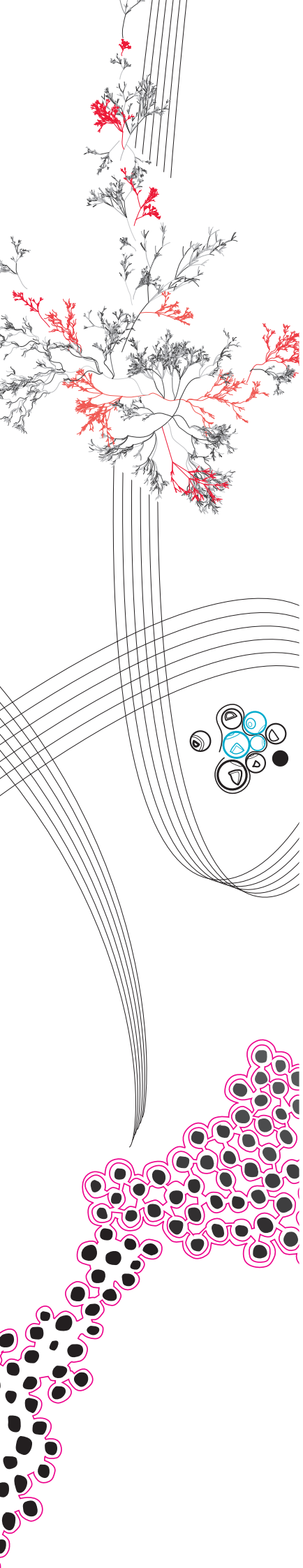


MASTER THESIS

OFFLOADING HASKELL FUNCTIONS ONTO AN FPGA

Author:
Joris van Vossen

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Computer Architecture for Embedded Systems (CAES)

Exam committee:
Dr. Ir. C.P.R. Baaij
Dr. Ir. J. Kuper
Dr. Ir. J.F. Broenink

December 2, 2016

UNIVERSITY OF TWENTE.

Abstract

Hardware and software combined designs have proven to be advantageous over the more traditional hardware- or software-only designs. For instance, co-designs allow a larger trade-off between hardware area and computation time. To date, translating a high-level software design to hardware is often performed from the imperative programming language C, thanks to its wide usage and its readily available compilers. The functional programming language Haskell, however, has shown to be a better alternative to describe hardware architectures. In the computer architecture for embedded systems group (CAES) at the University of Twente, the Haskell-to-hardware compiler *CλaSH* has been developed. This thesis aims to provide a well-defined basis and a proof of concept for semi-automating the process of offloading a subset of Haskell functions, from an arbitrary Haskell program, onto reconfigurable hardware, called a field-programmable gate array (FPGA).

Significant amount of research has already been done on HW/SW co-design in general and several tools are available, however, a Haskell specific co-design approach has not been extensively researched. During the research phase of this thesis some related work regarding co-design workflows was analysed and optimal solutions for fundamental implementation problems were found.

A design space exploration has been performed to realize a design that featured the optimal solutions. It was decided to focus on the following two use scenarios: semi-automated offloading and manual offloading. The main reason for this division was that a pipelined utilization of offloaded functions is more effective, but deemed too complex to automatically generate from a standard software-only Haskell program. Some of the prominent design choices were: a system-on-chip implementation platform that contained an ARM processor and an FPGA, a FIFO buffer based interconnect, and the use of a Haskell compiler plugin to automate the function offloading process. The implementation phase required a significant portion of the thesis' time to fully implement the proof of concept on the chosen platform, of which the result has been briefly described in the thesis and a user manual has been written.

Besides proving that automatic offloading of specific Haskell functions is possible, the results of benchmarking the proof of concept also show that only large enough Haskell functions will truly benefit from offloading, which is even more prominent in the case of non-pipelined utilization of offloaded functions in the automatic offloading proof of concept.

Therefore, the main conclusion was that automatic function offloading in its current implementation is possible but not very efficient for HW/SW co-design and that therefore either future work is required on automatic pipelining of Haskell programs or, as seems more appropriate, the focus should be shifted to the manual function offloading approach that allows for more design freedom. Lastly, the dataflow support in CλaSH should be expanded more to allow offloading of more complex *Signal* based functions.

Acknowledgements

This thesis is the final phase of my Embedded Systems MSc. study at the University of Twente. During this study I gained a lot of knowledge on embedded systems, which I already found very interesting during my Electrical engineering HBO bachelor. For this thesis I had help from multiple persons, whom I would like to thank.

First of all I would like to thank Jan Kuper and Christiaan Baaij, for offering me this master thesis project in the first place, and for providing support and extensive feedback. I really enjoyed working with CλaSH and hope to keep using it in the future.

Furthermore, I would like to thank my entire exam committee for providing support and time during this graduation.

Robert, Hermen, and Arvid, thanks for being my roommates in your respective periods. I really enjoyed the conversations about holidays, hobbies, and technical things.

John, I also would like to thank you for the extended cooperation before and during this master. I really appreciated our discussions on our work and the daily conversations.

Bert Molenkamp, for always having time for checking & updating my individual study program and other general problems related to the embedded systems master.

All members of the CAES group, thanks for the conversations, frequent cake bets, and the time together during the coffee breaks and lunch walks.

And lastly, I would like to thank my family for their support during my master and this thesis. Even though they did not really understood everything from my master study, I still really appreciated their look on things.

Joris,
Hengelo, December 2016

Contents

Acronyms

ARM	Advanced RISC Machine, a processor architecture.
AXI	Advanced eXtensible Interface; protocol for interconnects in a SoC.
CAES	Computer Architecture for Embedded Systems group
CλaSH	CAES Language for Synchronous Hardware
CPU	Central Processing Unit
DMA	Direct Memory Access
FIFO	First in, First out; A method to organize a data buffer between a source and sink.
FPGA	Field programmable Gate Array
GHC	Glasgow Haskell Compiler
GHCI	Interactive GHC environment
GSRD	Golden System Reference Design
HDL	Hardware Description Language
HPS	Hard Processor System; The ARM processor on the SoCKit
HW/SW co-design	A combined design of hardware and software components
IO	Input and/or Output
IP core	Intellectual Property core, a hardware design created and licensed by a group.
OS	Operating system
SoC	System on Chip
SSH	Secure shell; Used for (remote) network connections.
SystemVerilog	Extension of Verilog HDL
TH	Template Haskell; Used for compile-time code execution.
Verilog	Verilog Hardware Description Language
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1 Introduction

Algorithms, with the right properties, can have a faster and more efficient implementation with the aid of hardware accelerators than is possible in a software-only implementation on a general-purpose processor (e.g. the Intel Pentium), as these accelerators sometimes allow a more parallel representation of the algorithm. There are several potential properties of an algorithm that will prevent us from implementing it in a more parallel fashion. A data dependency between two steps of an algorithm, for instance, is one of these properties. If we take a multiply-accumulate (MAC) operation as an example, which could arbitrarily be used within an algorithm, then the multiplication of this MAC operation has to take place before the accumulation of the values can be performed. It is therefore often a challenge to identify if and where an algorithm would benefit from a more parallel implementation on a hardware accelerator.

When designing a system that will feature hardware acceleration, we can resort to either an application-specific integrated circuit (ASIC) design or implement the accelerator in reconfigurable hardware. Designing and manufacturing an ASIC is often a long process and is only cost-effective in larger production volumes. Using reconfigurable hardware instead, such as a field-programmable gate array (FPGA), is a more cost-effective approach to realize hardware accelerators.

In this thesis the focus lies on the investigation and development of a solution for embedded system designers which allows the design, rapid prototyping and realization of hardware accelerators on an FPGA. In the next sections it will become clear why it is necessary, what the proposed solution would be, and the involved problems.

Within the context of this thesis the term 'Function offloading' is used to indicate the process of transforming a part of an algorithm with potential for acceleration, to a hardware representation and integrating it with the remaining software implementation. Different design methodologies are necessary when applying function offloading or when creating a hardware/-software combined design (HW/SW co-design) in general [1]. In addition to new methodologies, it also requires the knowledge of both hardware- and software-only design principles. To date, a great amount of the embedded system designs were realized in a hardware- or software-only engineering approach. Among these new HW/SW co-design methodologies is a system-wide design space exploration (DSE), where the goal is to find the optimal partitioning of hardware and software. Another methodology is the use of co-simulation [4], where concurrent execution of software and hardware components in a simulation environment is possible.

Due to technological evolution, Systems-on-Chip (SoC) solutions have become more prominent. These SoCs, which consist of several types of processing architectures, are a useful platform for realizing HW/SW co-design. Function offloading, for instance, can be implemented on a SoC containing a combination of a processor and an FPGA. These SoC subsystems are connected via complex interconnects [2]. The prominently available SoC solutions for this thesis' implementation of function offloading, are integrated with a processor based on the ARM

architecture. To this day the ARM processors remain the industry leader in embedded and application specific computing. However, when looking at the roadmap of Intel, which is also an industry giant for processors, it becomes apparent that it is a critical part of their growth strategy [3] to integrate an FPGA in their x86-instruction set based processors in the future. With these increasing amount of SoC solutions, a HW/SW co-design engineering approach has become more accessible and, with the rising need of complex system designs, it will also become a must in the future [1]. In order to make HW/SW co-design more accessible and, subsequently, allow for the use of the associated design methodologies, a way to describe a system at a high-level is desired.

There are several tools available that can transform C-code to hardware in order to realize HW/SW co-designs. Examples of these tools are those included in the design suites of the major FPGA manufacturers Altera and Xilinx [5]. This process of interpreting a high-level algorithm and transforming it to a hardware description with the same functional behaviour is called High-level synthesis (HLS). As shown by Smit et al. [8], in comparison with the often used imperative programming language C, the purely-functional language Haskell [6, 7] is more suitable for describing and evaluating algorithms, as it is very close to both mathematical specifications and architectural structures on an FPGA. Therefore, in a Haskell program, it is easier to reason about exactly which functions are most suitable for hardware acceleration. For these reasons, it would be more appropriate to use Haskell as the input language for the HLS transformation and more importantly function offloading. One particular "Haskell-to-Hardware" compiler is in development at the Computer Architecture for Embedded Systems (CAES) group at the University of Twente and it is called CλaSH (pronounced 'clash') [9, 10]. It beholds a functional hardware description language that borrows its syntax and semantics from Haskell. In essence, it is able to transform a subset of the high level Haskell functions to a low-level hardware description language (HDL) such as VHDL, Verilog or SystemVerilog. Therefore having the possibility to create HW/SW co-designs from a Haskell description would also be a desirable addition to the CλaSH compiler.

1.1 HW/SW Haskell example

Before describing the proposed solution, we will look at an example of a Haskell description that proves to be a good target for function offloading. The example in question is about an Nth-order finite impulse response (FIR) filter as given in the discrete-time Equation 1.1. Essentially, each output value of such a FIR filter is a weighted sum of the last N input values.

$$y[t] = \sum_{n=0}^{N} b_n x[t-n] \tag{1.1}$$

A FIR-filter is often used as a low-pass frequency filter. A practical application, for instance, is to filter out the carrier frequency component of a received signal in Gaussian frequency-shift keying (GFSK) demodulation[11]. A partial Haskell description of the GFSK demodulator with a FIR-filter can be seen in Listing 1.1. For demonstration purposes we will merely focus on the FIR-filter as the remainder of the GFSK demodulator description is trivial.

```
1   module Demodulator where
2
3   import ...
4
5   demodulate input    = output
6       where
7           filterPass  = mapAccumL fir (...) input
8           output = ...
9
10  fir :: (Vec n (a)) -> a -> (Vec n (a), a)
11  fir states input    = (states', dotproduct)
12      where
13          states'     = input +>> states -- shift new input in states
14          dotproduct  = foldl (+) 0 (zipWith (*) coeffVector states)
```

Listing 1.1: A Haskell demodulator exemplary description using the FIR filter function

The *fir* function, as given in Listing 1.1, includes the *foldl* and *zipWith* higher-order functions. A higher-order function is, in general, a function that applies one or more argument functions onto its input data and/or internal states. In Appendix A, we can see the structural representation of the two higher-order functions in the *fir*.

The *fir* function begins with updating the state vector [1] *states'* by shifting the latest received input into the state vector from the previous execution. Subsequently, the *zipWith* applies the multiplication function in parallel onto the *states* and coefficient vectors, which results in a single vector of products. The *foldl* function then uses the addition argument function to sum the products vector.

The *fir* function requires n multiplications and $n-1$ additions (without any optimisations). A structural hardware representation of the FIR filter is given in Figure 1.1. We can notice how closely related the Haskell description is to the structural representation as the row of multiplications are identical to the *zipWith* function and the additions are closely related to the *foldl* function.

The FIR filter is a good example of a function that can be offloaded onto an FPGA as it can be computed in a more parallel fashion as all the costly multiplications can be performed at once and the additions in a tree-like fashion. Subsequently, we can also optimise the function further by either implementing it as a transposed variant, or by reducing the number of multiplications if the coefficients are symmetrical, or by implementing it as a multiplier-less implementation (i.e. only bitwise shifting and additions).

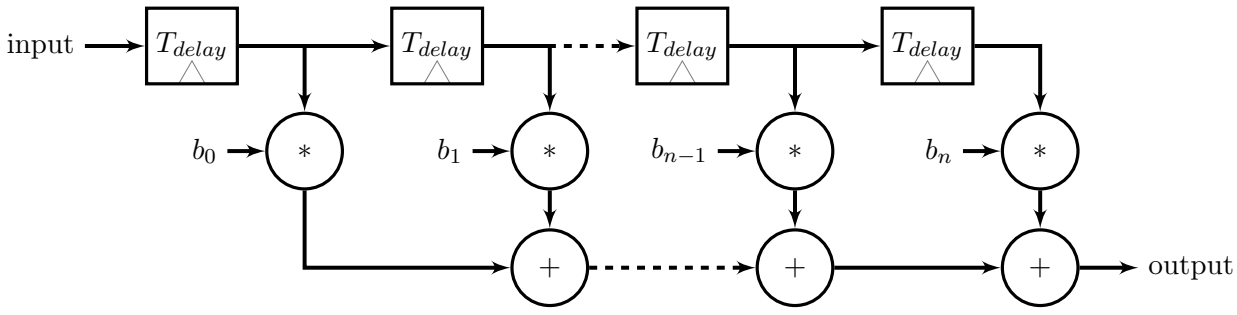


Figure 1.1: A hardware representation of the FIR-filter function used in the demodulator example.

[1] A vector in CλaSH represents a fixed-size list of data and the associated type notation is: *Vec (length) (data type)*.

The effectiveness of function offloading this FIR-filter function is primarily depending on characteristics such as the required bit-resolution and the order of the filter itself. From this GFSK demodulator example, it turned out that certain Haskell descriptions can definitely benefit in terms of speed and power consumption if they are implemented in a HW/SW co-design.

1.2 Assignment

With the example of section 1.1 in mind, we can formulate our main problem as follows: "How can we automatically offload selected functions of a Haskell program onto an FPGA and subsequently call upon them remotely from the remainder of the Haskell program?". The automation of function offloading can be interpreted in more than one way, for instance, it can mean that it should automatically analyse the bottlenecks in a Haskell program and subsequently select, and offload, the optimal functions. In this thesis, however, the assumption is made that the user will select the offloadable functions themselves and therefore automatic function offloading strictly implies that a user specified set of functions in a Haskell program are automatically offloaded onto the FPGA. The proposed solution in this thesis project is to develop a tool, which is to be used as a proof of concept, for offloading one or more functions from a Haskell program onto an FPGA. Offloading a complete Haskell program, however, is not the goal of this project as it would impose that it becomes a hardware-only design instead of the intended HW/SW co-design. Additionally, it can be argued that a hardware-only design is already possible by using CλaSH.

The ideal solution would be that we merely have to annotate[12][2] the function within the original Haskell program and that a compiler will offload the functions automatically. This will require a framework that can automatically generate the FPGA architecture according to the offloaded functions. Depending on the used SoC solution, it will require additional manual steps, such as separately synthesizing the generated hardware description files and programming the associated devices. In Listing 1.2 we can see how our previous *fir* function only has to be annotated within the GFSK demodulator Haskell description.

However, this automated way of function offloading inherently has a reduced performance due to the latency in the communication between the hardware and software partitions. This problem can essentially be solved by transforming the Haskell program into a pipelined co-design, which is something that will become clear later on. Automating this pipelining transformation is considered to be too complex for this thesis and therefore an additional approach for a more manual way of function offloading will also be included. It will allow for a less restricted HW/SW co-design in comparison to the automated solution, which will become clear later on in this thesis. Within the manual solution, we will have direct access to some of the underlying functions used in the automated offloading process. For instance, this manual approach allows for the FIR-filter example to be implemented in a pipelined fashion, which can result in a co-design with a higher throughput.

