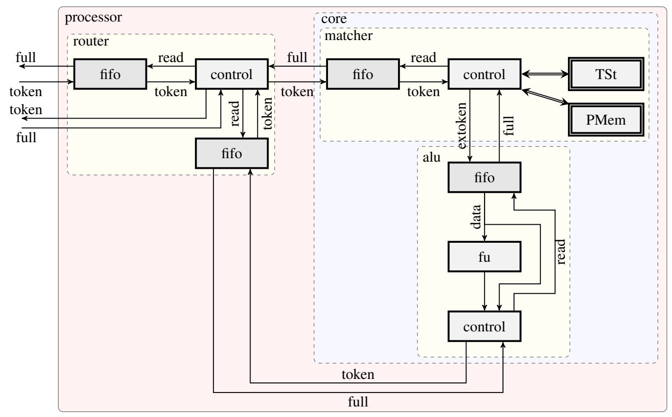

Figure 1. Schematic of the proposed architecture

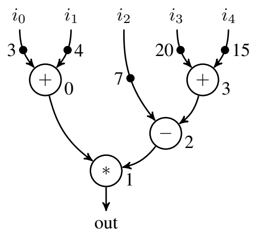

Figure 2. Example: Graph for the expression $(i_0 + i_1) * (i_2 - (i_3 + i_4))$

Table I
LIST OF TOKENS FOR GRAPH IN FIGURE 2

Value	Destination
3	$0, L$
4	$0, R$
7	$2, L$
20	$3, L$
15	$3, R$

or pattern matching and polymorphism. As CλaSH is integrated into ghc, simulation of the design is very fast compared to a full VHDL simulation.

The clock does not have to be explicitly defined. The designer describes the desired functionality of a module between two clock cycles as a transition from the current state to the next.

IV. Proposed Architecture

The proposed architecture is based on the principles of dataflow processors found in literature [2]. It is implemented as a static dataflow machine like [3], but with the explicit token store (ETS) principle presented in [4].

An overview is displayed in Figure 1. The processor consists of three main modules, namely a router, which arbitrates data from both the external and the internal input, a matcher, which is responsible for the matching process, i.e. the central principle of a dataflow machine, and an arithmetical logical unit (ALU), which performs calculations of the data sent by the matcher. Tokens travelling through the processor contain a data value and the destination address.

The whole processor can itself be considered a dataflow graph. This means that every connection between the modules corresponds to an arc on which tokens can be stored. In the proposed architecture, buffers at the input of each module are used to store those tokens. The same holds for the modules, every module in the processor corresponds to a node which operates using the firing rules of dataflow.

The router is responsible for managing incoming data from the outside (the external input) and data from within the processor (the internal input). Data which is present in the buffer of the internal input has priority over data in the external input. Also, the router can send data out of the processor.

The matcher consists of the token storage (TSt), which implements the ETS principle, the program memory (PMem), which stores the operation in form of an *opcode* and the destination address(es) for every node in the graph, and a control unit that takes care of the matching process. For each incoming token from the router it is checked whether it can be matched with a token already in the token storage. If not, the token is stored. If a match is found, the values of both tokens, i.e. the stored one and the incoming one, are sent to the ALU together with the *opcode* and the destination address(es) from the program memory. The token which was stored in the token storage is then deleted from the storage.

The ALU can perform either an addition, a subtraction or a multiplication. With the *opcode*, it is determined which operation has to be performed. Each computation takes one clock cycle, i.e. there is no pipelining and the result is immediately sent to the output.

By connecting the modules like shown in Figure 1, a complete (though limited) dataflow processor is constructed.

A. Execution of dataflow graphs

The dataflow processor is programmed by defining the destination of each node in the graph. Suppose a graph like the one shown in Figure 2. The graph represents the expression $out = (i_0 + i_1) * (i_2 - (i_3 + i_4))$ with $i_0 = 3, i_1 = 4, i_2 = 7, i_3 = 20, i_4 = 15$ as example input values.

In order to calculate the result for a given set of inputs, the input values are sent to the corresponding inputs in forms of tokens. In order to calculate $(3 + 4) * (7 - (20 + 15))$, which is also used in Figure 2, 3 has to be sent to the left input of node 0, 4 to the right input of node 0 and so on. The list of tokens is shown in Table I.

The temporary data values, i.e. the values resulting from one computation and travelling to the next computation, have to be sent to the correct destination. The destination is determined from the program memory. For the example graph, the program memory is shown in Table II.

V. Implementation

In this section, a brief introduction to the implementation of the processor is given.

