Comparing CλaSH and VHDL by implementing a dataflow processor

Anja Niedermeier, Rinse Wester, Christiaan Baaij, Jan Kuper, Gerard Smit
Zilverling 4078, P.O. Box 217, 7500AE Enschede Computer Architecture for Embedded Systems
University of Twente

***Abstract*—As embedded systems are becoming increasingly complex, the design process and verification have become very time-consuming. Additionally, specifying hardware manually in a low-level hardware description language like VHDL is usually an error-prone task. In our group, a tool (the CλaSH compiler) was developed to generate fully synthesisable VHDL code from a specification given in the functional programming language Haskell. In this paper, we present a comparison between two implementations of the same design by using CλaSH and hand-written VHDL. The design is a simple dataflow processor. As measures of interest area, performance, power consumption and source lines of code (SLOC) are used.**

The obtained results indicate that the CλaSH-generated VHDL code as well as the netlist after synthesis and place and route are functionally correct. The placed and routed hand-written VHDL code has also the correct behaviour. Furthermore, a similar performance is achieved. The power consumption is even lower for the CλaSH implementation. The SLOC for CλaSH is considerably smaller and it is possible to specify the design in a much higher level of abstraction compared to VHDL.

I. Introduction

With ever smaller features sizes in today's technology, it is possible to integrate complex functionality into a very small design. Systems are becoming smaller, faster and more complex, embedded systems are present in nearly every part of our daily life. But along with it comes a serious drawback: The design and verification complexity has immensely increased over the past years. A design can easily contain several ten thousands lines of HDL code and also the simulation time is becoming a bottleneck during the design process. Therefore, it is desirable to have a different design method than the traditional HDL, usually being VHDL or Verilog.

In our group we developed CλaSH [1], a compiler that uses a Haskell description of the desired architecture to generate a VHDL netlist. The generated VHDL netlist can be simulated with any VHDL simulation tool and is fully synthesisable for both FPGAs and ASIC technologies. In this paper, we present a comparison between an architecture which is fully implemented with CλaSH and one which was described in VHDL. The architecture described in this paper is a simple dataflow processor. As measures, we will use area usage and maximum clock speed on an FPGA and power numbers and area for 90 nm TSMC low power libraries. Also, the lines of code is compared.

The idea to describe hardware using functional languages is not new. Before CλaSH, other approaches were presented like Lava [2], which is an HDL embedded in Haskell and ForSyDe [3], 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. A general discussion on high level synthesis can be found in [4].

The remainder of this paper is structured as follows: First, an introduction to dataflow graphs and dataflow processors is given. Afterwards, the design process using CλaSH is described. Then, the simple dataflow processor which is our design example is explained. Following, details on the implementation in both Haskell and VHDL are given. Then, the results are presented by means of the measures explained above. Finally, a conclusion is given.

II. Dataflow graphs and processors

Dataflow graphs [5] are a way of representing mathematical expressions in a graph, where operations are presented as actors (also called nodes) and dependencies between operations are represented by arcs between the actors. Data travels in form of tokens on the arcs, whenever all required tokens are available at the inputs of an arc, the actor fires, i.e. it consumes all tokens and produces output tokens.

Dataflow processors are machines that can directly execute dataflow graphs. Instead of a central program counter as von Neumann architectures have, they use the firing rules of dataflow nodes to trigger the execution of operations. The first machine capable of executing dataflow graphs was the static dataflow machine developed at MIT [6], later, more sophisticated architectures were presented ([7], [8]).

The drawback of inefficient token storage was solved by the Monsoon [9], which implemented an explicit token store (ETS). 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 occurred, 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