[2]Similar to the *#pragma* for a C-code compiler.

```
1 ...
2
3 {-# ANN fir Offload #-}                    -- Example annotation
4 fir :: (Vec n (a)) -> a -> (Vec n (a), a)
5 fir states input    = (states', dotproduct)
6     where
7         states'     = input +>> states
8         dotproduct  = foldl (+) 0 (zipWith (*) coeffVector states')
```

Listing 1.2: Annotated FIR filter function example

The main problem may be divided in two partitions in order to realize the two previously mentioned solutions. The first part is related to the design and realization of the interconnect between the two SoC components and the additional HW/SW required on either sides of the connection to make manual function offloading possible. The second part is the actual automation of the offloading procedure. The following research questions are introduced for the first part.

- How does function offloading, or in general High-Level Synthesis (HLS), work in other programming languages and how does it relate to Haskell?
- How can the offloaded functions on the FPGA be called upon remotely?
- What type of communication would be the best for function offloading, when also keeping a platform-independent implementation in mind?
- How can offloading of Haskell functions be implemented on the chosen development board?
- What, if any, restriction apply to offloadable functions?

It is imperative that the above research questions are answered first, as automating the offloading process will build upon it. This second part will introduce the following questions:

- How can a function be altered at compile-time within Haskell for the purpose of function offloading?
- How can the process of offloading Haskell functions be automated?
- What additional requirements will this automated process impose on the HW/SW co-design?

The result of answering all the previous questions will be used to implement the proof of concept to solve the main problem in this thesis. As this implementation itself is also a HW/SW co-design itself, we will also apply a system-wide design space exploration to find the optimal solutions. This proof of concept is in conclusion also benchmarked to document the performance difference between a HW/SW co-design implementation and a traditional software-only design.

1.3 Overview

This thesis has started with describing the main problem, its context, and the necessity for a solution. An example was given in order to introduce the problem and the subsequent proposed solution. In the second chapter, related literature is reviewed and background information is given. This mainly includes the literature on combined hardware and software design and on distributed computing in Haskell. The rest of the second chapter is dedicated to providing more background information that is used in the subsequent chapters. This second chapter and Appendix B can be considered as the result of the *final project preparation study* as required for an Embedded Systems master at the University of Twente.

The third chapter beholds the design overview and the design space exploration that is performed in order to find the optimal design for the proposed solution. Here, the usage scenarios are defined, which are used to perform the subsequent design space explorations. Many of the explored topics are concluded by means of a scoring table containing the possible solutions. Chapter four describes the actual implementation of the previously proposed solution. It uses the design choices and considerations from chapter three to achieve the optimal solution. The fifth chapter contains a brief listing of the requirements for HW/SW co-design in Haskell and a brief user manual for the implemented proof of concept. Chapter six describes the results of this thesis. It includes a section on verification of the implementation and on performance benchmarks.

In chapter seven the problems encountered during the work for the previous chapters will be discussed and the relevant conclusions are drawn. It also points out several recommendations for future work.

2 Background

This chapter beholds a brief literature review and relevant background material on subjects related to this thesis. Some of the research questions imposed in chapter 1 are either partially answered or in full. These solutions and the provided background information form the basis on which the design space exploration is performed as described in chapter 3. The first literature subject is on hardware/software co-design within several programming languages. We will subsequently use this reviewed literature and two practical applications to come up with a workflow for co-design in Haskell. The second literature review is focussing on parallel computing within Haskell and how it can be partly used for our solution. These subjects are reviewed to identify optimal solutions for this thesis' implementation problems and to prevent making similar mistakes found in the literature.

2.1 HW/SW Co-design

Combined hardware and software design, as already introduced in chapter 1, is an interesting practice when using a high-level functional language like Haskell. A lot of research has already been done on HW/SW co-design in other programming languages and several tools have been developed. Teich published a paper on the past, present, and future of HW/SW co-design [1]. He states that current co-design tools are still not fully mature as they fail to both overcome the complexity wall and fail to handle runtime adaptability that is introduced with complex heterogeneous multi-core processing chips. This statement does not necessarily apply to the proposed solution in this thesis, as it specifically focusses on a SoC with a single FPGA and CPU. However, in subsequent work related to this thesis, it will most likely have to be considered in the future. Teich also shows the importance of methodologies such as design space exploration, co-simulation, and co-synthesis in HW/SW co-design. They allow for an optimized result due to the possibility of early trade-off analysis and reduced development time. These methodologies are also in some form applicable to a Haskell HW/SW co-design process, which is something that will become clear later on.

In chapter 1 it has been described that CλaSH focusses on hardware-only designs. Related work on the topic of hardware descriptions in functional languages has been done for several years with results such as muFP[13], Hydra[14] and Lava[15], but a specific focus on hardware and software combined design in functional languages is a relatively lesser researched topic. Mycroft et al. [16] propose an implementation that can transform a program in their own functional language *SAFL* to a hardware and software partition. In contrast to the intended purpose of this thesis, they target the software partition to run on the FPGA fabric in custom generated soft processors. This approach allows for a larger design space that can be explored by the exploiting of the area-time spectrum[1]. However, designing a custom soft processor that can execute Haskell functions is not really feasible, as Haskell is significantly more complex than

[1] A trade-off between the number of logic gates and amount of clock cycles required to achieve a result.

the *SAFL* language. An additional reason to not use a soft processor is that their operating frequency is significantly lower than a hard processor.

Mycroft made the assumption that the user will explicitly define the hardware and software partitions themselves. As we have already described in chapter 1, we will also make this assumption as it is not the intention to automatically partition and optimize a Haskell program with respect to the area-time trade-off, which is a very complex task.

With the knowledge gained in this brief review on the current state of HW/SW co-design, we can begin analysing available HW/SW co-design tools for their workflow on function offloading.

2.1.1 SoC Co-design workflows

Two practical cases of HW/SW co-design workflow will be analysed to get a better understanding of how co-design can be applied using Haskell. Many of the existing solutions for HW/SW co-design on SoCs are designed for imperative programming languages like C and Matlab [5]. In the following two paragraphs we analyse a model-based Matlab workflow and one for the C programming language.

MathWorks workflow The first analysed case is a co-design workflow proposed by MathWorks in Matlab & Simulink as it provides a more abstract model-based approach with their HDL coder and HDL workflow advisor tools [17]. This model-based co-design, in contrast to language-based co-design, has the advantage of a strong mathematical formalism that is useful for analysis and realisation of the co-designed system [1]. As mentioned in chapter 1, the functional language Haskell is also closely related to mathematics and so it can be said that Haskell has similar advantages. Matlab is still an imperative language underneath the model-based design and therefore a HW/SW co-design approach based on the purely functional language Haskell will remain interesting.

MathWorks provides several co-design example applications that are ranging from simple LED blinking to advanced image processing [18]. All these examples start with a Simulink model. The process of offloading a subsystem of the model onto the FPGA requires a workflow that is summarized in List 2.1.

1. The preparation of the SoC hardware and installation of the associated tools.
2. A design space exploration on a system level, leading to the partitioning of the design for a hardware and software implementation.
3. Generation of the IP core in the HDL workflow advisor.
4. Integration of the IP core in an HDL project and programming the FPGA.
5. Generation of a software interface model.
6. Generation and execution of C-code on the ARM in order to interface the Simulink model on the host computer with the IP core in the FPGA.

List 2.1: MathWorks co-design workflow

For the simple LED blinker example the resulting co-design will then look like Figure 2.1. The blue block represents the embedded CPU, the green arrows represent the interconnections, the dark orange block features the programmed FPGA fabric, and the white block is the host computer running the Simulink model.

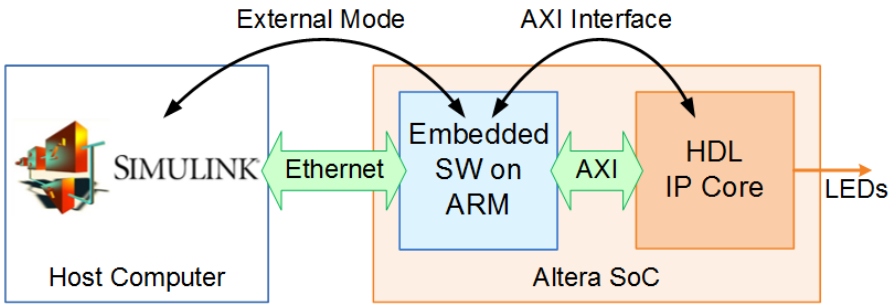


Figure 2.1: Resulting HW/SW co-design for the LED blinker example Matlab application [18].

This workflow imposes that the MathWorks tools 'HDL coder' and 'HDL workflow advisor' use the Simulink model to automatically configure and generate the interconnection and the additional hardware and software necessary on the SoC. As can be seen in List 2.1 and Figure 2.1, a part of the Simulink model still runs on the host computer. This may be undesired depending on the application requirements. A potential solution is to run the remainder of the model on the SoC by generating C-code in Simulink.

Xillybus workflow The second analysed HW/SW workflow case is a C-language based co-design workflow. It is proposed by Xillybus, which is the designer of an interconnect managing IP core that is used in this thesis [19]. The workflow has quite some similarities to the Matlab example, but the main difference in contrast to the model-based workflow is that it requires considerably more user interaction. The second workflow goes as follows:

1. The workflow starts with a C program `prog` that contains a function `f` that is the target for offloading onto the FPGA. This function is to be manually separated from the main program file and inserted into a new file.

2. Some modifications should then be applied to this new file, such as compiler pragmas that indicate the input and output ports as used by a high-level synthesis (HLS) tool.

3. Provided that the implementation platform is ready and the associated tooling is installed, then the user will start compiling the function `f` with the HLS tool and include it within an updated version of the demonstration HDL project that is made available with the Xillybus IP block. This demo project features both a Direct Memory Access (DMA) and a First In, First Out (FIFO) types of data communication on the interconnection between the FPGA fabric and the HPS. For demonstration purposes, the DMA method will suffice, but additional hardware and software may be necessary for more complex applications.

4. Finally the user has the task to alter the original program `prog` such that it remotely calls upon the offloaded function instead of the original version.

List 2.2: C language based co-design workflow by Xillybus

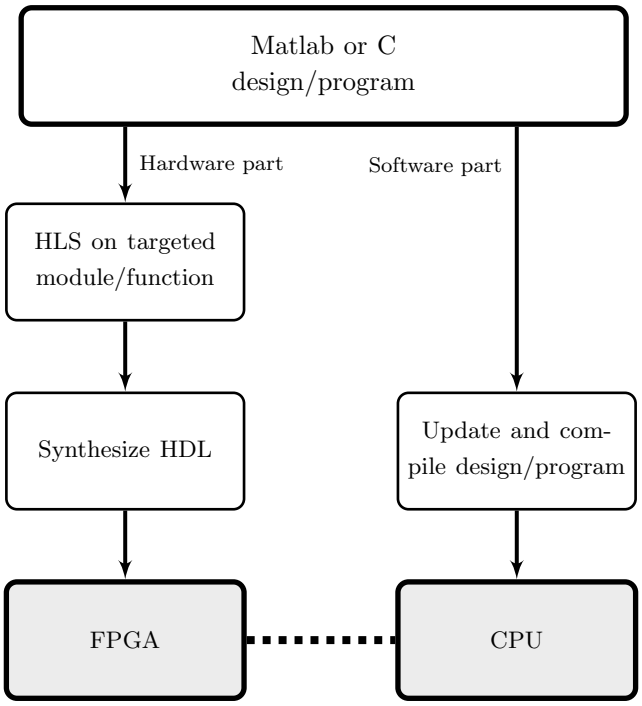


Figure 2.2: Workflow summary for hardware/software co-design in a Matlab and a C based approach.

Both the Matlab and C based examples can be generalized as pictured in the diagram in Figure 2.2. Both workflows still require the user to generate or even create and integrate the additional software and hardware on either side of the interconnect. Some restrictions on the offloadable functions will be required in order to allow a more automated workflow. In the next section we will propose a workflow for function offloading in Haskell, but first we will have to get a basic understanding of describing hardware in Haskell.

2.2 Proposal of HW/SW co-design in Haskell

With Listing 1.1 and Figure 1.1 we already saw an example of how Haskell is closely related to a hardware representation. We will use the solutions found in the previous literature review to propose a workflow for HW/SW co-design in Haskell.

Traditionally, Haskell descriptions are only executable on a processor architecture and not on an FPGA. In order for a Haskell function to become implementable on an FPGA it should be rewritten to a representation that is compilable with the CλaSH compiler. CλaSH is only able to interpret a subset of Haskell functions in order to transform a design into a hardware description with the same functional behaviour. These CλaSH compilable Haskell functions can be translated to either only pure combinational logic or into synchronous sequential logic, such as a Mealy machine [23, 24] as shown in Figure 2.3. In contrast to combinational logic, the output of sequential logic depends not only on the present value of the input signals but also a set of previous input and intermediate values. These past values are stored in registers, which update their values based on a clock signal.

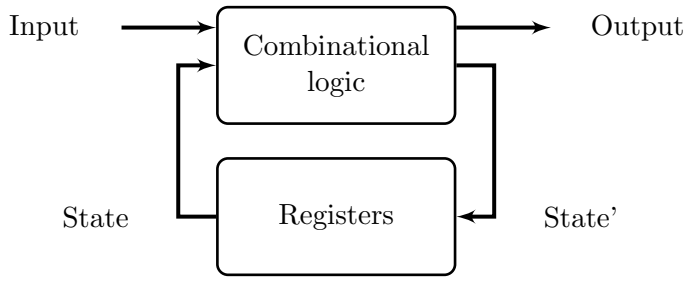


Figure 2.3: Mealy machine hardware representation

The function *f_mealy* in Listing 2.1 is a Haskell representation of a Mealy machine as shown in Figure 2.3. Similar to the figure, it can be seen that it essentially wraps state registers around a trivial combinational logic function *f_comb*. A major advantage of allowing these types of sequential functions to be offloaded onto the FPGA in addition to standard combinational functions, is that we can keep the state register data locally stored. This reduces the amount of data that has to be transferred between the CPU and FPGA.

```
1 f_mealy :: s -> i -> (s, o)
2 f_mealy state input             = (state', output)
3     where
4         (state', output)    = f_comb state input
```

Listing 2.1: A Mealy machine in Haskell

In CλaSH the primary method of describing and simulating synchronous sequential logic is by writing it as a *Signal* type based representation. These *Signal* type based functions essentially use a standard Haskell input list as the implicit clock signal and for each element in that input list will produce the corresponding output element in the *Signal* type based output. However, as shown in the next part of this section, we cannot easily offload these *Signal* based functions without additional modifications.

In order to interface combinational on the FPGA with a program on the CPU (which essentially is a very complex sequential circuit), it has to be transformed to sequential logic itself. Additionally, the registers in this sequential logic should only be updated if correct input data is available. These two problems are solved by using dataflow principles, which is something that will be described in more detail later on. These dataflow principles mainly allow the combinational and sequential logic to only produce a valid output if a new correct input value is received from the CPU. Currently the dataflow composition implementation in CλaSH is limited to the following types of functions that are not *Signal* based:

- Pure function: `i -> o`
- Mealy function: `s -> i -> (s, o)`
- Moore function: `(s -> i -> s) -> (s -> o)`

However, going back to the *Signal* based functions, only possible to offload such a function if it already adheres to the dataflow principles and therefore can be lifted to a dataflow typed function. However, for automatic function offloading it would impose that beforehand the software-only design of the Haskell program also has to deal with the dataflow principles, which

undesirably leads to additional overhead and design complexity. As discussed later on, this can still be used in the manual function offloading use scenario.

Finally, it is also possible to combine multiple of the dataflow typed functions to create a more complex dataflow function, i.e. a multi-component design such as the GFSK demodulator from section 1.1. Similar to the previously mentioned way to offload *Signal* based functions, it requires the original software-only Haskell program to deal with dataflow principles and therefore we will also only allow this for manual function offloading and not for the automated variant.

With this gained knowledge we can state that a function which is a potential candidate for offloading must, at least, adhere to the following criteria:

- A function has to be (re)written in the sub-set of Haskell functions that CλaSH supports.
- A Haskell function should either be written as a pure[3], Mealy, or Moore function as described in the CλaSH Prelude[24]. As later becomes clear we may also manually create combinations of these three functions.
- The function has to conform the rules and limits for compiling in CλaSH [25], e.g. the function may not contain infinite data sets or have unbounded recursion[26].
- It must not exceed any limit of the targeted hardware platform. This includes resource limits such as required logic blocks, embedded memory, clock trees and any other dedicated blocks.

