System-Level Modelling of Dynamic Reconfigurable Designs using Functional Programming Abstractions

B.N. Uchevler, Kjetil Svarstad
Department of Electronics and Telecommunication, NTNU, Norway
najafiuc@iet.ntnu.no, kjetil.svarstad@iet.ntnu.no
Jan Kuper, Christiaan Baaij
Department of Computer Science, UTwente, Netherlands
J.Kuper@utwente.nl, C.P.R.Baaij@utwente.nl

***Abstract*—With the increasing size and complexity of designs in electronics, new approaches are required for the description and verification of digital circuits, specifically at the system level. Functional HDLs can appear as an advantageous choice for formal verification and high-level descriptions. In this paper we explain how to use high-level structures and concepts like *higher-order functions*, and *parametrization* together with *partial evaluation* implementation technique, to describe run-time reconfigurable systems in Haskell. We use the CLaSH tool to translate high-level Haskell descriptions into RT level, synthesizable VHDL. A simple design is used to show the ideas and is implemented on Suzaku-sz410 board for practical proof of concept.**

***Index Terms*—Key Words*: Run-Time Reconfiguration, Self-Reconfiguration, Functional HDL, Partial Evaluation**

I. Introduction

The diversity of applications for electronic systems increases everyday. Applications typically include software and hardware parts interacting together, and while software parts usually handle the control-driven and simple computation tasks, the hardware parts run the computation-intensive tasks of the application. Implementing specific computation-intensive tasks as Application Specific Integrated Circuits (ASIC) can cost a lot of time and money. Re-programmability and reconfigurability of hardware can introduce benefits like high flexibility, less area consumption, less time to market, even power saving. Generally speaking, *Reconfigurable Hardware* can fill the gap between ASIC and software in terms of performance, power consumption, and time to market criteria. The need for reconfigurability can be driven by 3 main factors: *Multiability* (to perform different functions at different times), *Evolvability* (to adapt to the environment and changes in standards over time), and *Survivability* (the system remains functional despite having a few failures) [20]. Different architectures have been proposed for reconfigurable hardware, but FPGAs are the most common reconfigurable hardware in practice. Using FPGAs as ASIC substitute was effective only in low-volume products before, but nowadays high volume production of FPGAs and rising None Recurring Engineering (NRE) costs of ASICs, shift the balance toward FPGAs [12]. Some of the reconfigurable architectures are capable of being reconfigured partially at run-time, which means it is possible to change some parts of the design on FPGA, without affecting other running parts. This feature is called Run-Time Reconfiguration (RTR).

Because of the exponentially growing number of transistors on silicon [10], and using low-level tools and methodologies for system design, a productivity gap is created which increases the time to market factor for electronic designs. In order to cope with this productivity gap, high-level tools and methodologies are required. Designing in a higher levels of abstraction can lead to improve the time to market factor and reduce the productivity gap.

It is possible to define specific areas inside FPGA as *Reconfiguration Regions* which communicate with other parts of the design through fixed and static channels such as *busmacros*.

Modules that are intended to be configured to the same reconfigurable area, are called *Reconfiguration Candidates* here. Therefore, every reconfiguration region has a group of reconfiguration candidates.

Figure 1 shows a typical run-time reconfigurable system with two reconfiguration regions (RR1 and RR2), in which, RR1 has A, B, and C modules as its reconfiguration candidates and RR2 has D and E modules as its reconfiguration candidates

Design Space Exploration (DSE) in higher levels of abstraction can discover better design alternatives in the early stages of the design flow. Extensive research has taken place on designing in higher levels of abstraction, but mainly for non-reconfigurable systems. Most of these works use high-level languages such as SystemC, Simulink or UML as a language for describing hardware and software together. In a system that includes RTR hardware, it is possible to share the hardware resources in temporal domain, thus design space exploration and partitioning for a RTR hardware should consider a new dimension which is the *time* dimension

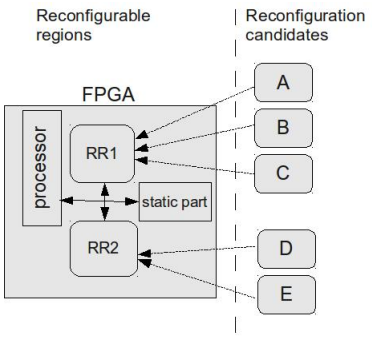

Figure 1. A typical run-time reconfigurable system

besides the spatial dimension.