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

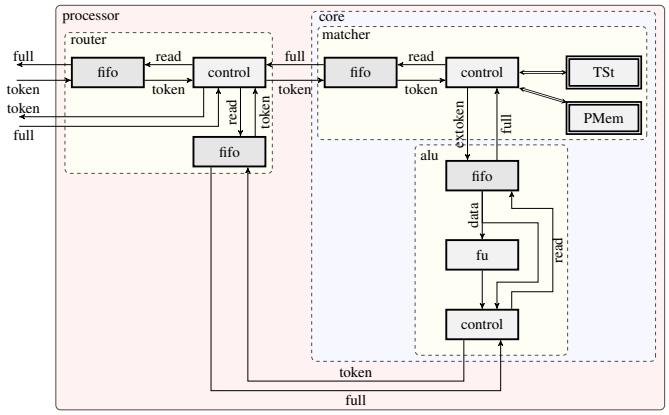

Figure 1. Schematic of the proposed architecture

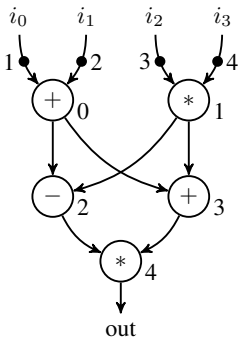

Figure 2. Example: Graph for the expression $((i_0 + i_1) - (i_2 * i_3)) * ((i_0 + i_1) + (i_2 * i_3))$

Synchronous Hardware. The CλaSH compiler was recently developed at the CAES group at the University of Twente. 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.

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 or pattern matching and polymorphism. As CλaSH is integrated into ghc [10] (the Haskell compiler environment), 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.

A detailed description of the working principle of CλaSH can be found in [1], several design examples in [11].

IV. The Architecture

The proposed architecture is based on the principles of dataflow processors found in literature [5]. It is implemented as a static dataflow machine like [6], but with the explicit token store (ETS) principle presented in [9].

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 fully functional (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 $out = ((i_0 + i_1) - (i_2 * i_3)) * ((i_0 + i_1) + (i_2 * i_3))$ with $i_0 = 1, i_1 = 2, i_2 = 3, i_3 = 4$ as 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 $((1+2)-(3*4))*((1+2)+(3*4))$, which is also used in Figure 2, 1 has to be sent to the left input of node 0, 2 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.

Table I
LIST OF TOKENS FOR GRAPH IN FIGURE 2

Value	Destination
1	$0, L$
2	$0, R$
3	$1, L$
4	$1, R$

Table II
PROGRAM MEMORY FOR GRAPH IN FIGURE 2

Node	Operation	Destination
0	ADD	$(2, L); (3, L)$
1	MUL	$(2, R); (3, R)$
2	SUB	$(4, L)$
3	ADD	$(4, R)$
4	MUL	*out*

V. IMPLEMENTATION IN CλASH

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

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. For flow control, 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 (*state*), a control function *control*, and data input and output. Furthermore, the module can have one or eventually more additional functional combinatorial block, in the picture denoted with *fu*. *control* is a function of the current input and the internal state of the module and it determines both the new state and the output of the module. As Haskell does not have a notion of state by itself, the state is fed back into the module. The implementation looks as follows:

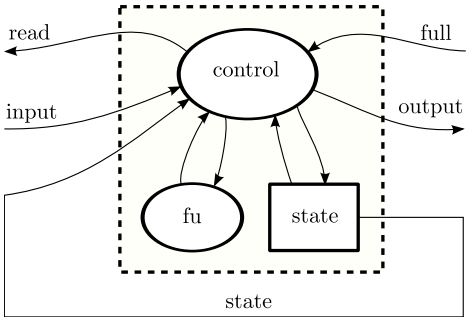

Figure 3. General implementation of the modules

```
module (State s) i = ((State s'),o)
    where
        (s',o)              = module_control s o
        module_control s o = ...
        fu ...              = ...
```

where `s` denotes the current internal state, `s'` is the new state, and `i` and `o` are input and output. The type of the output is a `Maybe` type to distinguish between a token present at the output and no token present.