List 2.3: Fundamental requirements to allow function offloading in Haskell.

2.2.1 Proposed workflow

An overview of the workflow we propose for the process of offloading one or more functions of a Haskell program is pictured in Figure 2.4. The overview can be described in the following steps:

1. It starts with an arbitrary Haskell program. Verification of this Haskell program is then recommended before proceeding, which can be achieved by static analysis or by simulation within the interactive compiler environment (GHCI) or a custom testbench.
2. Subsequently, the designer can start selecting functions for offloading by means of a design space exploration. These functions should meet the requirements described in List 2.3 and so it might be the case that they have to be altered. At this point, simulation of the hardware partition with the software part may be performed in Haskell to verify and analyse the complete system. In addition, co-simulation may also be possible by simulating the hardware partition in a HDL simulator [4].
3. Depending on the use scenario, which may happen either by hand or by means of an automated process, the targeted functions have to be separated from the rest of the Haskell program. After this any necessary updates to both partitions should be performed. These updates are essentially the additions of hardware and software necessary to let the two partitions communicate with each other.

[3]Pure means the function output is only influenced by the most recently received input, i.e. no state registers.

4. These updated partitions are then separately compiled to HDL and an executable respectively. The HDL files are then added to a template project and synthesized to an FPGA programmable file.

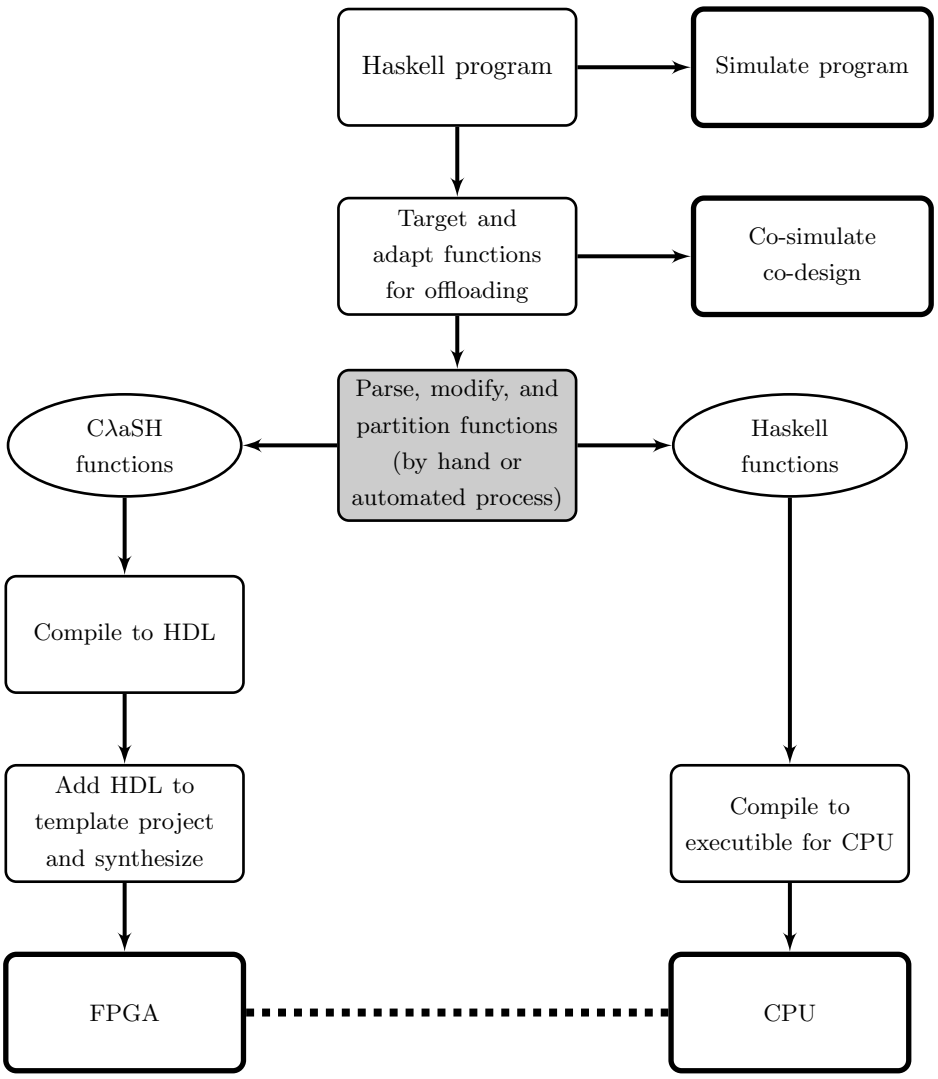


Figure 2.4: Workflow for offloading Haskell functions onto the FPGA.

In the next section we will explore how to fully automate the process of function offloading according to our proposed workflow.

2.3 Automation of function offloading

Fully automated function offloading is essentially the process that performs all the steps starting from the annotated offloadable functions in the grey block from Figure 2.4, without interaction of the user. Automating the step of parsing, altering and partitioning the targeted offloadable functions is paramount for full automated offloading. It is assumed that the subsequent steps of compiling, synthesizing, and programming can be performed through means of a non-trivial

scripted process and therefore our focus in this section lies on the initial step as pictured by the grey block in Figure 2.4.

In section 1.2 it was already mentioned that a framework has to be designed in CλaSH , which can generate the hardware partition for the FPGA, by merely passing the targeted offloadable functions as input arguments. For the software partition we will need to modify the original Haskell program such that the targeted offloadable functions are replaced by functions that will interface with the offloaded function in the hardware partition.

Partitioning and modifying a Haskell program at compile-time is possible in two main ways, which are both directly related to the phases in which Haskell compiles a given program [27]. The process in which GHC compiles a Haskell program to an executable file consists of the steps shown in List 2.4. The listing briefly describes what the function is of each phase and which in- and output type the phase has.

1. **Parsing** (`File/String -> HsSyn RdrName`)
   In this first step the Haskell program file is interpreted and parsed into an expression with the abstract syntax required for the next phase.
2. **Renaming** (`HsSyn RdrName -> HsSyn Name`)
   In this phase all of the identifiers are resolved into fully qualified names and several other minor tasks are performed.
3. **Typechecking** (`HsSyn Name -> HsSyn Id`)
   The process of checking that the Haskell program is type-correct.
4. **Desugaring** (`HsSyn Id -> CoreExpr`)
   Transforming the typechecked Haskell source to a more computer friendly Core program.
5. **Optimisation** (`CoreExpr -> CoreExpr`)
   This phase is used to optimize a Core program.
6. **Code generation** (`CoreExpr -> Desired output`)
   The final Core program is compiled to either an intermediate interface file or executable code depending on if the initial Haskell source is imported or the top-level module.

List 2.4: Haskell compiler phases

The two possible ways to partition and modify a program at compile time are before the *parsing* phase and by means of a Core plugin [29] at the start of the *optimisation* phase. Modifying before the parsing stage implies that a Haskell program file (which contains all the targeted offloadable functions) has to be modified before it is parsed by the Haskell parsing phase of the GHC compiler. To achieve this, a pre-compiler has to be built which consists of the phases shown in List 2.5. It first parses the Haskell program file to an expression tree that can be easily modified. Subsequently, the partitioning and modifications are performed and finally the resulting two expression trees have to be unparsed into Haskell program files again, such that it can be compiled into a software and a hardware partition with respectively the GHC and CλaSH compilers.

1. **Parsing** (`String -> ExprTree`)
   Phase that parses a Haskell program string into an expression tree with the type of a subset of the Haskell grammar.

2. **Modifier** (`ExprTree -> (ExprTree,ExprTree)`)
   The process of partitioning the hardware and software parts happens here. Subsequently the software part is also modified to interface with the hardware partition.

3. **Unparsing** (`((ExprTree,ExprTree) -> (String,String)`)
   The last phase constructs the new Haskell program strings of the modified expression trees, which results in two new files that can be used to compile both co-design partitions.

List 2.5: Phases of the proposed Haskell pre-compiler that allows for compile time program modifications.

The second approach, building a Core plugin, requires knowledge of the *Core* type to which all of Haskell gets compiled, as shown in the last three phases of List 2.4. As can be seen in Listing 2.2, the explicitly-typed Core is quite small as it does not require any syntactic sugar for human readability. The exact details are not important here, but a simple example increment function in the left column of Table 2.1 will result in the relatively large *Core* tree expression in the right column.

```
1  type CoreExpr = Expr Var
2
3  data Expr b                              -- "b" for the type of binders,
4      = Var      Id                        -- Variables
5      | Lit    Literal                     -- Literals
6      | App    (Expr b) (Arg b)            -- Type abstraction and application
7      | Lam    b (Expr b)                  -- Value abstraction and application
8      | Let    (Bind b) (Expr b)           -- Local bindings
9      | Case   (Expr b) b Type [Alt b]     -- Case expressions
10     | Cast   (Expr b) Coercion           -- Casts
11     | Tick   (Tickish Id) (Expr b)       -- Adding Core information
12     | Type   Type                        -- Types
13
14 type Arg b = Expr b                      -- Top-level typed expression
15 type Alt b = (AltCon, [b], Expr b)       -- Case alternatives
16
17 data AltCon = DataAlt DataCon | LitAlt  Literal | DEFAULT -- Alt constructor
18
19 data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))] -- Top-level and local
       binding
```

Listing 2.2: Entire Haskell Core type[28]

Standard Haskell	Core expression equivalent

`increment :: Int -> Int`
`increment x = x + 1`
	`Main.increment :: GHC.Types.Int -> GHC.Types.Int`
	`[GblId, Arity=1, Str=DmdType, Unf=OtherCon []]`
	`Main.increment =`
	`  \ (x_sxV :: GHC.Types.Int) ->`
	`  let {`
	`  sat_sxW [Occ=Once] :: GHC.Types.Int`
	`  [LclId, Str=DmdType]`
	`  sat_sxW = GHC.Types.I# 1 } in`
	`  GHC.Num.+ @ GHC.Types.Int GHC.Num.$fNumInt x_sxV sat_sxW`

Table 2.1: Simple increment Haskell function and its Core expression equivalent.

Modifications to a *Core* expression can be done during the optimisation compiler phase with the previously mentioned Core plugin. When a Haskell module has been compiled to the *Core* expression representation, then by default, modifications can only be performed using the in-scope *Core* expressions. This is a limitation of the current Core plugin implementation, however as it is not a fundamental problem, it may be removed in the future. This limitation prevents us from constructing our replacement function without its environment already being in-scope. As will become clear later on in the thesis, it requires quite some work to construct whole functions within a Core plugin.

Besides making modifications, we also need to partition the *Core* expression into a hardware and software part. There are two ways to achieve this by means of a Core plugin:

- We can choose to have a separate Core plugin for the hardware and the software partition compiler. This means that two versions of the plugin have to be designed.
- We can limit ourselves to a single Core plugin for the software partition and then, similar to the pre-parser approach, we will generate a Haskell module file that can be used to generate the hardware partition.

In summary, both approaches for compile-time modifications are good candidates, but each have their own advantages and disadvantages. In the next chapter we will perform a design space exploration on these two solutions, which will mention these (dis)advantages.

In the next section we will look at distributed computing in order to find possible solutions that can be used to realize our proposed solution for the main problem.

2.4 Distributed computing in Haskell

Offloading Haskell functions onto reconfigurable hardware is closely related to the field of parallel programming. Haskell itself has wide support for pure parallelism and explicit concurrency. An example of related work is distributed computing, in which an algorithm is executed on a network of computers. Distributed computing allows for explicit concurrency, but it also introduces other characteristics such as asynchronous communication between distributed functions and independent failures of components. Cloud Haskell [20, 21] is a practical example of concurrent distributed programming in Haskell. It essentially provides a programming model and implementation to program a computer cluster as a whole, instead of individually. This model is based on Erlang [22], which is an industry proven programming language for massively scalable and reliable soft real-time systems. The programming model features explicit concurrency, has lightweight processes with no shared states, and has asynchronous message passing between the processes.

A basic communication topology in Cloud Haskell can be described as a Master-Slave pattern, in which the master node controls the other slave nodes in the computer cluster. The master node has the task to create and send a Haskell function closure[2] to start a process on a slave node. Cloud Haskell makes use of serialisation (i.e. a bytestring) to send a function closure to other nodes. This approach imposes that the slave node should be able to receive and execute the function closure during runtime. If this would be implemented on a slave node that contains reconfigurable hardware, then the slave or its master node has the task to reconfigure the respective hardware at runtime. Doing this often on an FPGA is not very desirable as the function closure has to be translated into a hardware description and subsequently synthesized to a programmable bitstream, which are time consuming tasks. In addition, translating a function closure into a hardware representation might not even be possible at all times, e.g. if the closure's environment contains additional functions that are not synthesizable. An original Erlang feature that allows for updates to functions closures at runtime on a node is not implemented in Cloud Haskell. Two potential solutions for offloading a function to an FPGA in distributed computing without the previously mentioned problems would be either to separately synthesize all potential functions for the FPGA beforehand or to configure the FPGA only at the start of the program, such that it will include all of the desired offloaded functions that are necessary during runtime.

Once a function closure is set-up as a process on the slave node, then the master node can call upon the function through message passing. In Cloud Haskell, this is achieved by using the *send* and *expect* functions that require a unique process identifier. Such an identifier is also a potential solution for identifying offloaded functions and their associated messages.

Cloud Haskell supports multiple types of network transport layers. In contrast to a computer cluster, that often communicate asynchronously through TCP connections on a local network, a system-on-chip platform has dedicated interconnections that are designed for high performance and may feature direct memory access (DMA). This allows for a more application specific implementation and, unlike the asynchronous message passing approach of TCP communication, it can perform more efficient as it does not necessarily require additional buffering and sorting of messages.

In summary, some design choices in Cloud Haskell can also be used for solving the problems faced in this thesis, as will become clear in chapter 3. And in chapter 7 we will discuss the combination of Cloud Haskell and our proof of concept implementation.

This background chapter has been about the review of literature and related work to find optimal solutions for the design and implementation of the proof of concept for function offloading in Haskell. In the next chapter we will use this background information during the design space exploration and the subsequent creation of the proof of concept design.

[2] A closure is a record storing a function with its environment.

3 Design

In the previous chapters we described the thesis' main problem and answered a number of the research questions through literature reviews and background information. A rough representation of the proposed solution was given in section 1.2. In this chapter we will apply this gathered knowledge in a design space exploration in order to put together the optimal design for the proposed solution.

3.1 Design considerations

A few use scenarios and their requirements have to be specified in order to perform the design space exploration successfully. Due to the analysis of the HW/SW co-design workflows of related tools and the literature review in chapter 2, we can describe the generalised use scenario for this project to be that the user desires to offload one or more functions of a Haskell program onto the FPGA such that it will be more efficient. We will divide this main scenario in an automated, a semi-automated and a manual scenario, such that the user will have the possibility to optimize their HW/SW co-design in a more manual way if deemed necessary.

`Fully automated offloading` - This allows the user to automatically offload functions in a Haskell program by merely annotating them, as shown in the example function in Listing 1.2 of chapter 1. An automated process will then perform the steps of HW/SW partitioning, applying modifications, compiling, synthesizing, and programming without assistance of the user.

`Semi-automated offloading` - Here the user will still be able to annotate the functions for offloading. The main difference with the full automated scenario is that only the steps of HW/SW partitioning, modifying, and compilation of the software partition are automated. The user then has to manually compile the hardware partition to a HDL, which is subsequently synthesized within a template HDL project to obtain the programmable bitstream for the FPGA. At this point, the user can execute the compiled Haskell program, which will now use the offloaded functions instead. As can be seen by this scenario description, it does not behave as a completely automated process.

`Manual offloading` - This scenario is similar to semi-automatic offloading except that the user will also manually partition and modify the Haskell program into HW/SW parts. This may be used to create a more efficient streaming application, something that will be described in greater detail later on.

As will become clear later, the fully automated offloading scenario is out of the scope of this thesis due to restrictions in the chosen implementation platform. However, it is assumed to be possible to implement this scenario in the future when a different implementation platform is

used.

3.2 Design space exploration

With the use scenarios defined, we can now begin exploring the design space to find the optimal solutions for this thesis. Some of the major design space explorations are explained by means of a scoring table. The score for a category ranges from worst to best by means of the following ordinal scale: $--$, $-$, 0, $+$, and $++$. In the last column of each table is the weighted scoring summation given to indicate the best solution[1]. This total score ranges from an *1* (worst) to a *5* (best).