There are several works in modeling and design methodologies for RTR systems based on high-level description languages like SystemC [23], [22], [18], [15], [19], [21], [2], [8], [4] Simulink [9], and UML [7], [17], [26], [11], [24], [25], [1], [14]. Because of the lack of features like *polymorphism, partial application, higher order functions*, etc, it is not easy to model RTR systems in traditional HDLs [5]. Because of these features and better support for formal verification, functional HDLs has appeared as an alternative. In this paper we will describe how to model RTR systems with Haskell which is a functional programming language. We use a tool named CLaSH to generate hardware from Haskell description of RTR systems.

In the next section we will discuss several modeling approaches for RTR systems mainly in SystemC and UML. In section III a general information about Haskell is presented and CLaSH tool will be introduced. In section IV we will discuss the proposed approach together with a simple RTR design example in Haskell. Finally a short conclusion is presented in section V.

II. Previous Work

The work in [15] is one of the most referenced works in the area of RTR modeling. The idea is to collect all reconfigurable alternatives of a specific reconfigurable area into a module called *Dynamically Reconfigurable Fabric* (*DRCF*) and configure the specific module when it is requested from other parts of the system. First, an analysis of modules is done to extract their interfaces and ports, then all the instances of the modules are found and analyzed in the top level (at the same level of hierarchy). After this phase, a *DRCF* component is created which includes all the interfaces and ports collected in the analysis phase. The *DRCF* module will contain all the reconfigurable alternatives at the same level of hierarchy and its instantiation replaces instantiations of all reconfiguration candidates. When a call to a specific reconfigurable module happens, if it is not already the active module in the *DRCF*, it will be activated in the *DRCF* component. There are several functions in the template to read and write from/to component, or to get low and high addresses of the memory space that contains the configuration bits of each *DRCF* component. In [19] authors have used OSSS methodology as a base and added RTR performance evaluation and synthesis support to it. In order to keep the reconfiguration details away from the designer, they use polymorphism like the polymorphism exists in C++ language [19]. It supports synthesis of inheritance, objects, classes, etc. Since the synthesis of pointers is challenging, for definition of parent classes and objects, they use a reconfigurable object which is a base for each reconfigurable *group*. Each reconfigurable group can have several reconfiguration candidates. Assigning one of the reconfiguration *candidate objects* to a base class of that object, or calling a function of a specific object can initiate the reconfiguration process to bring that reconfiguration candidate into action. They use Fossy tool to synthesize from C++ to VHDL. In order to do functional simulation of the design after getting the VHDL output of the Fossy tool, instances of all modules are included in the design connected with mux/demux to other parts of the design. For low-level synthesis, they use EAPR approach to generate partial bitstreams. More information on details of the tool-chain can be found in [19].

There are some other similar works done in the area of modeling RTR systems with SystemC in [3], [22], [16], [4].

In [Beierlein2003] which is based on Model Driven Architecture, they propose a modeling flow based on UML. In order to describe both the application and architecture model they use *use-case* diagrams, *state* diagrams, *collaboration* and MOCCA (Model Compiler for Configurable Architectures) language. MOCCA is designed by them which is similar to Java language. Control and software-friendly components are mapped to a general purpose processor and the computation-intensive components are mapped to a reconfigurable hardware. Output of the synthesis stage will be VHDL files (for reconfigurable tasks) and C++ or Java code (for software tasks) [7]. Some components are decomposed and some of them are composed to map them to different hardware/software components.

In [26] they proposed a methodology that receives both the application and platform models as input and produces bitstream files for FPGA. The application is defined as a set of communicating components. Communication between components are shown through ports and behavior of them is defined with state machines.