D. Program memory

The program memory is implemented as a vector of length 128, i.e. currently 128 nodes in the dataflow graph which is executed on the processor are possible. Each element of the vector contains the *opcode* for the corresponding node in the graph and one 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 D128
            (Op,(Vector D4 (Maybe Dest)))
```

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

E. 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

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

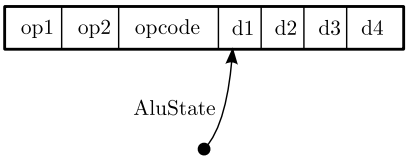

Figure 4. ALU State

program memory, the number of the node is used as address. The implementation in Haskell looks as follows:

```
type TSt = Vector D128 (Maybe Word)
```

F. Matcher

The structure of the matcher 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.

G. ALU

The ALU is taken as example to explain the implementation in more detail for both Haskell and VHDL. In this section, the Haskell implementation is presented, later in VI, the VHDL implementation is shown.

The ALU structure corresponds to the one shown in Section V-C. In Figure 4, the ALU state is illustrated. The rectangular block represent the input buffer, it stores the two operands *op*1 and *op*2, the opcode and the four destinations. The ALU state is thus a pointer to the current destination which is handled.

The fu module represents the function unit within the ALU, i.e. the block that performs the actual computation. It can execute, as described before, addition, multiplication and subtraction, the correct operation is selected according to the op-code which is sent to the function unit.

The control unit manages the incoming signals, i.e. the full signal from the destination module and the data input from the input arc. It also handles the state update, i.e. it checks if all valid destinations which are required for the current operation are processed. The procedure is depicted in Figure 5. The control implements this procedure by distinguishing three different cases:

1) The output arc is full, i.e. no data can be sent out. In that case, no data is sent out, no data is taken out of the fifo and the state remains the same.
2) The output arc is not full and the incoming data is valid, i.e. it contains values to compute. Then, the result together with the current destination is sent to the output, and the read signal and new state are determined depending if it was the last destination.

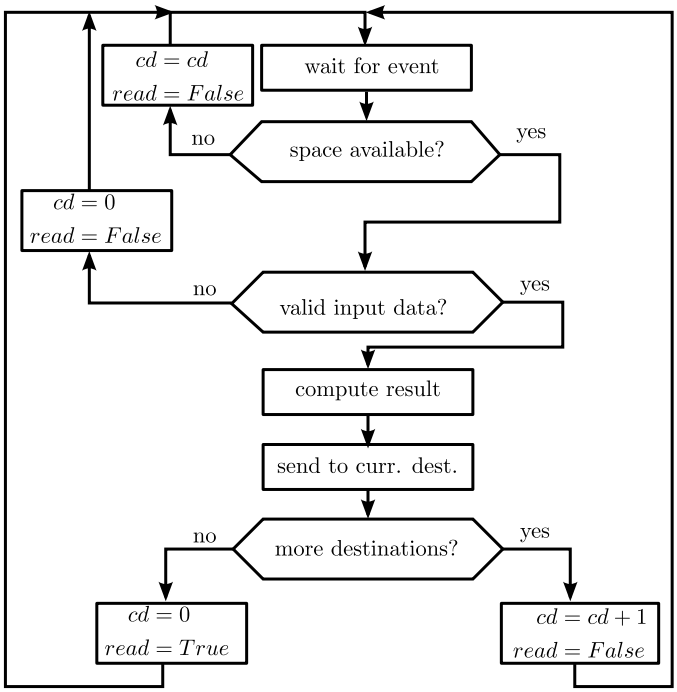

Figure 5. Handling of multiple destinations in the ALU

3) The default case, i.e. there is no valid incoming data. Here, no data is produced, nothing is taken out of the fifo and the index of the current destination, i.e. the state, is set to zero.

```
alu_control (di,fi,cd)
 | fi            = (Nothing,False,cd)
 | di /= Nothing = ((Just (res,cd)),r1,c1)
 | otherwise     = (Nothing,False,0)
   where
    (r1,c1)
     | next_dest == Nothing = (True,0)
     | otherwise            = (False,(cd+1))
    next_dest
     | cd < 3    = ((fth(di))!(cd+1))
     | otherwise = Nothing
    res = case (fu di) of Just a -> a ;
          Nothing -> 0
```

The function unit responsible for the actual computation is implemented as follows:

```
fu (Just (a,b,ADD,_)) = Just (a+b)
fu (Just (a,b,SUB,_)) = Just (a-b)
fu (Just (a,b,MUL,_)) = Just (a*b)
fu  _                 = Nothing
```

H. Composition of the system

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.

I. Complete design flow

The complete design flow was as follows: Two different design methods are used to implement the dataflow processor, namely using Haskell and using pure VHDL. The VHDL code from both implementation methods is simulated to verify correct functionality and subsequently synthesised, placed, and routed using TSMC 90nm technology. After the synthesis and place and route steps, the result is simulated to verify that the generated netlist still shows correct behaviour. Finally, both designs are compared in terms of the measures of interest mentioned above using a four-point FFT application as a benchmark.

VI. VHDL IMPLEMENTATION

The same processor design is also implemented using VHDL. Again data is fed to the processor via a router which sends the data further to the matcher. The matcher sends the opcode and both operands to the ALU when a match occurs. The ALU calculates the result within one clockcycle. The resulting token is sent to the router again which forwards the token to the dispatcher, completing the cycle of the architecture.

A. Relation to dataflow graphs

Also the VHDL implementation is based on a dataflow graph representation of the processor. Therefore every connection is an arc on which tokens can be stored and every node accepts all tokens on the input arc. The arcs are implemented using a FIFO with feedback signals *full* and *empty*. Every input of all nodes in the graph of the processor contains an arc and every output is connected to such an arc. A node can only fire when there are enough tokens on the input-arc and the arc connected to the output is not full.

The whole processor is composed by three (Router, Matcher and ALU) component instantiations in the top-level design. The synchronisation is implemented automatically by the dataflow principle: All nodes fire only if all required data and storage space is available.

B. Matcher

The matcher created in VHDL also contains a program memory and a tokens store. An additional node called presencebit memory[9] is used to determine whether a match is found. A separate module is used here because VHDL has no support for `Maybe` types. The principle of execution is still the same for both implementations: The ALU may execute if both left and right operands are available.

C. ALU

The processor contains a small ALU which is only able to perform additions, subtractions and multiplications. This ALU can execute nodes with up to four destinations in the dataflow program. The result is calculated in the first cycle and is directly sent to the router with the first destination. When the node addresses more destinations the result from the first cycle is resent with every destination. During the last cycle the arcs, forming the inputs of the ALU, are read such that new instructions can be accepted. The implementation in VHDL has the same structure as the one in CλaSH shown in figure 5. Figure 6 shows the structure of the ALU.

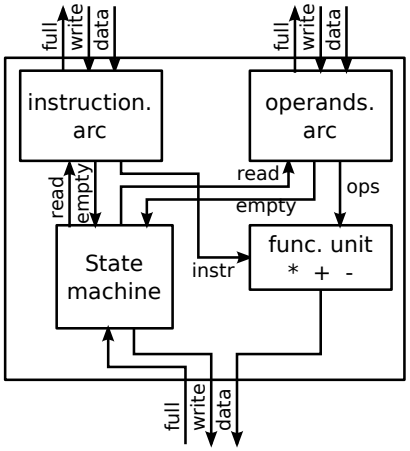

Figure 6. Structure of ALU in VHDL

As shown in figure 6 instructions arrive an the instruction arc and the operands arrive at the operands arc. A statemachine implements the firing rule: Output tokens are produced only when the instruction and operands are available and if there is space on the output-arc (the input of the router). The instructions itself are performed in the functional unit *(*func. unit) from which the results are forwarded to the output. Depending on the instruction token, the ALU sends the same result to several destinations.

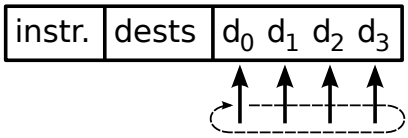

Figure 7. Instruction format of ALU

Figure 7 shows the instruction which is fed to the ALU. The instruction selects the mode of the functional unit (multiply, add or subtract). Followed by the instruction is *dests* which represents the number of valid destinations. During execution of a dataflow node, a pointer iterates over the set of valid destinations d_0 to d_3. When the result of the operation combined with the last destination is sent, the destination-pointer is reset and points to d_0 again.

VII. RESULTS AND SUMMARY

Both implementations were synthesised for an Altera FPGA and an Xilinx Virtex-6 FPGA (6VLX240TLFF1156). The results for the Xilinx FPGA are shown in Table III.

It was expected that the tooling should choose RAM-blocks to implement the memories of the Matcher. All memories

Table III
RESULTS FOR FPGA

FPGA	VHDL	CλaSH
CLB Slices	234	347
Function generators	936	1386
DFFs or Latches	175	2092
Max. Frequency $[MHz]$	178	189

Table IV
RESULTS FOR ASIC

ASIC	VHDL	CλaSH
area $[\mu m^2]$	189845	70850