The exploration starts in the next section on what implementation platform is to be chosen. Subsequently, the exploration continues on how to implement the interconnection between the CPU and FPGA on the chosen platform. This is followed by an exploration on design options related to the CPU implementation. And finally, the implementation considerations for the hardware partition are described. This chapter ends in an overview of the made choices and the resulting design, such that they may be used in the next chapter.

3.2.1 Implementation platform

In this section we briefly explore the platforms on which we can implement our proposed solution. One of the prominent platforms is a System-on-Chip solution. There are multiple SoC platforms available on which this thesis project can be realized. A particular SoC platform was already regarded as a potential candidate, namely the SoCKit development board. A technical overview of the SoCKit and some related background topics can be found in Appendix B. To summarize the appendix, the SoCKit is a board that contains the Cyclone V SoC as pictured in Figure 3.1. This SoC combines an FPGA and an ARM processor (which we may refer to as the Hard Processor System (HPS)) through means of a configurable high-speed interconnect. The appendix shows that Haskell programs can be compiled and executed on the HPS. In Appendix B it has been shown that a Haskell compiler should be installed on the HPS itself instead of using an often quicker cross-compiler. This former option allows the use of dynamic-loading and Template Haskell, which, as will become clear later on, are paramount for this thesis.

[1]Total row score is calculated by summing the product of the ordinal score and its weight in each criteria and dividing it by the sum of the weights.

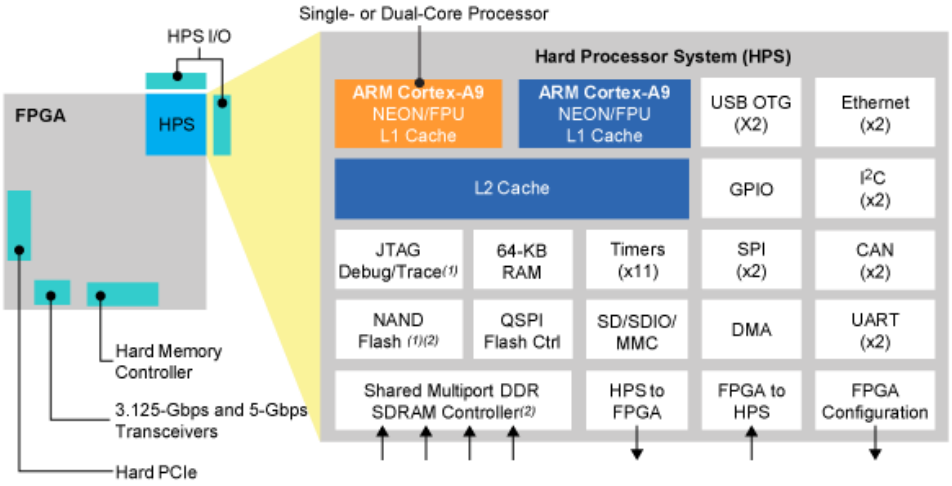


Figure 3.1: Altera Cyclone V System-on-Chip overview [32]

By means of the technical review in Appendix B, it is estimated that the SoCKit platform can be used to implement our proposed solution of section 1.2. In the rest of this section we will use the background information gathered in Appendix B to perform the design space exploration on the implementation platform.

In Table 3.1 we see the results of the exploration. The first solution is the SoCKit, which is a combination of an FPGA and a hard processor on a single chip. We have already discovered that the processor is fully capable to run Haskell programs and that it is a desired platform for function offloading by embedded systems designers that target low-power or applications specific designs.

Implementation platform	**Haskell execution capable**	**Offloading effectiveness**	**Develop time**	**Score**
Weighting	*3*	*2*	*1*	
SoC with FPGA & CPU	++	++	+	4.83
FPGA with soft processor	−−	0	−−	1.67
CPU & PCIe FPGA	++	+	++	4.67

Table 3.1: Design space exploration on the implementation platform.

A potential alternative solution is using only an FPGA with a soft processor inside its fabric, which was something that was already hinted at in the literature research on the work of Mycroft et al. Soft processors potentially have the advantage that additional instructions can be added and so the effectiveness of function offloading can argued to be less or even unnecessary. However, existing soft processors such as the MicroBlaze of Xilinx and the NIOS II of Altera, do not have support for Haskell and designing a new custom soft processor ourselves is out of the scope of this thesis.

The final prominent alternative is a standalone FPGA connected to a processor. A practical example of this is an FPGA that is interfaced through a PCI express bus with a high-performance processor based on the x86 instruction set. Haskell has been primarily designed for this type of processor. In addition, the synthesis tools for FPGAs are only available for this type of processor and so only with this solution can we adhere to the full automated offloading

scenario. In contrast to the ARM processor on the SoCKit, this type of processor is often not intended for low-power or application specific designs. For this reason we will favour the SoCKit for function offloading effectiveness.

In conclusion, the SoCKit remains the optimal choice. The CPU and standalone FPGA combination is a very prominent alternative and shares some resemblance with the SoCKit. It is therefore estimated that in the future the SoCKit implementation should be relatively easily adapted for this alternative.

3.2.2 SoCKit Interconnect

Now that we are determined to use the SoCKit development board, we can continue the design space exploration with the possible implementations of the interconnect on SoCKit. This interconnection between the FPGA fabric and the HPS can be utilized in multiple ways, as seen in subsection 3.2.1. It already became clear during the related work analysis in chapter 2, that the default method is to generate it with the Altera provided tools, i.e. Qsys and Embedded Design Suite. In order to implement the semi-automatic offloading scenario for more than one function, the interconnection has to be configured as an abstract channel such that it can transport multiple types of messages. As shown in section 2.4, Cloud Haskell uses serialization combined with a protocol to communicate with the remote functions.

Designing and managing the interconnect on the FPGA and the CPU partitions is not a trivial task when considering that an efficient buffering of data is likely to be required. For this reason, a design exploration was performed on alternative solutions that would do the task for us. In the limited time for this exploration only one prominent alternative was found, namely the Xillybus IP core [40]. The advantages that Xillybus provides are the following main features:

- The IP core comes with an Ubuntu Linux image with a working package manager.
- No need to develop hardware drivers for the ARM. All communication with the FPGA is done by plain user-space programming with standard file operator functions.
- It utilizes DMA buffers on the processor side to minimize data management overhead.
- The interconnection, including DMA buffer sizes, can be automatically configured and optimized according to the expected bandwidth and desired latency.
- An almost identical Xillybus implementation is, for instance, also available for PCIe based FPGA and processor combinations. This allows future users to use this thesis' proof of concept without much effort on other implementation platforms.

Together with the advantages of the IP core and limited time for this thesis, it was chosen to use the Xillybus IP core instead of designing and managing the interconnect ourselves. It is assumed that this IP core can be replaced in the future with a user created interconnect that has a similar functional behaviour. In the following sections we will attempt to find the optimal configuration of the interconnection by first explaining the Xillybus IP core in more detail and then how it can be configured.

3.2.2.1 Xillybus IP core

The Xillybus IP core manages the data exchange between the FPGA and ARM processor. It is created by the company Xillybus Ltd., which also creates similar solutions for other implementation platforms. A graphical representation of how the IP core is integrated in the SoCKit

system can be seen in Figure 3.2. The Xillybus IP core uses several licensing formats [41]. This thesis is performed under the "Educational license" that allows free usage for study related projects. For commercial usage there are other licenses available such as a "Single copy" license that requires a 1,000 USD fee per IP core used. For this reason, the ideal end-point would be to not require this IP core for the offloading of the Haskell functions when using the SoCKit.

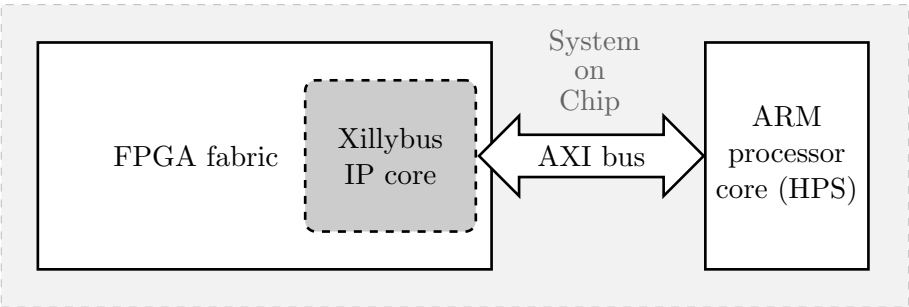


Figure 3.2: Overview of Xillybus IP core integration in the Cyclone V SoC.

The Xillybus IP core essentially consists of two parts: a Xillybus Quartus project that includes all the files necessary for programming the FPGA and an Ubuntu LTS 14.04[2] image for the HPS. The Xillybus IP core comes standard with a configuration for demonstration purposes. In order to configure the IP core according to application requirements, the user has to regenerate the core in the IP factory on the Xillybus website[43]. This process produces new HDL files to replace the old configuration in the Quartus project. By using the Xillybus IP core and the associated Ubuntu image, the device files related to the chosen configuration are automatically generated upon booting the OS. These device files can then be used in user-space programming languages to communicate with the FPGA. The Xillybus Quartus project is restricted to Quartus II version 13.0sp1, as newer versions have introduced changes to some IP cores that the Xillybus IP core relies on. Next we will perform a design space exploration to determine the best Xillybus configuration for our proposed solution.

3.2.2.2 Xillybus configurations

Several types of Xillybus IP core configurations can be generated on the Xillybus IP factory website in order to optimize it for an application. As can be seen in the Xillybus demonstration project, there are essentially two main types of configurations possible:

1. A direct address/data interface configuration, which resembles a direct memory access (DMA) for the CPU to a user defined memory block in the FPGA fabric.

2. A FIFO buffer configuration between the CPU and FPGA. The actual FIFO buffer is implemented in the FPGA fabric, but the CPU has a local DMA buffer to reduce data management overhead. The DMA buffer is then used by the Xillybus IP core for transferring data from or to the FIFO buffer.

These two options, which also can be used concurrently, also have several configurations themselves. The design space exploration begins by analysing these two main configurations. The first DMA method allows for configuring the address range, data width, and settings regarding desired latency and expected bandwidth. The FIFO buffer configuration includes, besides

[2]Manually upgraded to version 14.04

settings of the actual FIFO buffer in the FPGA fabric, options for controlling the direction of the buffer, the data width, the CPU DMA buffer page size (for streaming applications), and other settings related to latency and bandwidth.

The key difference between the DMA and FIFO configurations is the amount of overhead on the HPS. The DMA method requires the HPS to specify the right address in the memory block on the FPGA and will require asserting and polling of read and write flags in the memory to allow clock domain crossing of data, whilst the FIFO implementation on the HPS is just a write to local buffer and read from local buffer operation. Furthermore, the DMA configuration is not suitable for streaming applications, which is something that will be explained later, as it is not intended to buffer more than one data set without having to emulate a FIFO buffer in the FPGA memory. In that case, it would result in more overhead on the CPU for controlling the FIFO buffer structure in the FPGA memory in comparison with the standard FIFO configuration.

The DMA method also makes designing our FPGA architecture quite complex. Besides that the CPU will have more overhead in controlling the memory structure, the hardware architecture will also require automatically generated control logic for the shared memory in order to achieve the (semi-)automated use scenario.

An important characteristic of a FIFO buffer is its buffer size. Determining the buffer size is often a trade-off between meeting specific application requirements and using the least amount of required memory (which can then be used for other purposes). In our proof of concept there is no knowledge of how and what type of offloaded functions will be used and so an optimal FIFO buffer size cannot be chosen. A second prominent characteristic is the data type of an element in the buffer. In the Xillybus IP core the options are 8, 16, or 32-bits integers. The interconnection data width for the SoCKit is at most 128-bits as shown in Figure B.2, but this is not accessible within the Xillybus IP core. In order to get the most bandwidth, especially for streaming applications, the 32-bits configuration should be chosen.

The requirement that more than one Haskell function can be offloaded, introduces an additional solution for this particular topic of the design space exploration. This solution is to use a separate up- and downstream FIFO buffer pair for each offloaded function in order to prevent waiting for shared buffer queues when multiple functions were called from the CPU. Multiple pairs of FIFO buffers, however, will not work with the Xillybus IP block as it can't be directly reconfigured at compile time and thus failing the (semi-)automated use scenario. An argument for using separate FIFO buffer pairs is that it will allow each offloaded function to have its own clock domain, however this may also be achieved in the single FIFO solution if separate clock domain crossing logic is implemented before and after each offloaded function.

Xillybus configuration	**HPS overhead**	**Automatic scalability**	**Streaming efficiency**	**Design difficulty**	**Score**
Weighting	*2*	*2*	*1*	*1*	
DMA	−	+	−−	−	2.5
Single FIFO	+	++	+	++	4.5
Mult. FIFOs	++	−−	++	+	3.5

Table 3.2: Design space exploration for the Xillybus IP configuration.

In Table 3.2 a summary of the previous discussed points is shown. It clearly shows that a single FIFO configuration is the optimal solution for our design.

3.2.2.3 Message protocol

Using a single FIFO buffer for multiple offloaded function implies that the interconnection will require a protocol on top of the already serialized data. This protocol will need to have a header identifier to indicate the start of a message, a function identifier, and a data length indicator. In the literature research some of the related work use the Transmission Control Protocol (TCP) to transport its messages. This protocol uses sequential numbering and additional messages, such as the SYN-, FIN-, and ACK-flags, to reliably transmit over a network. These measures are necessary to prevent data loss as the used transmission channel is, in general, shared with other computers. The interconnect between the CPU and FPGA on the SoCKit, however, is not shared. This implies the reliability measures of the TCP-protocol are not truly necessary. Altera SoC FPGA devices are designed for high levels of error resilience by means of several techniques [44]. It is therefore assumed that the probability of errors during data transmission is almost neglectable. A disadvantage of using these additional reliability measures is that they introduces extra overhead on data transmissions.

For these previous reasons and because this thesis remains a proof of concept, we will not include these reliability measures and only implement a relatively simple protocol to indicate each message, its associated function, and data size. The chosen protocol itself is described later on in chapter 4.

In the next section the design space exploration on the processor is performed. It includes topics, such as the general software partition implementation, data serialisation, and automation of function offloading.

3.2.3 Hard Processing System

The next topics of the design space exploration are related to the hard processor system (HPS) of the SoCKit and the Xillybus IP core. Interacting Haskell user-space programming with the device files of the Xillybus IP core is rather straightforward. For this purpose, Haskell includes the *System.IO library*, which contains the *openFile*, *readFile*, *writeFile* and *hClose* functions. These are the Haskell equivalents of the *Posix.IO* library. The Foreign Function Interface (FFI) of Haskell can also be used to call C language functions directly. In both cases impure functions are introduced, which will introduce the IO Monad to the Haskell program. As this is impossible to be solved without rewriting the original given Haskell program, it is circumvented by using the *unsafePerformIO* function [42]. This exploit is regarded as a bad habit to have in the pure Haskell language, but this will be discussed later.

In order to transform an offloadable Haskell function into a function that calls the offloaded function in the FPGA architecture, it has to be replaced with a polymorphic function that has the behaviour as pictured in Figure 3.3.

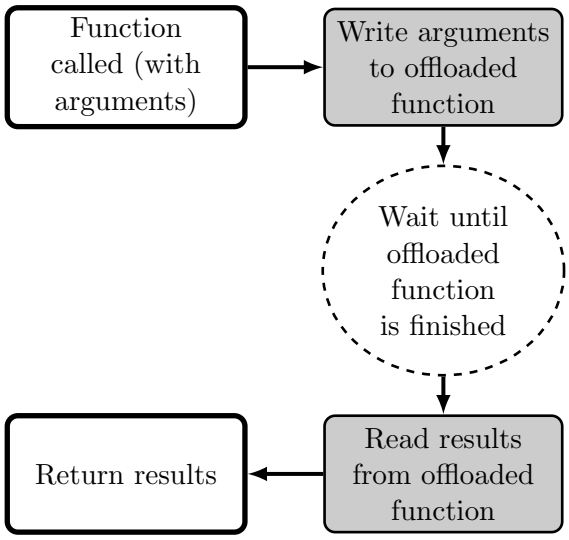


Figure 3.3: Flowchart of proposed behaviour of the replacement function for an offloaded function in Haskell.