The platform components are modeled in the same manner. The mapping stage is done manually using specific stereotypes called *allocate*. After the mapping stage, the translation of the components from UML into software/hardware starts. They use Rhapsody tool to generate hardware/software codes (VHDL and C++). Adding the "Reconfigurable" stereotype to the allocation allows the code generation part to add reconfiguration calls while generating the low-level code. It automatically inserts busmacros into module wrappers. In [25] they have improved their modeling approach by *Design Patterns*. They use two common design patterns for modeling reconfiguration candidates, the *strategy* design pattern , and *state* design pattern. The definitions

of these design patterns can be found in [25]. More works based on UML can be found in [17], [11], [24], [1], [14].

III. CLaSH

Hardware Description Languages (HDLs) are proper for describing detailed hardware, but they can be cumbersome for describing higher levels of abstraction in larger designs such as polymorphism, higher order functions and parameterization[5]. Functional HDLs enable us to use features like polymorphism, higher order functions, and parametrization that makes it easy to describe larger designs. They also have the so called denotational semantics which makes it affordable to prove the equivalency of two designs and formal verification of hardware designs [5]. Being close to mathematical and formal description of hardware can enable us to avoid exhaustive tests of large designs. This feature makes functional HDLs more valuable these days.

CAES Language for Synchronous Hardware (CLaSH) is an experimental tool that accepts a subset of Haskell language as its input language.

A. Haskell

Haskell is a functional programming language in which computation is similar to the evaluation of mathematical functions. In contrast to *imperative programming,* Haskell is a *lazy* language and expressions are only evaluated when they are needed.

In Haskell it is possible to have functions as arguments of other functions which are called higher-order functions. Together with polymorphism it is possible to achieve higher level of generality in descriptions. For example, the following code describes a simple map function that puts higher-order functions, parametrization, and polymorphism together.

$$
\begin{aligned}
&map \;::\; (t \to u) \;\to [t] \;\to [u] \\
&map\ f \quad [\,] \quad = \; [\,] \\
&map\ f \quad (a : x) \quad = \; f \quad a \;:\; map\ f\ x
\end{aligned}
$$

Figure 2. Definition of the map function

It has an input argument of type $(t \to u)$, which is a function itself with input and output types of t and u respectively in which u and t can be of any type including user defined types. The second argument is a list of type t and the result is a list of type u. The map function simply applies an operation over a list of of type t and produces a list of of type u. Describing different applications with such general functions is easier than describing with functions specialized for different type such as Int, $Char$, etc.

In the example shown in Figure 2 specialization of the general function map to specific types is done during the compile time in software world. This example employs type-polymorphism with types t and u, higher-order function with accepting a function in its input arguments, and parametrization with accepting any function that matches the input-output signature of the input function without specifying it functionality.

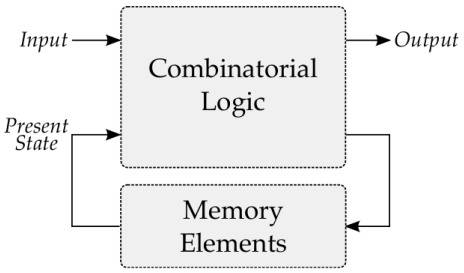

Figure 3. The basic Mealy machine [5]

$$
\begin{aligned}
&fullAdd\ \ carryIn\ (a\ ,\ b) \;=\; (sum\ ,\ \ carryOut) \\
&\quad where \\
&\qquad (sum1\ ,\ carry1) \;=\; halfAdd\ \ a\ \ \ b \\
&\qquad (sum\ ,\ carry2) \;=\; halfAdd\ \ \ carryIn\ \ \ sum1 \\
&\qquad carryOut \qquad\quad = \; hwxor\ \ \ carry2\ \ \ carry1
\end{aligned}
$$

Figure 4. Description of a FullAdder in CLaSH [5]

Concepts like polymorphism, higher-order functions, and parametrization are used to create more generality. By making the descriptions more general, there is a greater chance to reuse functions which increases the productivity of the designer, if it is done in higher levels of abstraction.