Table II
PROGRAM MEMORY FOR GRAPH IN FIGURE 2

Node	Operation	Destination
0	ADD	$1, L$
1	MUL	*out*
2	SUB	$1, R$
3	ADD	$2, R$

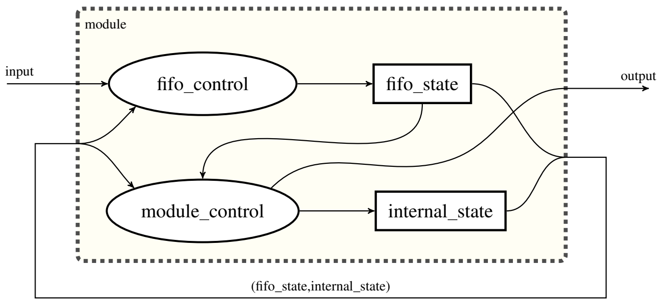

Figure 3. General implementation of the modules

A. Buffers

To implement the buffers at the inputs, fifo buffers were used. The input of the fifo consists of a token wrapped in the so-called `Maybe` type[2], i.e. either a new token was received or not, and a *read*-signal from the module which indicates if a value has been read from the fifo and can be erased. The output is a `Maybe` token and a *full*-signal indicating if the fifo is full. As flow control mechanism back pressure is used, i.e. when the buffer is full, its *full* line is set to true which notifies the sending module that no more data should be sent.

B. Tokens

Tokens consist of a value and a destination. The destination consists of an address which represents the node in the graph and the input of the node, i.e. *left* or *right*. The implementation in Haskell is as follows:

```
data Side  = L | R
type Word  = Int16
type Dest  = (Int7, Side)
type Token = (Word,Dest)
```

The keyword `data` defines new data values. The keyword `type` is used to define a new data type by using existing data types.

The data which is sent from the matcher to the ALU is an extended token which combines two data values, the *opcode* and four destinations which are wrapped in the `Maybe` type. The Haskell implementation of the extended token type looks as follows:

```
data Op      = ADD | SUB | MUL
type ExToken = (Word,Word,Op,
                (Vector 4 (Maybe Dest)))
```

C. General implementation of the modules

The general implementation of the modules is shown in Figure 3. The structure is similar for all modules, as they all have an internal state (*internal_state*), a state of the fifo(s) (*fifo_state*), and data input and output. The internal state and the fifo state are controlled by their respective control functions *module_ctrl* and *fifo_ctrl*. *fifo_ctrl* depends on the data input and the state of the fifo, *module_ctrl* is a function of the current output of the fifo and the internal state of the module. Both control function resemble a mealy machine. Both state elements are then combined and fed back into the module as Haskell natively does not have a notion of state. The implementation looks as follows:

[2]a Haskell datatype which can have the values `Just x`, indicating a valid value `x` or `Nothing`, indicating no value.

```
module (State (fs,ms)) i = ((State (fs',ms')),o)
    where
        (fs',fo) = fifo_control   fs i
        (ms',o)  = module_control ms fo
```

where `fs` and `ms` denote the current fifo and internal state, `fs'` and `ms'` are the new states, `i` and `o` are input and output and `fo` is the current output of the fifo. The type of the output is a `Maybe` type to distinguish between a token present at the output and no token present.

D. Matcher

As the matcher is the central module which implements the dataflow principle, its implementation is described in more detail. The structure is implemented as shown in Figure 3. The internal state of the matcher consists of the token storage *TSt* and the program memory *PMem*. The state input to `module_control` are the states of *TSt* and *PMem*, the state output is only the new state of *TSt* as the program memory does not change during execution. The data input consists of the token sent by the router `(Just (v,(u,s))`, where `v` is the value and `(u,s)` is the destination) and the *full*-signal from the ALU input buffer (`fi`). The data output consists of the *read*-signal to the fifo which indicates if a value was taken out of the fifo and the data packet which is sent to the ALU. The Haskell implementation of `module_control` looks as follows:

```
module_control (tst,pmem) (Just (v,(u,s)),fi)
 | not (check_match u tst)  = (tst',(True,Nothing))
 | check_match u tst && fi  = (tst,(False,Nothing))
 | otherwise     = (tst'',(True,(Just (l,r,op,d))))
   where
    (l,r) | s == L    = (v,(tst!u))
          | otherwise = ((tst!u),v)
    tst'   = addToken tst (v,u)
    tst''  = clearPBit tst u
    (op,d) = pmem!u
```

The function is divided into three cases. (1) No match is found for the incoming token; (2) a match is found but the ALU buffer is full, i.e. it cannot receive any new data; or (3) a match is found and the ALU can receive data.

The different cases are modelled with Haskell guards, their principle is illustrated at the example of an inverter:

```
f :: Bool -> Bool
f  i
 | i         = False
 | otherwise = True
```

If the input `i` is `True`, i.e. the first case, the result is `False` If `i` is `False`, which is modelled in the `otherwise` case (which is the default case if all other cases evaluate to `False`), the result is `True`

Furthermore, some helper functions are used in `module_control`. `check_match u tst` is `True` if there is a match for the current token at address `u` in `tst`, otherwise it is `False`. `addToken tst (v,u)` adds a value `v` to `tst` at index `u`, and `clearPBit tst u` clears the presence bit in `tst` at index `u`, i.e. sets the content to `Nothing`. The `!` operator is the indexing operator in a CλaSH vector.