Streaming applications, which can benefit significantly from hardware acceleration, can utilize the flexibility of the Xillybus IP core to operate more efficient by optimizing the data transmission flow[45]. Traditional programming data flow, such as given in Figure 3.3, where we call a function and then wait for it to return its value, is not an efficient way to call an offloaded function on the FPGA. Instead we want to separate the original program in two parts (i.e. threads) as shown in Figure 3.4, where the first thread only sends data to the offloaded function and the second thread receives and processes the result. Using this multi-threaded programming paradigm to pipeline the HW/SW co-design minimizes the hardware's latencies, since neither the writing or the reading partition of the original program is effectively waiting the latency time. Therefore, the throughput of the co-design will be increased.

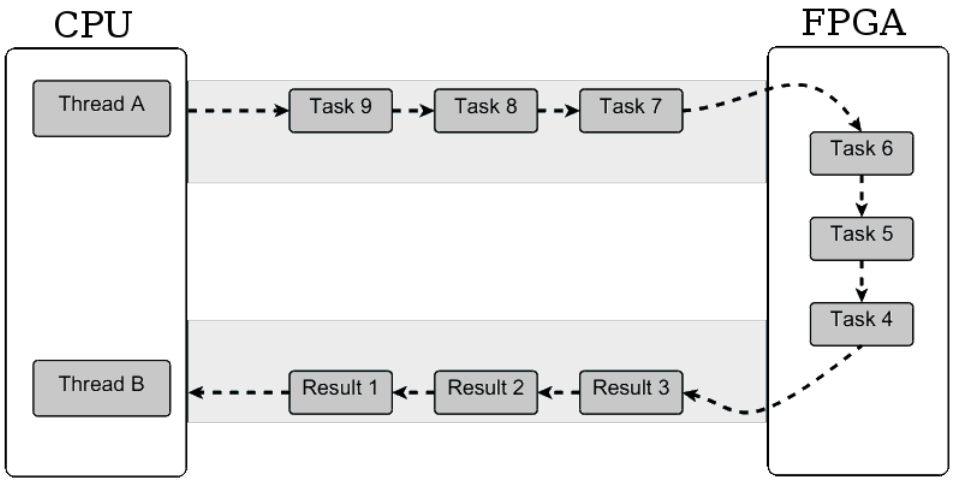


Figure 3.4: Optimal use of Xillybus IP core for streaming applications.

In the (semi-)automated offloading use scenario we are not able to implement this concept as we have no intention to automatically transform the Haskell program into a multi-threaded

representation. However, in the manual use scenario we can leave it up to the user to partition the Haskell program with this pipelined technique. Therefore, we need to give the user access to the separate *read* and *write* functions for interfacing with the offloaded functions on the FPGA.

3.2.3.1 Data serialization

On either sides of the interconnection data has to be transformed into messages that are ready for transmission over the FIFO buffers. The chosen Xillybus IP core configuration can be modelled as an abstract channel that can only transmit 32-bits words at a time. So a method of serializing any data set to a 32-bits word list is required to transport any type of argument or result data.

In the FPGA implementation this can be achieved by using a specific *Bitpack* package [46] in the CλaSH compiler. It is coincidently the optimal solution, because in an FPGA a parallel set of data can be directly interpreted as being serialised by means of correct wiring and multiplexers. However, this is not the case for a software implementation on the Haskell side of the interconnection, which will be our next design space exploration. Within Haskell there is a choice of also using the same *Bitpack* package or creating a custom serialising module that uses Haskell's own optimized serialization packages.

The name of the CλaSH package *Bitpack* already implies that it packs data to a vector of bits. This is useful in a hardware representation, but for a processor that has a 32-bits architecture, it means that serialization would involve a significant amount of bit operations. So essentially the design space exploration can be rephrased to if serialising at a bit level or at 32-bits word level (i.e. *Wordpack*) is more optimal.

Serialization method	**HPS overhead**	**Function restrictions**	**Implementation difficulty**	**Score**
Weighting	*1*	*2*	*1*	
Bit level	−	+	++	3.75
32-bits Word level	0	−	−−	2

Table 3.3: Design space exploration for serialization of function argument and result data.

In Table 3.3 is an overview of the serialization method design space exploration. The HPS overhead for serialization can be potentially reduced by restricting the offloadable function type to one that consists of byte or word sized data types common to a 32-bits architectural processor. Most of the Haskell programs are designed using these word sized data types, but it does restrict the offloadable function to a specific subset of CλaSH data types, which can be undesired in specific cases. For example, 32 booleans[3] can be combined in a single 32-bits word in the *Bitpack* method, but in the *Wordpack* method, which uses the standard Haskell serialization modules, it will be represented as 32 separate bytes (or possibly even in 32-bit words). This will cause the transported message to become considerably larger.

The *Wordpack* method also introduces a problem that potentially results in more program runtime overhead and makes automatic offloading nearly impossible without literally copying the internals of the *Bitpack* module. In the case of the *Bitpack* method, the serialization function and the protocol header is parametrized at compile time due to CλaSH requiring fixed size vector types. However, the default serialization modules of Haskell operate on unknown sized lists. This essentially means that the program has to perform additional computations to serialize the data and create the protocol header at runtime.

[3] A boolean can be either 'True' or 'False', which can be represented as a 1-bit data type.

The final argument in this topic of the design space exploration is the implementation difficulty. The *Bitpack* method is already available and is considerably easier to implement, as it is just a matter of adding the *pack* and *unpack* functions in the right stages of the polymorphic interfacing function as introduced in Figure 3.3. Besides having to create the *Wordpack* module itself, it also includes additional hardware on the FPGA side to filter some of the mentioned side effects.

The conclusion of this design space exploration is that the bit-level serialization provided by the *Bitpack* module in CλaSH is the optimal choice. Because we have chosen this solution an additional requirement, shown in List 3.1, is necessary to offload functions.

> A function that is to be offloaded should have an argument and result type that can have the BitPack[46] class constraint, i.e. the *pack* and *unpack* functions can be applied to the argument and result types of the offloadable function.

List 3.1: Additional requirement for a offloadable function imposed by the *Bitpack* module in CλaSH.

3.2.3.2 Automation of function offloading

In section 2.3 two potential methods for automatic function offloading were introduced and in this section these two options are explored and a choice is made for the optimal solution. The first method regarded creating a pre-compiler and the second approach was by using a Core plugin. The design space exploration overview can be seen in Table 3.4, which will subsequently be explained.

Automatic offloading method	**Implementation difficulty**	**User interaction**	**Extendibility**	**Score**
Weighting	*1*	*1*	*1*	
Pre-compiler	−	−	+	2.67
Core Plugin	−	++	0	3.33

Table 3.4: Design space exploration for two potential automatic function offloading methods.

Developing a Core plugin has the disadvantage that additional knowledge of the GHC API is required. As already mentioned in section 2.3 this involves the understanding of the Core type and how to modify it. The pre-compiler approach requires the implementation of a parser, tree modifier method and an unparser, which can all be done through standard Haskell programming. Even though only a subset of the Haskell grammar has to be parsed in order to complete its task, the pre-compiler still requires a significant amount of work in order to be implemented. The Core plugin uses the built-in Haskell parser. For these reasons, the implementation difficulty for both methods is estimated to be relatively equal.

Either method requires a basic set of user interactions and requirements that are listed in List 3.2.

- All the function offloading requirements listed in List 2.3 and List 3.1 are required.
- The functions have to be annotated, as shown in the example in chapter 1, such that they can be identified by the pre-compiler or Core plugin.
- Either the module that contains the offloadable functions should be completely compilable by CλaSH (e.g. no Haskell Prelude and no annotations) or the user has to copy those functions in a separate module. This is necessary to compile the FPGA partition, which is something that is shown in chapter 4.

List 3.2: User interactions and other requirements necessary to make semi-automated offloading possible.

In addition to List 3.2, both methods have their own additional requirements and user interactions. The pre-compiler method requires us to do an additional compiling stage, in which it generates updated Haskell files with the replaced offloadable functions and a top-entity Haskell module file that can be used for the FPGA architecture compilation. These slightly altered files are then to be compiled again with the normal Haskell compiler to run the program with offloaded functions.

The Core plugin method, however, only requires the user to add an additional compiler annotation to the module containing the offloadable functions such that the Core plugin will be applied to the compilation process of that module.

The last criterion in this topic of the design space exploration is related to how easily the methods can be extended in the case that Haskell itself or CλaSH is modified in the future. The pre-compiler will be most likely accessible to any user that has a basic understanding of Haskell and so intuition would say it to be the most extendible option. As previously mentioned the Core plugin requires the knowledge of the GHC API and Core type, but if that knowledge is present then it would not be much more complex to extend the plugin in comparison to the pre-compiler. As shown by the final scores in Table 3.4 the Core plugin method is the optimal solution.

3.2.4 DSE overview

At this point we have explored all the important design space exploration topics and for each topic the optimal solution was chosen. The prominent choices made were as follows:

- The SoCKit development board is chosen as the implementation platform.
- The Xillybus IP core is used to manage the interconnection configuration.
- The interconnect is configured to have a single up- and downstream FIFO buffers to communicate between the FPGA and CPU.
- Protocol checksum and additional message reliability measures were not deemed necessary for this proof of concept. Only a basic protocol is to be used.
- The *Bitpack* package from the CλaSH compiler is used to serialise data on the ARM.
- The Core plugin is used to automate the process of function offloading.

In the next section these choices will be included in the design of the proposed solution.

3.3 Design overview

Combining the background information in chapter 2 and the design space exploration choices made earlier in this chapter, we can describe the overview of the design. In section 1.2 we had chosen to divide the proposed solution in two partitions. The first part featured the fundamental basis that will allow for function offloading, whereof we can see the design in Figure 3.5, and the second part implements the actual automation of function offloading.

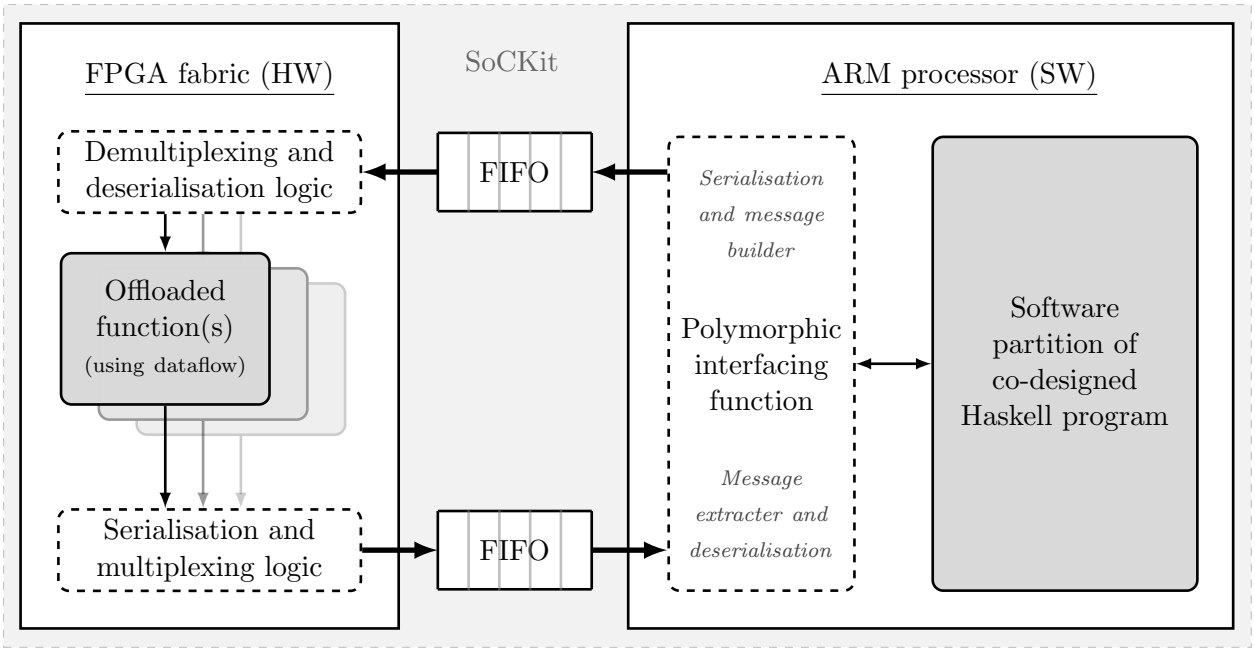


Figure 3.5: Design overview of the fundamental implementation of function offloading.

Figure 3.5 shows that the design consists of several components. The design shall be implemented on the SoCKit board that features the FPGA and ARM processor. The interconnect between these two components is configured as a up- and downstream FIFO buffer pair. As chosen in subsubsection 3.2.3.1, both the hard- and software partitions will use the *BitPack* class functions from CλaSH to serialize and subsequently deserialize data. Similarly, both partitions will also make use of the same protocol, as proposed in subsubsection 3.2.2.3, in order to associate data with the appropriate offloaded function. In subsection 3.2.3 we already saw the proposed design of the polymorphic interfacing function, which includes the steps of serialization, protocol application and sending and receiving of data. Additionally, this polymorphic function will also have a split version, i.e. a separate write and read function, such that it allows for the pipelined HW/SW co-design.

In combination with the previously mentioned design aspects, the FPGA architecture will also have the design requirements that are listed in List 3.3. When the FPGA architecture implementation meets these design requirements, then it will be usable for all the three use scenarios specified section 3.1.

- An FPGA architecture is required that can be generated in both the manual and semi-automated use scenarios by merely providing the list of offloadable functions. This imposes that it should be implemented in a CλaSH design.
- The architecture shall be connected to the read and write FIFO buffers as chosen in subsubsection 3.2.2.2. This allows the architecture to always keep its top-level input and output signals identical and thus allowing an automated generation process.
- The user should be able to either implement an offloadable function as pure, Mealy, Moore type, or a combination thereof, which, as described in section 2.2, can be achieved by using dataflow principles.
- A basic protocol error detection should be implemented to alert users if they have made mistakes in the manual use scenario. Error correction is not the goal for this proof of concept. In addition, a manual reset is also required on the FPGA implementation.

List 3.3: Requirements for the FPGA architecture design.

Now that the design of the fundamental basis for function offloading has been described, we can proceed with describing the design of the semi-automated process of function offloading that is primarily performed by the Core plugin as chosen in subsubsection 3.2.3.2. Essentially, this semi-automated process will be able to take a software-only Haskell design that meets specific requirements, and subsequently transform it into a HW/SW co-design, as pictured in Figure 3.6. The Core plugin is designed to output the altered software partition and also the top-entity module that is used to generate the hardware partition. After this process the right most block of this figure is essentially a parametrized equivalent of Figure 3.5.

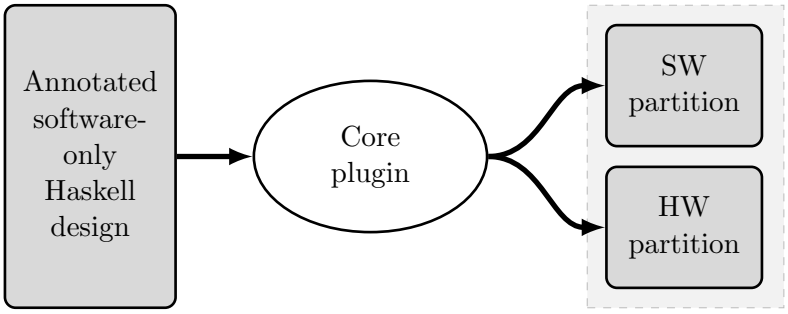


Figure 3.6: Design of the automated function offloading process.

In the next chapter the designs of Figure 3.5 and Figure 3.6 are implemented in order to realize the proof of concept as proposed in section 1.2.

4 Implementation

In the previous chapter we made the best possible choices in our design space exploration. These design choices were subsequently used to create a design for our proposed solution that we described in section 1.2. This chapter briefly describes the resulting implementation and how it may be used. It will also mention some of the major obstacles found during implementation and how they affected the final implementation. The majority of the required files to reproduce the exact implementation of this thesis can be found in the GitHub repository[53]. The associated code is intended to be mostly self-explanatory.

This chapter, regarding the implementation description, is divided in the three major partitions of the SoC platform, namely the CPU, FPGA, and the interconnect in-between them.

4.1 Interconnection

In chapter 3 we concluded that using the Xillybus IP core to manage the interconnect instead of building one ourselves was the best solution. The IP core has been configured with the IP Core factory[43], such that it facilitates the device files as listed in Table 4.1. Both the *xillybus_read_32* and *xillybus_write_32* device files represent a FIFO buffer that will be used for the communication between the CPU and FPGA. The IP core and the DMA buffers on the CPU have been automatically set by the IP Core factory for minimal latency and maximal bandwidth. The IP core factory indicates a maximum bandwidth of 200 MBytes/s for both the up- and downstream FIFO buffers. The dual-clock FIFO IP cores [47] in the FPGA fabric are configured to a buffer depth of 256 words. In our case, reasoning about optimizing the buffers (in order to reduce used memory elements) is not possible as there is no exact knowledge of the HW/SW co-designs of future users. Therefore, it is left up to the future users themselves to optimize the buffer sizes. In addition, the actual FIFO buffers are implemented as dual-clock IP cores. This allows the FPGA architecture to be manually set to a desired clock domain rather than the domain the Xillybus IP core utilizes.