B. Modeling With CLaSH

CLaSH is used for modeling synchronous designs. Synchronous hardware can be represented by a *Mealy* machine as shown in Figure 3.

The Mealy machine can be described in CLaSH including both combinatorial parts and memory elements. It has been enhanced with state-hiding techniques using *Arrows* of Haskell [13]. Memory elements of the design have to appear in both input and output ports to make next-state values from present-state values, so the general signature of state-full modules, without using any state-hiding approach, will be as the following:

$state \;\to\; inputs \;\to\; (state\,,\, output)$

State arguments have the same type (on both sides) and represent the values of memory elements, but input and output arguments can have different types. Using state-hiding techniques, it is possible to describe designs with memory elements inside, without representing them in input and output ports of the module, which make it more readable and easier to deal with. CLaSH generates synthesizable VHDL code from Haskell code and it supports most of the high-level functions and operations of Haskell.

A simple example on how to describe a stateless circuit such as a full-adder in CLaSH is shown in Figure 4.

It is also possible to do a complete simulation with a regular Haskell compiler because all parts of the simula-

tion including circuit description, simulation code, and test inputs are valid Haskell. More details on CLaSH compiler including details, extensions, and applications can be found in [5],[13], and [6].

In the next session we will see how we extend CLaSH to support description of RTR systems with higher-order functions, parametrization, and partial evaluation. We use specific compiler annotations in our Haskell description which is discussed in the next session.

IV. Proposed Approach

In this section we focus on the ideas we used to model RTR concepts with Haskell. We investigate modeling of RTR systems using parametrization, partial evaluation, and higher order functions in Haskell. It is possible to use these high-level concepts in hardware as well as software. For example in [19] they exploited polymorphism in the description language to describe RTR systems in higher levels of abstraction. By defining proper candidate objects, it is possible to model RTR behavior in a high-level language like SystemC. We used higher-order functions and parametrization as generalization rules to provide higher levels of abstraction in the design flow of RTR systems. Higher-order functions in Haskell, are similar to function pointers in SystemC, but synthesizing pointers is not straightforward.

As a test case, we described two simple filters in Haskell and switched between them at run-time in real hardware in a self-reconfigurable system on FPGA. The following figure shows the block diagram of the FIR and IIR filters used.

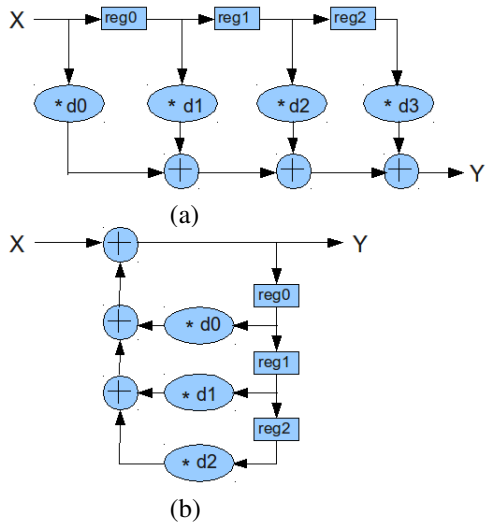

Figure 5. (a) An FIR filter with constant coefficients. (b) An IIR filter with constant coefficients.

```
...
dotp1 as bs z = vfoldl (+) z (vzipWith (*) as bs)
fir ds regs x = (x +>> regs, dotp1 regs ds 0)
...
```

(a)

```
...
dotp2 as bs z = vfoldr (+) z (vzipWith (*) as bs)
iir ds regs x = ( (dotp2 regs ds x) +>> regs,
                  dotp2 regs ds x)
...
```

(b)

Figure 6. (a) Partial description of the FIR filter [5] . (b) Partial description of the IIR filter

The main functionality of the IIR and FIR filters are described in Figure 6 with *iir* and *fir* functions respectively. The rest of the code which is for defining data types and handling states is not shown for the sake of simplicity. For more details about details of the hardware design in CLaSH refer to [5]. *Zipwith* and *fold* functions are the main functions used as the operations on vectors in this example.