gates	89685	33470
cells	20984	7460
power consumption $[mW]$	20.2	6.9

have synchronous write but asynchronous read such that every operation takes only one clockcycle. The RAM-blocks however can only be used with synchronous read. The tooling therefore implements the memory using single registers which require a lot of area. This problem occurs both with Xilinx and Altera FPGAs.

Compiling both designs to the FPGA showed a big difference in the amount of flipflop (175 for plain VHDL v.s. 2092 for CλaSH) inferred by the tooling, while other results shown in table III are more or less similar. The tooling is not able to map the memories of the CλaSH design onto complete LUTs while for the VHDL implementation this does happen. We think that this is caused by the way CλaSH handles registers, CλaSH introduces a feedback-loop wit a mux to either load the same state agian or a new one. The tooling is therefore not able to combine a set of registers and map them to a single LUT in a CLB. However a more detailed analysis is needed to comfirm this.

Afterwards, both implementation were synthesised for 90 nm TSMC low power libraries. Here we wanted to analyse the designs in terms of power consumption. For power estimation, the design was synthesised and placed and routed. The resulting netlist was simulated using a four point FFT as testbench. During simulation, a VCD (value change dump) file was generated, in this file, every change of each signal in the design is logged. The VCD file together with the library information for the target technology was used to extract power numbers.

First, the synthesis was performed for a clock frequency 200 MHz which was no problem for both implementations. During analysis of the power consumption we noticed that the CλaSH implementation did not insert any clock gating cells although clock gating was enabled during synthesis. In the VHDL implementation, clock gating cells were inserted. A quick analysis revealed that the CλaSH generated VHDL does not include *write enable* signals which are required for clock gating.

To have a fair comparison, we decided to deactivate clock gating during synthesis. Then however, the VHDL implementation could not be synthesised for 200 MHz anymore as the timing was not met. We decided to synthesise the design for 100 MHz which could be achieved for both designs. The resulting power consumption values were a bit surprising as the VHDL implementation consumed roughly three times more power than the CλaSH implementation. Also the number of gates and cells and thus consequently the area was bigger (by a factor of roughly 2.5). The results are displayed in Table IV.

At this moment, we are not sure why the area differs so greatly. We are busy with a detailed analysis of possible reasons.

VIII. CONCLUSION AND FUTURE WORK

During FPGA synthesis we discovered that the tooling did not use RAM blocks because of the asynchronous read signal. The next implementation therefore requires pipelined memory operation such that the tooling recognises the synchronous read and chooses RAM-blocks for implementation.

It turned out that the CλaSH generated VHDL does not allow automatic insertion of clock gating. This is a serious issue, as especially for embedded systems, where power consumption is a crucial factor. We are currently looking into a solution of this problem. Furthermore, we observed that, when clock gating was not activated, the CλaSH generated design was more power efficient than the VHDL implementation. Also this issue is currently being investigated.

Generally it can be said that with Haskell and CλaSH is it possible to describe complex designs with less lines of code (≈ 300 in CλaSHv.s. ≈ 1500 inVHDL), which simplifies both the actual design process but also verification and debugging. The CλaSH generated VHDL code is fully synthesisable and resembles the intended functionality which was described in the Haskell code. Furthermore, the synthesis results showed that in terms of area and performance it can be, when taking clock gating not into account, even better than hand-written VHDL code.

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] 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.

[3] 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.

[4] R. Gupta and F. Brewer, "High-Level Synthesis: A Retrospective," *High-Level Synthesis*, pp. 13–28, 2008.

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

[6] 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.

[7] J. Gurd, C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[8] K. Arvind and R. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, no. 3, p. 318, 1990.

[9] 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.

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

[11] 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.