Name	Direction	Data width	Details
xillybus_read_32	FPGA to host	32-bits	Used as FIFO
xillybus_write_32	host to FPGA	32-bits	Used as FIFO

Table 4.1: Xillybus IP core device files

In the previous chapter we have already read how we can connect the Haskell program and FPGA logic to the chosen interconnect configuration. In short, our custom FPGA logic is to be connected to the FIFO IP cores in the Quartus template project and the Haskell programming user-space can access the FIFO buffer device files with the file operation functions in the standard IO library.

4.1.1 Message protocol

As already mentioned in subsubsection 3.2.2.3, using a single FIFO buffer and offloading multiple functions requires the use of a protocol. The protocol is implemented as shown in Figure 4.1.

# Bits	32	32	32	32	32	32	32	32
Contents	Start flag	Function Identifier	Data length n	Data 0	Data 1	..	Data n-1	Data n

Figure 4.1: Protocol for data transmission through the interconnect FIFO buffers.

All data types of the protocol are of the FIFO 32-bit signed integer type, which we will refer to as the *XillybusType*. The *start flag* is a fixed value to indicate the start of a new message. The *function identifier* is used to identify which offloaded function the data belongs to. Additionally, the protocol is also used to specify the length of the data as required for the design and implementation of the FPGA architecture.

Additional hardware, or software, is required to transmit the data sequentially as was already seen in section 3.3. On the FPGA, for instance, the received data has to be deserialized to a parallel representation with the original argument type of the offloaded function. Once all sequential data has been received and made available in parallel, then it may be inserted in the offloaded function, as will become clear in the next section. After the offloaded function has produced a valid output, it will have to be serialized again in order to be transmitted back to the HPS. The same protocol in Figure 4.1 is also applied to the returning output data.

Although the data type of the function identifier and data length is a 32-bit signed integer, it is unrealistic and impractical to have a large amount of offloaded functions or extremely long data messages. However, as this is a proof of concept, these values will not be restricted. At the end of this thesis, the potential points for optimisation of this protocol are discussed.

4.2 FPGA architecture

When a Haskell function meets the prescribed requirements of List 3.1, it may be offloaded onto the FPGA. We will need to design an FPGA architecture based on data-flow principles, because the offloaded function should only be executed if a valid input argument is available. This data-flow approach allows for a pure, Mealy, or Moore function to be executed for exactly one clock cycle, which then produces the resulting output that has to be sent back to the Haskell program.

The data-flow type [48] is given in Listing 4.1 and the graphical representation can be see in Figure 4.2 as the central block. A dataflow function in CλaSH can essentially be described as a circuit with bidirectional synchronisation channels that functions as follows:

- The input valid signal has to be asserted before a data-flow function should consume input data, i.e. the input contains valid argument data.
- The data-flow function must only update its output when the incoming back-pressure ready signal is asserted, i.e. the hardware on the output is ready to receive the result data.

```
1 Signal i              -- Incoming data.
2 -> Signal Bool        -- Flagged with a single input valid bit.
3 -> Signal Bool        -- Incoming back-pressure, ready bit.
4 -> ( Signal o         -- Outgoing data.
5    , Signal Bool      -- Flagged with a single output valid bit.
6    , Signal Bool      -- Outgoing back-pressure, ready bit.
7    )
```

Listing 4.1: Data-flow type in CλaSH [48]

The higher-order CλaSH functions *pureDF*, *mealyDF*, or *mooreDF* can be used to create a dataflow representation of the offloadable functions. Additionally, if desired in the manual use scenario, the dataflow composition combinator function *seqDF* can be used to combine multiples of these three composition functions into a sequence of dataflow typed function.

Naturally, it is also possible for the user to create their own *Signal* based function which adheres to the dataflow principles and has the function type as was given in Listing 4.1. This custom function should then be lifted to the correct dataflow type using the *liftDF* before it can be used for function offloading.

Now that we have a data-flow representation of our offloaded function, we can begin to implement the serialization logic blocks around it. This can be divided in a deserializer and a serializer partition as was described in subsection 4.1.1. Such a deserializer block will transform the serialized data from the incoming FIFO buffer, which is received over the span of several clock cycles, to a parallel representation that is required by the data-flow function argument type. This is achieved by first writing each serialized data element into a subsequent element of a storage vector. Once this vector is full with all serialized data, then it will be deserialized into data with the argument type of the offloadable function, which is achieved by using the *unpack* function of the *BitPack* class, as chosen in subsubsection 3.2.3.1.

In turn, the deserializer block will do the reverse of the serializer to return the output data of the data-flow function over the interconnect. Both serialization logic blocks have been implemented as sequential logic in the form of a mealy machine, because the (de)serialization process requires a number of clock cycles that is equal or higher than the word length of the data in its serialized form.

Implementing such serialisation blocks in Haskell requires additional compile-time information as the compiler itself cannot calculate the serialized data length from merely knowing the original data type. This additional information is a set of 2 pairs of type-level naturals, which we will refer to as the *serialization naturals*. A pair of type-level naturals represent the total word size of the serialized data and the zero-bit extensions needed to achieve this word size from the original data type. An offloadable function with separate argument and result data types requires for each type a pair of type-level naturals. In the following example we find a practical case of what a pair in the *serialization naturals* can be:

> "The offloaded function's input data that has the type of *'Vec 5 (Signed 8)'*, i.e. 40-bits of data, has to be serialized into data with the type *'Vec n (Signed 32)'* such that it can be transmitted over the FIFO buffer (which acts as a 32-bits channel). As can be seen from this serialized type, the length n is unknown at compile-time. As we can only transport data in multiples of 32-bits, the original 40-bits argument has to be extended with several bits. The *serialization naturals* pair for this data type has to be: $\lceil 40/32 \rceil = 2$, and the number of zero-bit extensions that are necessary to achieve this word size is: $(\lceil 40/32 \rceil * 32) - 40 = 24$. This results in a serialized representation with the data type

'Vec 2 (Signed 32)'".

These *serialization naturals* are either manually inserted as function arguments or have to be computed by means of compile-time code execution with Template Haskell (TH), which will be explained later on.

A graphical representation of a data-flow wrapped function in-between the serialization blocks, which we will call the *function wrapper*, is shown in Figure 4.2.

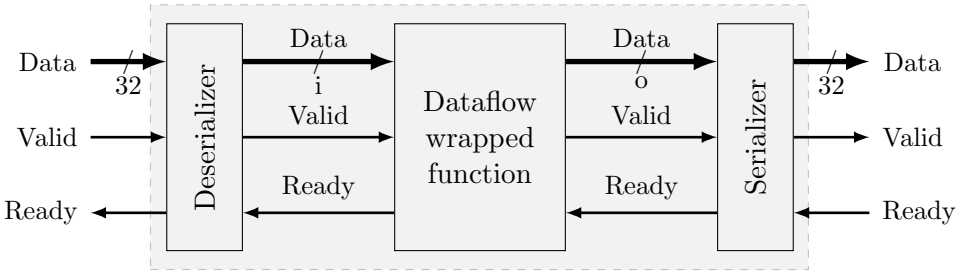


Figure 4.2: An overview of the *function wrapper* containing a single data-flow wrapped offloadable function in-between the serialization blocks.

This *function wrapper* has been implemented as a higher-order function with the type from Listing 4.2. The function type reveals that it is parametrizable with the data-flow composition of the offloaded function and the *serialization naturals* as initial input arguments.

```
1 functionWrapper :: (...)
2                 => (DataFlow Bool Bool i o)
3                 -> (serialization naturals)
4                 -> Signal (XillybusType, Bool, Bool)
5                 -> Signal (XillybusType, Bool, Bool, Bool, Bool)
```

Listing 4.2: Type of the serialisation wrapper for offloadable functions

In order to allow the automatic offloading of more than one Haskell function simultaneously, we need to represent all the offloadable functions in a vectorized form. However, as vectors in Haskell can only have elements with the same type and therefore it is paramount that we can wrap the offloadable functions into the monomorphic *functionWrapper* type from Listing 4.3.

```
1 functionWrapper :: Signal (XillybusType, Bool, Bool)
2                 -> Signal (XillybusType, Bool, Bool, Bool, Bool)
```

Listing 4.3: Function wrapper type after parametrizing

Now that we have a way to create a vectorized representation of all the offloadable functions, we can proceed to describe the implementation of how it is connected to the FIFO IP Cores in the FPGA fabric. To allow FIFO messages to reach the corresponding offloaded function in the vector, we have to demultiplex the received serialized data by means of a demultiplexer logic block. The protocol implementation as described in subsection 4.1.1, allows us to indicate for how long incoming data should be multiplexed to which offloaded function in the list. Subsequently, the multiplexer has to do the reverse of the demultiplexer, in order to send the function result data on the interconnect FIFO. Again, both the multiplexer and demultiplexer blocks are written as Mealy machines. This implementation, which we will call the *top wrapper*,

can be seen in Figure 4.3. In the middle of this figure we see the multi-layered grey blocks, which represents a list of the *function wrappers* functions from Figure 4.2.

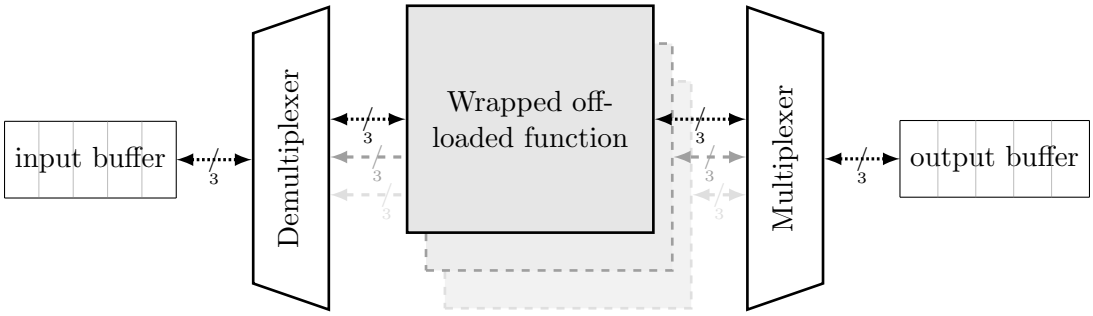


Figure 4.3: An overview of the top level wrapper in the FPGA architecture.

The type of this *top wrapper* function is shown in Listing 4.4. The first input represents the vector of wrapped functions as previously mentioned. After applying the function vector argument, the top-level block diagram of the FPGA architecture will look like Figure 4.4. This figure shows a bundle of four output error signals, which are connected to the LEDs on the SoCKit development board in the Quartus template project. What these errors represent will be explained in the in-depth user manual on [53].

```
1 topWrapper  :: (...)
2             => (Vec n (Signal(XillybusType, Bool, Bool) -> Signal(XillybusType,
                  Bool, Bool, Bool, Bool)))
3             -> Signal (XillybusType, Bool, Bool)
4             -> Signal (XillybusType, Bool, Bool, Bool, Bool, Bool, Bool)
```

Listing 4.4: Type of the higher-order top wrapper function

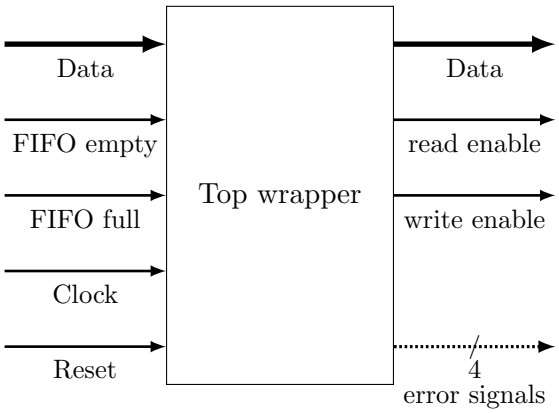


Figure 4.4: The top level diagram of the FPGA architecture.

The complete FPGA architecture is designed with data-flow principles in mind. For instance, if the outgoing FIFO buffer is full we will, by way of back-pressured signals, halt all the previous blocks in the architecture until we can move our data out again. The architecture is designed in such a way that only an offloaded function can be used when the previous function result data has been sent onto the outgoing FIFO buffer. This means that for a pipelined utilization of an

offloaded function (as proposed in Figure 3.4) the minimum latency for calling, and receiving the results of the called offloaded function will be dependent on previous calls to offloaded functions.

Below, the two ways of resetting the complete FPGA architecture (and also the FIFO buffers) are described:

- A manual FPGA architecture specific reset (push button) or a complete FPGA hard reset.
- A reset state by means of the Xillybus IP core when both the FIFO buffer device files are not being accessed by the Haskell program.

The first method will always be available, but the second method may not always be desired in HW/SW co-designs where the device files may be closed during runtime. Therefore, including this reset method is left up to the user, which will be described in the use guide.

This architecture meets all the design requirements set in List 3.3. The FPGA architecture can be compiled in CλaSH with a top-entity function that merely applies the vector of offloadable functions in their respective data-flow compositions and the associated *serialization naturals* arguments. Due to this relatively simple compilation process, it becomes possible to achieve automatic function offloading, which is something that will be discussed in a later section.

This implementation does, however, introduce the additional offloadable function requirement in List 4.1.

> A composition of the offloadable function and a higher-order data-flow function of CλaSH [48] has to be exist, i.e. the function can be passed as an argument to either the *pureDF*, *mealyDF*, *mooreDF* function, or *liftDF*. In the manual use scenario these dataflow compositions–when possible– may be combined into a larger offloadable function by using the *seqDF* combinator function.

List 4.1: Additional requirement for an offloadable function.

At the end of this chapter it will be explained how we can use this FPGA architecture in practice, but first we will look at the rest of the design implementation.

4.3 HPS implementation

Now that the FPGA architecture, SoC interconnect, and the protocol have been implemented, the Hard Processing System (HPS) implementation of the co-design will be described. This section starts with the implementation for the manual use scenario and it ends with the extension that allows for the semi-automated use scenario.

The HPS implementation in its simplest form, i.e. not a pipelined implementation as shown in Figure 3.4, can be described as a process, either manually or automated, in which the annotated offloadable function is replaced with a parametrizable polymorphic Haskell function. In Figure 3.3, we have already seen a proposed behavioural representation of this polymorphic function and in Figure 4.5 the implementation of it is pictured.

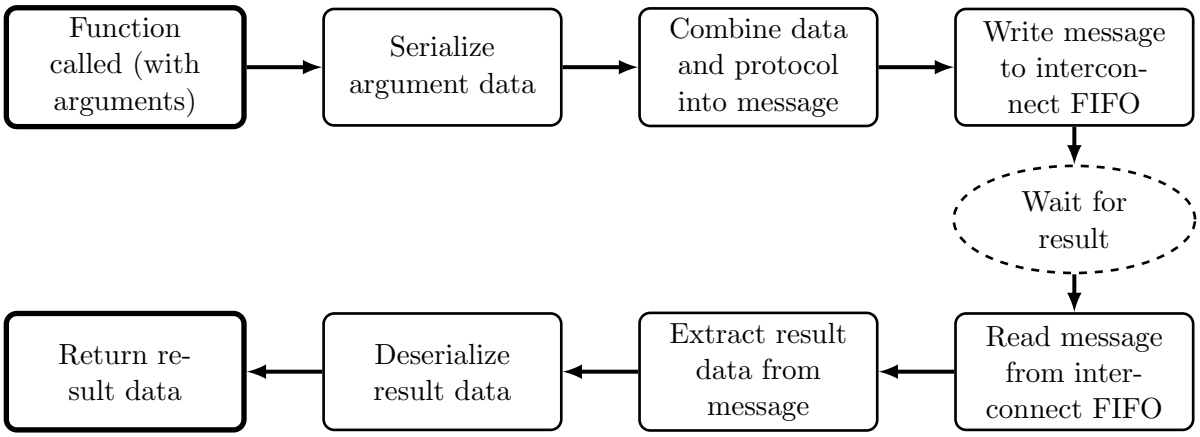


Figure 4.5: Implementation of the replacement function *offloadTemplate*.