Figure 7 shows the description of a simple function named *dsp* that selects one of two different filter functions as its main function, based on the value of the first argument which is of type *Unsigned D4*. The returned item is a function itself and that is either *filterIIR* or *filterFIR*. These are not complete filter descriptions, they only represent the main combinatorial description for functionality of IIR and FIR filters. Type $(DatainT \rightarrow DataoutT)$ means that it is applied to an input of type *DatainT* and returns a value of type *DataoutT*. Parametrization and higher-order functions are exploited in this example. In this example, the *dsp*

```
...
dsp :: (Unsigned D4) -> (DatainT -> DataoutT)
dsp sel = case sel of
                0 -> filterIIR
                1 -> filterFIR
...
```

Figure 7. A simple reconfigurable DSP unit

function has a *polymorphic place* of type $(DatainT \rightarrow DataoutT)$ that can accept any *candidate object* of this type. There are two candidate objects here: *filterIIR* and *filterFIR* functions. The polymorphic place is mapped to a reconfiguration region and each of the candidate objects is mapped to a reconfiguration candidate. Figure 8 shows both simulation and implementation models for the *dsp* function.

As it is shown in Figure 8 the simulation model includes a MUX to select either *filterIIR* or *filterFIR*. Based on the value of the sel input, either *filterIIR* or *filterFIR* function is evaluated and sent to the output. Since at every moment only one of these filters is active in the system, instead of having both of them implemented on hardware, it is possible to program the active filter between *datain* and *dataout* on hardware at run-time when it is selected. With any change on *sel* input, the new choice overwrites the

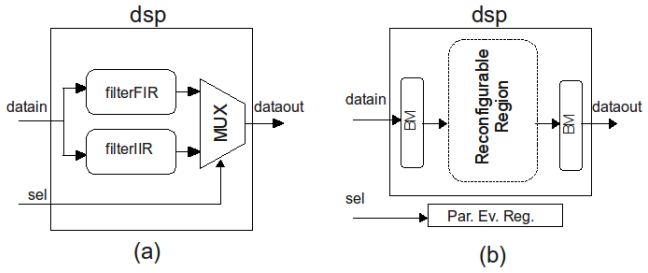

Figure 8. (a) Simulation model of the *dsp* function (b) Implementation model of the *dsp* module

old one on the hardware. For example when *sel* has value *'0'*, *filterIIR* is selected as the active filter and programmed into hardware. When it changes to '1', *filterFIR* is selected and programmed over *filterIIR* in hardware. In order to reconfigure the hardware at run-time we need a dynamic reconfigurable region which is shown in Figure 8(b). The process of reconfiguration is done by a processor in the system that writes bit-streams of selected modules into corresponding reconfigurable regions which is explained later in this section. The system is divided into two general parts, *dynamic part* that includes all reconfigurable regions and *static part* that is the rest of the system. The *sel* input is called *partial evaluation parameter* here since it is used to partially evaluate the MUX module.

In order to communicate with the rest of the system, every reconfiguration region should use special communication structures called *busmacros* (BM). Busmacros are fixed communication points between the reconfiguration region and the rest of the system. Every reconfigurable module that is programmed to a reconfiguration region must use busmacros to connect to the static part of the circuit. In each reconfiguration region, we instantiate as many busmacros as we need considering the communication ports of the reconfiguration candidates targeted for the reconfiguration region. The block diagram of the whole system is shown in Figure 9 which is our implementation platform. This is a self-reconfigurable system since there is an embedded processor on the FPGA that runs Linux and controls the whole reconfiguration process.