Table III
RESULTS FOR ASIC SYNTHESIS

area	gates	cells	power
70850 μm^2	33470	7460	6.9 mW

E. Program memory

The program memory is implemented as a vector of length 128, i.e. currently 128 nodes in the dataflow graph are possible. Each element of the vector contains the *opcode* for the corresponding node in the graph and one up to four destinations. The destinations are wrapped in the `Maybe` datatype to distinguish between valid and invalid destinations as nodes in a dataflow graph usually have a different number of destinations. The number of the node is used as the address, i.e. to index the vector. The Haskell implementation is as follows:

```
type PMem = Vector 128
            (Op,(Vector 4 (Maybe Dest)))
```

where the keyword `Vector` denotes the CλaSH datatype for a vector with a defined size.

F. Token storage

The token storage is also a vector of 128 elements, each element has space for one data value. The rest of the data token, i.e. the destination, is not stored as it can be derived from the second token which is matched with the token value stored in the token storage. Each element is a `Maybe` datatype to distinguish between empty and occupied spots, i.e. to model a presence bit required for the ETS principle. Like in the program memory, the number of the node is used as address. The implementation in Haskell looks as follows:

```
type TSt = Vector 128 (Maybe Word)
```

G. Arcs between the modules

The modules are connected using Haskell's so-called arrow-abstraction. Each module is wrapped into an arrow together with its initial state. Several modules can be grouped by defining a new arrow where the arrows of the modules are connected. The arrow for the processor is as follows:

```
processorA = proc (di,fi) -> do
  rec (d,d2,f,f2) <- routerA -< (di,d1,fi,f1)
      (d1,f1)     <- coreA   -< (d2,f2)
 returnA -< (d,f)
```

where `routerA` and `coreA` are arrows for the router and the core, respectively, `di` and `fi` are the inputs to the processor, `d` and `f` are the outputs. `d1`,`d2`,`f1`, and `f2` are the connections between the router and the core.

VI. RESULTS AND DISCUSSION

The generated VHDL code was verified using a four-point FFT and different mathematical expressions. The simulations showed that the design worked as expected, i.e. the CλaSH generated VHDL code resembles the functionality of the Haskell description.

The design was fully synthesised and placed and routed using TSMC 90 nm low power library for a clock speed of 100 MHz. A four-point FFT was used to extract power numbers. The results for both the area and the power consumption are shown in Table III.

The Haskell implementation of the complete processor was 300 lines of code, including comments and type definitions. Such a compact description also greatly simplifies debugging.

To evaluate the outcome of the CλaSH implementation, the same processor was also implemented using pure VHDL. The details are presented in [9] and are not discussed here, only a brief summary is given. Both implementations were synthesised for an FPGA and also for ASIC. For the FPGA synthesis, the results in terms performance were similar with only small deviations. The ressource utilisation differed for the number of registers, we assume it could be caused by the way CλaSH implements registers (e.g. no *write-enable* signal is used).

The ASIC synthesis (without clock gating) results were surprising, as the VHDL implementation required considerably more area (roughly a factor of 2.5) and consequently also more power (a factor of 3). At the moment, we are not sure what the reason for that is. However, when clock gating is enabled, the VHDL design is more power efficient, roughly by a factor of 8. The reason for that is that clock gating is not inserted during synthesis for the CλaSH implementation. We assume that this is again due to the missing *write-enable* signal.

VII. CONCLUSION AND FUTURE WORK

In this paper we presented the complete design of a dataflow processor by descriping the architecture in Haskell and then using CλaSH to translate the description to fully synthesisable VHDL code. The VHDL was synthesised and placed and routed, its correct functionality was verified by simulation. Due to full support of Haskell's features like control structures and polymorphism in CλaSH, the processor could be described in a very compact way. In Haskell, different levels of abstraction can be used, in general the level of abstraction level is very high and close to the mathematical description of a design. Internal implementation details like timing are completely hidden from the designer. Summarising, we can say that it is possible to design a complex architecture using CλaSH with relatively little effort. However, there are still many open issues which are being investigated at the moment, like the problem with clock gating.

REFERENCES

[1] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "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*, September 2010.

[2] A. H. Veen, "Dataflow machine architecture," *ACM Comput. Surv.*, vol. 18, no. 4, pp. 365–396, 1986.

[3] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ISCA '75: Proceedings of the 2nd annual symposium on Computer architecture*. New York, NY, USA: ACM, 1975, pp. 126–132.

[4] G. M. Papadopoulos, "Monsoon: an explicit token-store architecture," in *In Proc. of the 17th Annual Int. Symp. on Comp. Arch*, 1990, pp. 82–91.

[5] http://www.haskell.org/ghc/.

[6] J. Kuper, C. Baaij, M. Kooijman, and M. Gerards, "Exercises in architecture specification using cλash," in *Proceedings of the Forum on Specification and Design Languages*, 2010.

[7] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in Haskell," in *Proceedings of the third ACM SIGPLAN international conference on Functional programming*. ACM, 1998, pp. 174–184.

[8] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, p. 17, 2004.

[9] A. Niedermeier, R. Wester, K. Rovers, C. Baaij, J. Kuper, and G. Smit, "Comparing CλaSH and vhdl by implementing a dataflow processor," in *Proceedings of the Program for Research on Embedded Systems and Software (PROGRESS)*, 2010.