This function implementation, which we shall refer to as the *offloadTemplate* function, has the type shown in Listing 4.5. Similar to the FPGA architecture implementation, the *offloadTemplate* function has two arguments that are used to parametrize the function, namely the *function identifier integer* and *serialization naturals*. The integer argument is used to associate messages to the correct function and the *serialization naturals* are identical to the ones used in the FPGA architecture, as explained in the previous section. In this manual use scenario it is the task of the user to apply the protocol correctly, i.e. the function identifier has to be equal to the index of the offloadable function in the listed representation on the FPGA architecture. Once parametrized, the replacement *offloadTemplate* function will have the same function type as the original offloadable Haskell function. This allows the Haskell program to call upon the replaced function as if it was never modified.

```
1 offloadTemplate :: (...)
2                 => Int                -- The function identifier
3                 -> (serialization naturals)
4                 -> ArgumentType
5                 -> ResultType
```

Listing 4.5: Type of the polymorphic offload template function

It is not possible to simply replace an offloadable function in a Haskell program with the *offloadTemplate* if it is a sequential *Mealy* or *Moore* function (in which the function states are to be managed on the FPGA architecture itself). For such a sequential function it is expected that the main Haskell program passes the previous function state as an argument in the next function call. In this case we can either simply loop back the function state when the replaced offloaded function is called or the user is required to manually modify the main Haskell program to handle the function as a pure function. The first solution is the easiest approach, but the second method is more preferable for an optimized design as it completely removes the overhead in the main program of having to pass the state data for each function call.

As we have seen in Figure 4.5, the *offloadTemplate* function consists of several consecutive steps. When the *offloadTemplate* function is called from the main Haskell program it starts by serializing the argument data with the *BitPack* CλaSH module. The protocol is then applied to this serialized list of data to form a message that contains the offloaded function identifier, the data length, and the serialized data itself. This message is subsequently written to the FIFO

buffer device file or in case the FIFO is full it will halt the program until the writing task can be completed.

In subsection 3.2.3 it was also noted that the *IO monad* is introduced when we use the impure functions for accessing the device files of the interconnect. This was solved by using the *unsafePerformIO* function [42] to allow executing I/O operations in a pure function (i.e. `IO a → a`). An IO function that has been made pure with the *UnsafePerformIO* function can have the undesired side effect that Haskell will only evaluate the result once due to its lazy evaluation and return this initial value again in subsequent function calls. This has been solved by enforcing strict evaluation with the *$!* Haskell operator, but it might be more appropriate as future work to approach this *IO monad* problem differently by, for instance, creating and forcing a designer to use an *Offload monad* for offloaded function calls.

Once the message has been successfully written, the function will start reading from the returning FIFO buffer. It will keep doing so until the exact amount of bytes have been read from the buffer to reconstruct the offloaded function result data message. The resulting message is then quickly verified for protocol errors and subsequently the serialized result data is extracted. This data is deserialized into the data with the function result type and is then returned to the main Haskell program.

In chapter 2 we have set the requirement that an offloadable function should be written in CλaSH compilable functions. This means that a traditional Haskell program with variable sized lists and a superset of data types, has to interface with CλaSH code. Most of the numerical data types, including fixed- and floating-point data types, can be coerced to and from CλaSH types by using the *fromIntegral* function. The coercion requires that the data types are explicitly defined beforehand. Coercing between a list and a fixed sized CλaSH vector can be achieved with the two following functions:

- The *toVector* function may be used to convert a list to a vector.
- The standard CλaSH *toList* function can be used for vector to list conversions.

4.3.1 Manual pipelined implementation

In chapter 3 it was concluded that in the manual use scenario, it was desired to allow users access to a split version of the *offloadTemplate* function. These separate *write* and *read* functions will allow for users to increase the overall throughput of their HW/SW co-design by pipelining their application, as shown in Figure 3.4.

The *write* function behaves almost exactly like the first part of the *offloadTemplate* function as shown by the steps on the top row of Figure 4.5 and the *read* function is mostly identical to the bottom row. Due to limited implementation time, the *write* function simply returns a write success boolean to the function that called it. In a future version of this proof of concept, this may be replaced with the IO monad or a dedicated *Offload* monad to indicate failures in a more Haskell programming-friendly manner. The *write* function type is listed above in Listing 4.6 and the lower type is that of the *read* function.

For pipelined applications, the FIFO buffer device files must remain open during its complete lifespan, or else the data in the DMA buffers will be flushed, which, besides significantly increasing the latency, may also cause data loss in the read FIFO buffer. In our previous non-pipelined co-designs this will not be a potential problem, because all data is guaranteed to have been written and read from the FIFO buffers before the device files are closed. This means that the file descriptors in the pipelined applications should only be once opened at the start

of the main function. Subsequently, these file descriptors should then be passed as the *Handle* argument of the *read* and *write* functions as given in Listing 4.6.

```
1  offloadwriteTemplate :: (...)
2      => Int       -- The function identifier
3      -> Handle    -- Opened file descriptor of the write FIFO device file
4      -> (argument serialization naturals)
5      -> ArgumentType
6      -> Bool
7
8  offloadreadTemplate :: (...)
9      => Int  -- The function identifier
10     -> Handle   -- Opened file descriptor of the read FIFO device file
11     -> (result serialization naturals)
12     -> ResultType
```

Listing 4.6: Type of the polymorphic offload write template function

A problem now rises for the pipelined *read* function when we want to read with more than one thread concurrently. The issue in this case is that the *read* function will possibly receive messages on the shared FIFO that are meant for other offloaded function calls. And once read from the FIFO buffer, we are not able to put them back. The most prominent solution to this problem, which falls outside the scope of this thesis, is to configure the Xillybus IP core such that each offloaded function has a separate FIFO buffer pair and their associated device files. Each FIFO buffer pair is then able to use a unique version of our designed FPGA architecture. This solution allows a pipelined HW/SW co-design to efficiently use each separate offloaded function due to that the Xillybus IP core manages FIFO buffer data in the DMA buffers of the CPU. A second, but more computational intensive method to solve the concurrent reading problem, is to create a main thread that acts as a master reader, which will divert the messages to the correct slave reading threads.

4.3.2 Automatic offloading core plugin

The HPS implementation, up to this point, only allows for the manual use scenario as described in section 3.1. This section describes the extension that was implemented to achieve the semi-automated offloading use scenario. The FPGA implementation already has the potential for a (semi-)automated generation, which allows us to mainly focus on automating the transformation of an offloadable function into the polymorphic function *offloadTemplate*. In chapter 3 we already decided to use the Core plugin in order to modify the Haskell program at compile-time and we also set the preliminary requirements for it in List 3.2. For this implementation, it was chosen to generate the FPGA architecture top-entity module [1] during the Core plugin execution.

In order to automate the process of changing one or more offloadable functions, we required the user to annotate them. This is achieved by adding the following annotation [12] in the module for each targeted offloadable function:

```
{-# ANN functionName pureOffload #-}
```

In this annotation the *functionName* should be replaced with the actual targeted function identifier, such as shown in the example in Listing 1.2. This allows the Core plugin to search through all the function bindings in the current Core module for the annotated function(s). The

[1] Top-entity module is used to automatically compile the FPGA architecture with the annotated offloadable functions

pureOffload keyword indicates to the Core plugin that a function should be offloaded as a pure function. In the current implementation the *mealyOffload* and *mooreOffload* variants have not been fully implemented for reasons that are discussed further in this section, but instead they will notify the user to use manual offloading instead.

A Core plugin can only be active in the scope of the module it has been assigned to. In this automated function offloading implementation, this means that dividing the offloadable functions in more than one Haskell module will only be possible if data is shared in between Core plugin passes by using intermediate files. This has not been implemented for this proof of concept, but is expected to be possible in future updates. To ensure the plugin will be applied to our module of choice, and the annotations are in-scope, the two lines from Listing 4.7 have to be added at the top of the module.

```
1 {-# OPTIONS_GHC -fplugin=Offload.Plugin.OffloadPlugin #-}
2 import Offload.Plugin.OffloadAnnotations
```

Listing 4.7: Required lines of Haskell code to allow for assigning the Core plugin to a module in the semi-automatic function offloading implementation

In Figure 4.6 we can see a flowchart that describes the behaviour of the Core plugin. The dashed blocks in the chart represent the generation process of the top-entity module file during the plugin. This file is made available in the current compiler directory. Once the initial part of the top-entity file has been generated, the plugin will start looping over the available function core bindings. When an annotated binding is found, it will extract the argument and result types. The grey block in Figure 4.6 represents the process of generating the previously described *offloadTemplate* function with the two preciously extracted types.

Once a Haskell module has been parsed into a Core module, it can no longer be modified with standard user-space Haskell functions. Modifying this Core module by exclusively using Core type functions is, aside from being a complex task, also rather impossible in our case. For instance, if we want to automatically calculate the *serialization naturals* for the *offloadTemplate* function, then we require compile-time execution of code. In the manual use scenario Template Haskell was used to achieve this, but that can not be applied to a Core module in a Core plugin without some form of interactive Haskell compiler.

For this implementation it was chosen to generate temporary Haskell modules on the fly, that will contain the generated and parametrized *offloadTemplate* function. Such a temporary module is subsequently compiled to an intermediate Core module and the relevant function binder is then brought into scope of the initial Core module. This can be considered an undesirable method, because dynamic linking errors can occur when we copy a function binding from one Core module to another Core module. In our case a linker error will occur because the *BitPack* and *KnownNat* classes and the entire module containing the polymorphic offloading template function were not in scope of the initial Core module. To solve our dynamic linking errors, we will have to require users to add these classes and module to the original module containing the offloadable functions. This is achieved by inserting the three lines of code from Listing 4.8 in the Haskell module containing the offloadable functions.

```
1 import Offload.Haskell.OffloadFunctions
2
3 makeClassesInScope :: (BitPack Bit, KnownNat 0) => () -- required for Core plugin
4 makeClassesInScope = ()
```

Listing 4.8: Required lines of Haskell code to allow for semi-automatic function offloading

Once these classes are in-scope of the module that is being modified by the Core plugin, then linking errors are resolved by constructing the new function type with the in-scope linked classes. For this exact reason an additional offloadable function requirement is necessary, as described in List 4.2 to get the *BitPack* class in the scope of the first Core module. An example of this will be given later on.

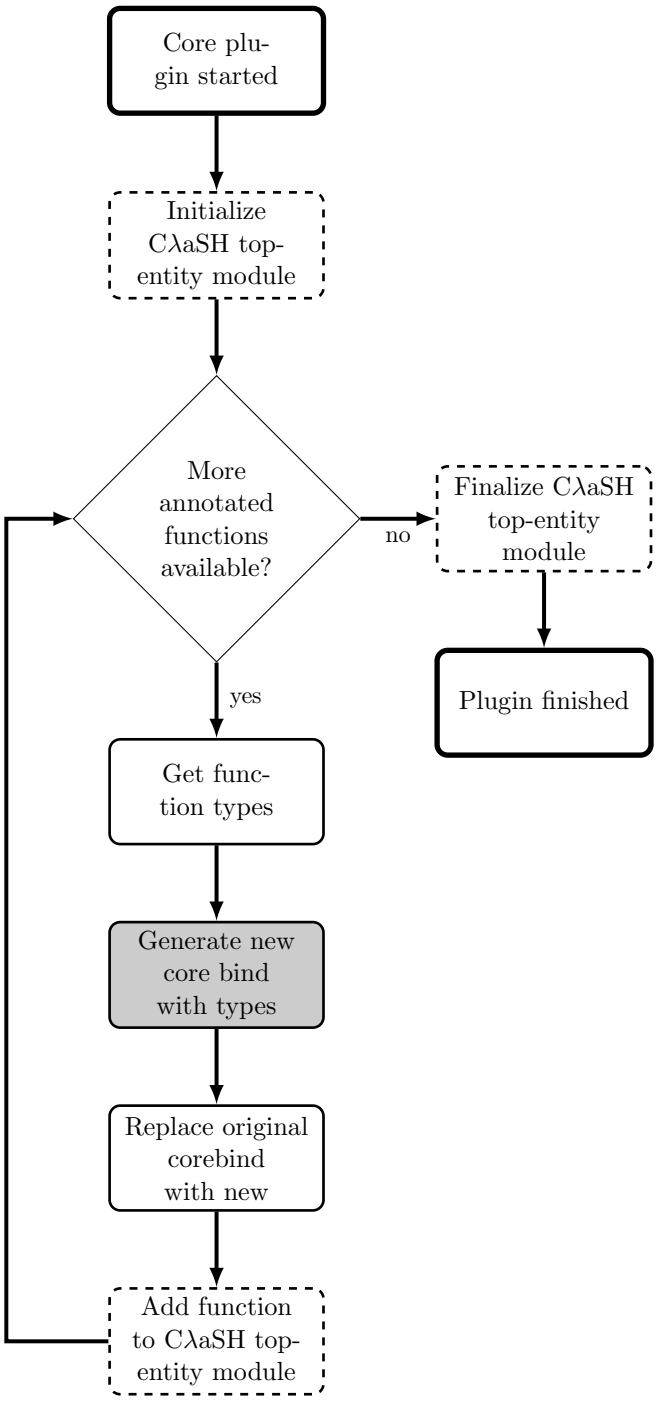


Figure 4.6: Flowchart of the Core plugin implementation.

A module containing the offloadable functions and which is used for semi-automated offloading, must contain the code from Listing 4.8.

List 4.2: Additional requirement for an offloadable function in the semi-automated use scenario.

We can now proceed with replacing the original function binder with the newly generated polymorphic function *offloadTemplate* that has been parametrized with the function types. In the last step of the loop in the flowchart in Figure 4.6 the function binding name is appended to the offloaded functions list in the top-entity file for the FPGA architecture. By doing this for each loop iteration, we can assure that the function identifier on the Haskell partition is exactly identical to the index function list on the FPGA part.

This loop will be active for as long as there are new annotated functions available and when it finally ends, the CλaSH top-entity module is finalized and the modified Core module will be used in the final Haskell compiler phase to generate the executable.

In the manual use scenario implementation, we have already stated that the offloadable sequential *Mealy* and *Moore* functions require additional modification to the main Haskell program to remove the managing of state data. In addition, the initial states of these sequential functions have are not known at compile time for the FPGA architecture, which prevents us from automating the offloading process. For these two reasons, it was chosen that we will have to restrict the Core plugin to allow only *pure* offloadable functions for the semi-automated use scenario.

4.4 Overview

All the resulting code of this implementation of the design from section 3.3 has been combined in a folder that represents the (unofficial) Haskell offload package. This package can be downloaded from the GitHub repository [53] into the development directory of choice. In chapter 6 the verification process of the implementation is described. In the next chapter is a brief user manual given that describes the intended workflow for both the manual and semi-automated use scenario. The next chapter also formally lists the Haskell HW/SW co-design requirements that were encountered during the design and implementation phases of this thesis.

5 User manual

Up to this point of the thesis we have been targeting this proof of concept for two use scenarios: semi-automated and manual Haskell function offloading. The final implementation imposes several requirements on a Haskell HW/SW co-design, which were already described during the design and implementation phases in previous chapters. These requirements have been formally listed in the following first part of this chapter. The manual and semi-automated use scenario have a different, but partly overlapping, set of requirements. For both the use scenarios, the following requirements are applicable:

- All Haskell functions that are to be offloaded should adhere to the following requirements:
  * The offloadable functions are completely compilable to a hardware description language[1] by CλaSH.
  * By default an offloadable function should be in the form of either one of the following abstract types [24]:
    - Pure function: `i -> o`
    - Mealy function: `s -> i -> (s, o)`
    - Moore function: `(s -> i -> s) -> (s -> o)`
  * A composition of the previously mentioned offloadable functions and a higher-order data-flow function of CλaSH [48] has to be exist, i.e. the function can be passed as an argument to either the *pureDF*, *mealyDF*, or *mooreDF* function.
  * The argument and result types should both be able to have an instance of the class *BitPack*[46], such that the functions *pack* and *unpack* can be applied to them.
- In order to compile the FPGA architecture with the listed representation of offloadable functions, all the targeted functions have to be within one or more Haskell modules that exclusively consists of CλaSH compilable functions.
- It is required that the user only designs a set of offloaded functions that, in combination with the rest of the FPGA architecture and Xillybus IP core, will fit within the resource budget of the FPGA.
- The user is required to fulfil the task of preventing non-deterministic behaviour by assigning the correct clock frequency to the FPGA architecture such that, in all possible system states, the propagation delays of the logic will not exceed the clock period.
- The software partition of a Haskell co-design should be thread-safe when calling upon offloaded functions, such that no race conditions can occur.

[1] In this implementation only a VHDL Quartus project was prepared, but Verilog should be possible too with some additional work.