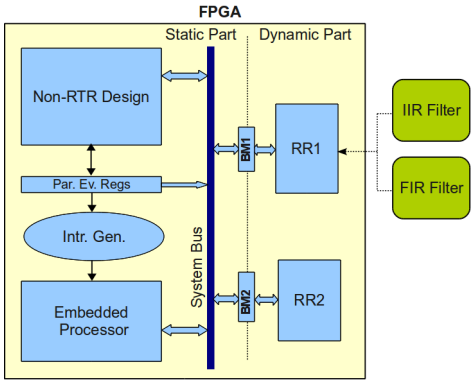

Figure 9. Realization of the DSP unit

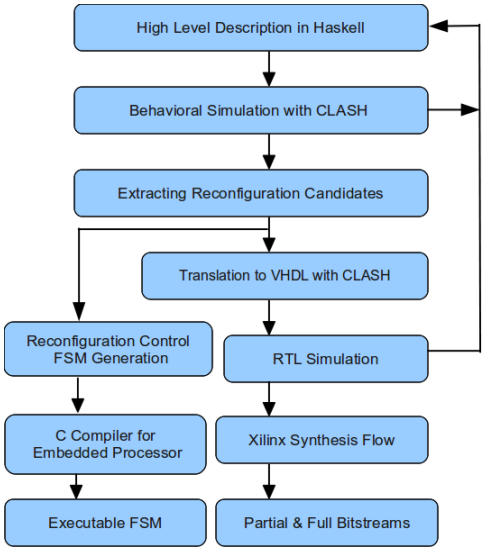

Figure 10. Main Design Flow

Configuration bits of every reconfiguration candidate is stored in the system and is loaded into *reconfiguration region* when it is needed. In this example, the *dsp* function is partially evaluated with *sel* input and the corresponding bit-stream is loaded into the target reconfiguration region. Any change on *sel* input leads to a reconfiguration of RR1 which is done by the embedded processor. The value of the partial evaluation parameter is kept in *partial evaluation register* which is readable by the processor. Any change in its value will generate an interrupt to embedded processor through the interrupt generation circuit. During the reconfiguration process, depending on the value of the register, a proper bitstream is loaded into the FPGA. In other words, the value of partial evaluation register, determines which configuration bitstream should be loaded into the FPGA.

The main design flow of this approach is shown in Figure 10.

After describing the design with high-level Haskell structures and expressions, all the reconfiguration candidates for each reconfiguration region is detected and extracted. For simplicity's sake we use only one reconfiguration region in this example. There are several important criteria for specifying reconfiguration candidates which are discussed in detail in [15]. After specifying the reconfiguration candidates (by specifying the partial evaluation parameter), a translation from Haskell to VHDL starts which translates Haskell functions into synthesizable VHDL entities using CLaSH tool. After the translation stage, all modules in the design are synthesized for the targeted FPGA using Xilinx ISE Tool. Each group of reconfiguration candidates are synthesized for a specific reconfiguration region inside FPGA, which is done by specifying special constraints including area and placement constraints.

Output of CLaSH includes VHDL files together with a constraint file which is also passed to Xilinx synthesis tools. The constraint file includes all constraints related to

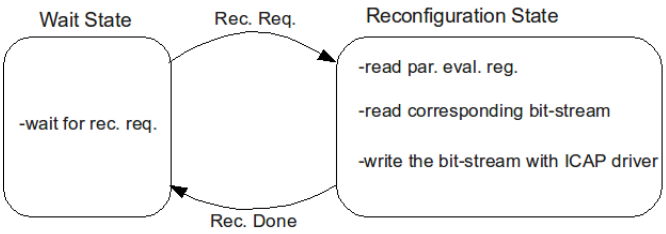

Figure 11. Simplified reconfiguration state machine

reconfiguration regions such as area constraints, placement constraints, and constraints related to busmacros. It also includes some basic information about the partially evaluated modules and their partial evaluation parameter for every reconfiguration region.

After generating the configuration bitstream of the design (including reconfigurable and static modules) configuration bits of every reconfiguration candidate is stored in a memory together with the configuration bits of the initial design. After loading the initial configuration bitstream to FPGA, any change on the partially evaluated parameter (e.g. *sel* input in the DSP example) will initiate a reconfiguration process. We use a specific C-program 'CLBRead' to readout partial bitstreams from a full bitstream. Partial bitstreams are saved in separate files which are accessible for embedded processor. Since we run Embedded Linux on the embedded processor, ICAP drivers for Embedded Linux are used to read and write partial bitstreams from/to FPGA at run time.

There is a state machine running on embedded processor on FPGA that controls the reconfiguration process based on different values of partial evaluation registers. A simple version of this state machine is shown in Figure 11.

We do not support totally automatic extraction of reconfiguration candidates yet. We use specific compiler annotations in CLaSH for detecting the reconfigurable module and different reconfiguration candidates. Furthermore, we have not included timing considerations in this design flow yet, but it provides a direct connection from high-level software-like design down to the underlying reconfigurable hardware which can be used for design space exploration on real hardware with real and accurate timing.

There are no specific types added to the Haskell description, so it is possible to use the same description for simulation and synthesis without any change.

We used Suzaku-sz410 board with Virtex4 FPGA, to test our ideas. Figure 12 shows the circuit implemented for the example discussed in this section. For the sake of simplicity we have only one reconfigurable region implemented on an FPGA here. The reconfiguration region includes *filterFIR* function programmed to it which will be replaced with *filterIIR* when it is called. It includes 3 CLB columns and every CLB column is formed of 16 configuration frames. Considering each frame to be 1312 bits, the partial bitstream will be 3*16*1312 = 62976 bits. The static part of the circuit includes hardcores like PPC, MAC and several controllers and peripheral circuits for DDR, UART, LCD, etc, which

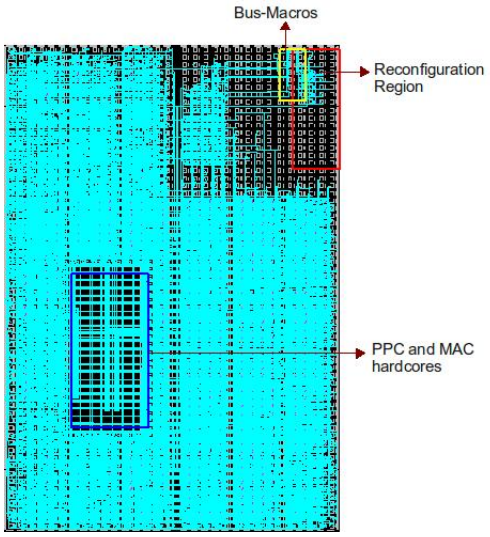

Figure 12. The DSP module implemented in FPGA

make it possible to run the Embedded Linux on FPGA.

V. Conclusions and Future Work

A modeling approach for run-time reconfigurable systems using Haskell is proposed. In order to decrease the productivity gap, we need to challenge higher levels of abstraction in the design flow. We use a functional HDL approach which uses Haskell as the description language to model RTR systems. High-level and abstract concepts such as *parametrization*, and *higher-order functions* together with *partial evaluation* implementation technique, are used for this purpose. Because of the parallelism in functional programming languages, modeling run-time reconfigurable systems is much different from imperative high-level languages. For proof of concept, a simple design is implemented on Suzaku-sz410 board. Including reconfiguration timing considerations and estimations for power, resource usage and performance can be useful for design space exploration in higher levels of abstraction. Since Haskell represent the mathematical-logical description of a system, it is also a better foundation for formal verification of run-time reconfigurable systems. This will be investigated in the future.

References

[1] Marian Adamski. Design of reconfigurable logic controllers from hierarchical UML state machines. *Electronics and Applications*, pages 82–87, May 2009.

[2] Carsten Albrecht, Thilo Pionteck, Roman Koch, Erik Maehle, and Ratzeburger Allee. Modelling Tile-Based Run-Time Reconfigurable Systems Using SystemC. *ECMS2007*, 4, 2007.

[3] Carlo Amicucci. SyCERS: a SystemC design exploration framework for SoC reconfigurable architecture. *ERSA2006*.

[4] Kenji Asano, Junji Kitamichi, and Kenichi Kuroda. Dynamic Module Library for System Level Modeling and Simulation of Dynamically Reconfigurable Systems. *Journal of Computers*, 3(2):55–63, February 2008.

[5] C.P.R. Baaij. CLasH: From Haskell To Hardware. Master's thesis, Univ. of Twente. *Master Thesis*, 2009.

[6] C.P.R. Baaij and et al. CLaSH: Structural Descriptions of Synchronous Hardware using Haskell. *EUROMICRO*, 2010.

[7] T Beierlein and D Fröhlich. UML-based co-design for run-time reconfigurable architectures. *Languages for system specification*, 2003.

[8] Alisson V. Brito, Matthias Kuhnle, Michael Hubner, Jurgen Becker, and Elmar U. K. Melcher. Modelling and Simulation of Dynamic and Partially Reconfigurable Systems using SystemC. *IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)*, pages 35–40.

[9] Fabio Cancare. Modeling Methodologies for Dynamic Reconfigurable Systems. *MSc Thesis*, 2008.

[10] Stephen Carven. A High-Level Development Framework for Run-Time Reconfigurable Applications. *MAPLD*, pages 1–10, 2006.

[11] Florian Dittmann. Design of Partially Reconfigurable Systems: From Abstract Modeling to Practical Realization. 2006.

[12] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An Overview of Reconfigurable Hardware in Embedded Systems. *EURASIP Journal on Embedded Systems*, pages 1–19, 2006.

[13] M.E.T. Gerards and et al. Hiding State in CLaSH Hardware Descriptions. *IFL*, 2010.

[14] Chun-hsian Huang and Pao-ann Hsiung. UML-based hardware/-software co-design platform for dynamically partially reconfigurable network security systems. *2008 13th Asia-Pacific Computer Systems Architecture Conference*, pages 1–8, August 2008.

[15] A. Pelkonen, K. Masselos, and M. Cupak. System-level modeling of dynamically reconfigurable hardware with SystemC. *Proceedings International Parallel and Distributed Processing Symposium*, 2003.

[16] Yang Qu and Kari Tiensyrjä. System-level modeling of dynamically reconfigurable co-processors. *FPL04*, pages 881–885, 2004.

[17] Imran Rafiq Quadri, Samy Meftali, and Jean-Luc Dekeyser. MARTE based modeling approach for Partial Dynamic Reconfigurable FPGAs. *2008 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, pages 47–52.

[18] Andreas Raabe. Describing and Simulating Dynamic Reconfiguration in SystemC Exemplified by a Dedicated 3D Collision Detection Hardware. *PhD Thesis*, 2008.

[19] A. Schallenberg. Dynamic Partial Self-Reconfigruation: Quick Modeling , Simulation , and Synthesis. *PhD Thesis*, 2010.

[20] Afreen Siddiqi and Olivier L. de Weck. Modeling Methods and Conceptual Design Principles for Reconfigurable Systems. *Journal of Mechanical Design*, 2008.

[21] Kamana Sigdel, Mark Thompson, Andy D Pimentel, and Todor Stefanov. System-Level Design Space Exploration of Dynamic Reconfigurable Architectures. *SAMOS'08*, pages 279–288, 2008.

[22] Martin Streub, Carsten Riedel, and Christian Haubelt. System Level Modeling and Performance Simulation for Dynamic Reconfigurable Computing Systems in SystemC. *GI/ITG/GMM-Workshop*, pages 59–68, 2007.

[23] K. Tiensyrja and J.-P. Soininen. SystemC-based Design Methodology for Reconfigurable System-on-Chip. *8th Euromicro Conference on Digital System Design (DSD'05)*, pages 364–371.

[24] Chih-hao Tseng and Pao-ann Hsiung. UML-Based Design Flow and Partitioning Methodology for Dynamically Reconfigurable. *Ifip International Federation For Information Processing*, pages 479–488, 2005.

[25] Jorgiano Vidal, Florent De Lamotte, Guy Gogniat, Jean-philippe Diguet, and Sebastien Guillet. Dynamic applications on reconfigurable systems : from UML model design to FPGAs implementation. *DATE2011*, pages 2–5.

[26] Jorgiano Vidal, Florent De Lamotte, Guy Gogniat, Jean-philippe Diguet, and Philippe Soulard. UML design for dynamically reconfigurable multiprocessor embedded systems. *DATE2010*.