In addition to the shared requirements, for each use scenario there are also separate requirements. For the semi-automated use scenario the following implementation requirements apply:

- Offloadable functions in the Core plugin can only be offloaded as pure functions, i.e. with the type: `i -> o`. The said type should not contain any separately defined types, because the Core plugin does not have access to them in the intermediate files. Each offloadable function should be annotated as described in subsection 4.3.2.
- All the offloadable functions should be in a single Haskell module that exclusively consists of CλaSH compilable functions[2]
- The GHC compiler should be started with the `-package ghc` command line argument. This is necessary as the Core plugin requires access to the GHC API modules[3].
- The module containing all the offloadable functions should contain the additional code from Listing 4.7 and Listing 4.8[5] in order to facilitate automated function offloading with the Core plugin.

For the manual use scenario the following implementation requirements apply:

- In addition to the three allowed basic offloadable function types (i.e. pure, mealy, and moore), it is also allowed to manually create sequential combinations of them, which are to be connected using the dataflow composition combinator function *seqDF* from CλaSH [48].
- It is also allowed to manually offload a function that is *Signal* type based [24]. It must be liftable to the dataflow type using the *listDF* function from CλaSH [48].
- It is the user's task to apply the protocol parameters correctly to the polymorphic functions and FPGA architecture. This implies that the function identifiers should be unique and that they are matching between the hardware and software partitions. Additionally, the *serialization naturals* required by the functions should be correctly given (either calculation using Template Haskell or manually).
- It is the task of the user to modify the main program to allow for the sequential *Mealy* or *Moore* functions. This means removing the process of folding the states
- The user should use the pipelined optimization approach with only a single reading thread, unless the in subsection 4.3.1 mentioned solutions are used.

Now that all the important requirements and restrictions for both use scenario have been listed, we can continue with describing the actual intended usage of the implementation. This user manual in this chapter will only briefly describe the steps involved in using the proof of concept, whereas the GitHub repository [53] contains a more in-depth user manual with practical examples. It is presumed that an identical development set-up of this thesis is available as described on the GitHub repository [53] and that the required files (i.e. our unofficial Haskell Offloading package) from this proof of concept have been downloaded and placed into the current working directory. If we recall the proposed workflow in subsection 2.2.1, then we already have a structural overview on how the implementation should be used for both scenarios.

[2]This occurs due to that this proof of concept does not take into account that multiple instances of the Core plugin can be active. In that case it would then result in duplicates of function identifiers and an incorrectly generated top-entity module for the hardware partition.

[3]Can be circumvented by compiling and installing the offloading Core plugin as *GHC.Plugin*.

Semi-automatic offloading guide The semi-automated function offloading implementation should be used as described in the following steps:

1. Start with a Haskell software-only design in *module A*.
2. Re-write the functions that will be offloaded according to the previously listed requirements for automated offloading and transfer them from *module A* into *module B*, which now will become an imported module of *module A*. Subsequently, simulation should verify the desired functional behaviour before proceeding.
3. The desired offloadable functions in *module B* should now be annotated and additional code is to be added to *module B* such that the Core plugin can be correctly invoked as stated in the automated offloading requirements above.
4. Compile *module A* (with *module B* as an import) to an executable object file with GHC on the ARM, which results in the software partition of the co-design. The Core plugin will also automatically generate the top-entity module file that is used by CλaSH to generate the FPGA architecture that contains the hardware partition of the co-design.
5. The resulting top-entity module file has to be compiled with CλaSH[4], which will parametrize and generate the desired FPGA architecture in VHDL. The annotations and additional Core plugin code in module B should be removed again or commented out[5] in order to allow compilation with CλaSH.
6. The resulting VHDL files are to be added to the Quartus template project and synthesized to a programmable FPGA bitstream[4].
7. The FPGA bitstream should now be uploaded to the ARM in order to program the FPGA at boot-time. Once the FPGA has been configured, then the in step 4 generated executable object file may be used to start the HW/SW co-designed Haskell program.

Manual offloading guide The manual offloading process allows for either a normal or a pipelined design by using the standard offload function or the separate write and read functions respectively. The process is somewhat different to the automated use scenario as shown in the following enumerated steps:

1. Start with a Haskell software-only design in *module C*.
2. Re-write the functions that will be offloaded according to the previously listed requirements for manual offloading and move them into one or more separate modules, which we will call the *module set D*. Subsequently, simulation should verify the desired functional behaviour before proceeding.
3. Create the top-entity module for the CλaSH compilation process of the hardware partition. A template file is available on [53], which includes practical examples. Import the *module set D* and correctly fill in the argument vector of offloadable functions of the *topWrapper* function in the template file.
4. The resulting top-entity module file has to be compiled with CλaSH, which will parametrize and generate desired FPGA architecture in VHDL.

[4] On the faster x86-based host PC

[5] Actually, creating a copy of module B in step 2 and manually importing it instead is also possible.

5. The resulting VHDL files are to be added to the Quartus template project and synthesized to a programmable FPGA bitstream.

6. The desired offloadable functions in the *module set D* should now be manually replaced[6] with the *offloadTemplate* function or in the case of a pipelined design with the *offloadWriteTemplate* and *offloadReadTemplate* functions. This includes the correct assignment of *function identifiers* and *serialisation naturals*. The *module C*, which calls upon the replaced offloadable functions, should now be modified accordingly. In the case of a pipelined utilization of the offloadable function, the *module C* should be converted into a design with separate write and read threads.

7. Compile *module C* (with the *module set D* as an import) to an executable object file with GHC on the ARM, which results in the software partition of the co-design.

8. The FPGA bitstream is uploaded to the ARM in order to program the FPGA at boot-time. Once the FPGA has been configured, then the in step 7 generated executable object file may be used to start the HW/SW co-designed Haskell program.

The actual in-depth user manual for both use scenarios is found on the GitHub repository [53]. Additionally, it includes all the information to re-produce the exact same set-up and subsequently the same results of this thesis and it includes several practical examples such as the benchmarked functions in the next chapter.

[6] It is recommended to make a copy of *module set D* here to retain the old functions

6 Results

With the implementation as given in chapter 4, combined hardware and software designs in Haskell can be developed and implemented relatively quickly. It is self-evident that these type of system designs are desired to be implemented efficiently. It is therefore necessary to analyse the performance of this proof of concept in several scenarios, but first we will need to know that the implementation is working correctly according to our design. This verification has been performed by initially testing each of the main design components separately as follows:

- The Xillybus IP core configuration has been extensively tested with C-code demonstration applications as provided by Xillybus.
- The FPGA architecture implementation has been tested in the CλaSH interactive compiler using a custom test-bench, which functions similarly to the in- and output FIFO IP cores to which it will be connected in the template HDL project. This test-bench allows for verification of the implementation with all possible average and boundary test scenarios, i.e. full/empty FIFO buffers, different data sets, and all potential offloadable functions. Subsequently, the FPGA architecture has also been tested on the SoC itself with an adapted demonstration application of the Xillybus IP core.
- Subcomponents of the HPS implementation, including the Core plugin, have been verified in the interactive compiler of Haskell. In order to verify the complete HPS implementation, a single FIFO buffer was used on the FPGA side of the interconnect, which directly loops back the serialized data to the HPS in order to simulate the FPGA architecture. This allowed for verifying many of the major test scenarios with the restriction that only offloadable functions were allowed that had the same argument and result type.

Subsequently, all three combined components have been verified with test-benches that are similar to the performance benchmarks in the next section.

6.1 Performance benchmarks

In this section we will see the results from benchmarking our proof of concept with several different offloadable functions on the following three platforms:

- A laptop with an Intel core i7 processor [49].
- Only the ARM of the SoCKit.
- The ARM and FPGA of the SoCKit.

The first two platforms will feature software-only designs and the last option is intended for the HW/SW co-design counterparts. The *Criterion* benchmarking library [50] is an often used tool to test the performance of Haskell functions, however, it is not thread-safe and therefore we will use our own test-bench instead. This test-bench calculates the execution time of a given Haskell function by means of the following steps:

1. Store initial time t_0.
2. Execute the Haskell function N times[1].
3. Store final time t_n.
4. Calculates the individual execution time with the formula: $(t_n - t_o)/N$.

The resolution of the timing function in Haskell is not sufficient to time an individual offloadable function execution correctly, therefore the test-bench has to execute the offloadable function multiple times and the resulting time is then divided by the amount of executions. This amount of offloadable function executions N has been set to 10^4 by default. It is important to realize that the test-bench itself introduces additional overhead due to its implementation in step 2, which means that the resulting execution times are not exclusively of the offloaded functions. However, as the same test-bench is used on all of the three platforms, it is assumed that these benchmarks will still indicate the effectiveness per platform and to a certain extent the performance per function. Additionally, all test-benches have been compiled to an executable object file as opposed to just executing them in an interactive GHC environment, which results in a better performance.

In the next part of this section an analysis is performed on the benchmark results of the following three offloadable functions: a trivial multiplication, a matrix multiplication, and a FIR-filter.

Multiplication benchmark This first benchmarked function is the 32-bits signed integer multiplication as pictured in Figure 6.1. The corresponding Haskell description is given in Listing 6.1. This simple function can be used to estimate the minimal latency that each offloadable function will have when calling upon them in a non-pipelined fashion.

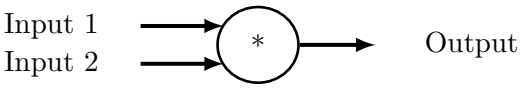


Figure 6.1: Multiplier structural block

```
1 mult     :: (Signed 32, Signed 32)
2          -> Signed 32
3 mult (ia, ib) =  (ia * ib)
```

Listing 6.1: A 32-bits signed integer multiplier Haskell function

The results of benchmarking the multiplication function on the different platforms is shown in Table 6.1. As expected, the software-only benchmark on the Intel processor is considerably faster than the ARM processor. The HW/SW co-designed benchmark shows an execution time of more than a millisecond, which is a factor 70 higher than the software-only design on the ARM. Therefore, it may be concluded that offloading simple functions onto an FPGA in a non-pipelined fashion will result in a significantly lower system throughput.

[1]The lazy evaluation in Haskell will not fully evaluate the benchmarked function results as they are not used. This is solved by forcing a strict evaluation by applying the *show* function to the complete result list.

Platform	Execution time* (μsec)
Intel processor	1.2
ARM processor	17
ARM + FPGA SoC	1186

Table 6.1: Benchmark results of a 32-bits signed integer multiplier function. *The execution time includes overhead of the test-bench.

Matrix multiplication The second benchmarked function is a matrix multiplication. An important aspect of this function is that it requires more data to be transferred between the FPGA and CPU for each function call. For instance, matrix multiplying the matrix A with matrix B will result in matrix AB, as given below, and thus all three matrices will have to be transferred over the interconnect to or from the offloaded function.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix}, B = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,p} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,p} \end{pmatrix}$$

$$AB = \begin{pmatrix} (AB)_{1,1} & (AB)_{1,2} & \cdots & (AB)_{1,p} \\ (AB)_{2,1} & (AB)_{2,2} & \cdots & (AB)_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ (AB)_{n,1} & (AB)_{n,2} & \cdots & (AB)_{n,p} \end{pmatrix}$$

Equation 6.1 shows that each entry in the matrix AB is essentially a dot product of a row of matrix A and a column of matrix B.

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj} \tag{6.1}$$

The Haskell description of the benchmarked matrix multiplication function is given in Listing 6.2. It is designed to multiply two matrices with both a length and width of 3 and 16-bits signed integer data types in each entry. The output of the function is also a matrix of the same specification. This means that the serialized argument data has a length of nine 32-bits words and subsequently the result data has a length of five. The matrix multiplication function is considerably more complex in comparison to the previously benchmarked multiplication function as it contains 27 multiplications and 18 additions.

```
1 matrixMult  :: ((Vec 3 (Vec 3 (Signed 16))), (Vec 3 (Vec 3 (Signed 16))))
2             -> (Vec 3 (Vec 3 (Signed 16)))
3 matrixMult (matrixA,matrixB)    = map (multVectors (transpose matrixB)) matrixA
4     where
5         multVectors matrixBT row= map (dotProduct row) matrixBT
6         dotProduct rowA rowB    = fold (+) (zipWith (*) rowA rowB)
```

Listing 6.2: A matrix mutliplication Haskell function for 3 by 3 matrices

The benchmark results of the matrix multiplication function are given in Table 6.2. It shows that the HW/SW co-designed benchmark has an execution time that is merely a factor 5 larger than the software-only benchmark on the ARM. The main reason for this improvement with respect to the benchmarked multiplication function, is that the function can be completely

implemented on an FPGA in parallel, which means that it only takes a single clock cycle on the FPGA to calculate the result instead of multiple clock cycles on the CPU. This benchmark also shows that the offloaded function execution time, which is 1.8 milliseconds instead of the previous 1.1 ms, is considerably influenced by the size of the argument and result data types. The main reason for this observation is that the data has to be serialized four times, which –especially in software– can be a somewhat computational intensive task for large and complex data types.

Platform	Execution time* (μsec)
Intel processor	24
ARM processor	365
ARM + FPGA SoC	1818

Table 6.2: Benchmark results of a matrix multiplication for 3-by-3 matrices with 16-bits signed integer types. *The execution time includes overhead of the test-bench.

FIR-filter The last benchmarked function is the FIR-filter, which was already extensively described in section 1.1. The benchmarked FIR-filter has been parametrized to a 24th-order filter with 32-bits signed integer data types, which means that it contains 24 multiplications and 23 additions. In contrast to the previous two pure type functions, this FIR-filter function has been offloaded as a Mealy type function, which means that the state vector (i.e. registers) is managed on the FPGA instead of transferring it between the CPU and FPGA for each function execution. The resulting function that is called upon by the main test-bench program to execute the offloaded function has therefore the type: `Signed 32 -> Signed 32`. The coefficient vector of the FIR-filter is deemed trivial for this benchmark as it does not feature any of the potential optimizations that were proposed in section 1.1.

Platform	Execution time* (μsec)
Intel processor	28
ARM processor	425
ARM + FPGA SoC	1128

Table 6.3: Benchmark results of a 24th-order FIR-filter function with 32-bits signed data types. *The execution time includes overhead of the test-bench.

The results of the benchmark are given in Table 6.3. The difference between the execution times of the software-only benchmark on the ARM and the HW/SW co-designed benchmark has been further reduced to 2.7. Therefore, it is estimated that when a function is sufficiently computational intensive, then it may become advantageous to offload it in a non-pipelined fashion. However, as will be shown in the next section, a pipelined utilization of this specific offloaded FIR-filter function will actually increase the performance significantly.

6.1.1 Pipelined benchmark

All three previously benchmarked functions have also been benchmarked in a pipelined fashion as proposed in Figure 3.4. This means that a *write* and a *read* thread run separately on the CPU processor, which essentially allows for omitting the minimal latency that is introduced by the data transmissions between the FPGA and CPU. In this pipelined test-bench the latency between the read function results is benchmarked.

Table 6.4 includes the results of the pipelined benchmarking of the three functions. It show that the pipelined multiplication function is still about five times slower than the software-only multiplication on the ARM. This was to be expected as the serialisation and execution on the FPGA will always be slower than a single local multiplication on the ARM itself. Secondly, the pipelined matrix multiplication shows that it also has no actual advantage over a software-only variant on the ARM, which mainly can be explained by the serialization process of the larger matrix data types. Lastly, the pipelined FIR-filter does show a significant improvement factor of about 5.2 in comparison to the software-only variant.

Function	Read latency* (μsec)
Multiplication	82
Matrix mult.	370
FIR-filter	82

Table 6.4: The pipelined benchmark results of the three previously benchmarked functions. *The read latency includes overhead of the test-bench

The implementation of function offloading in this thesis allows for even more complex offloadable functions in comparison to these three functions, which will most likely have a much greater performance gain in a HW/SW co-design. For instance, with the manual offloading use scenario, it is possible to combine multiple offloadable functions in a pipelined fashion, or even to create offloadable functions that take multiple clock cycles to produce their results.

7 Discussion and conclusion

Our primary problem for this thesis was formulated in the introduction as follows: "How can we automatically offload selected functions of a Haskell program onto an FPGA and subsequently call upon them remotely from the remainder of the Haskell program?". This problem has been primarily solved through means of successfully implementing a proof of concept that allows for an either manual or semi-automated approach of function offloading within Haskell. The final implementation is not fully automated, as was demanded by the problem formulation, however this thesis does indicate that it is theoretically possible by means of a scripted co-compilation process.

Up to now, the topic of hardware/software co-design in the functional language Haskell has not yet been extensively researched. In the reviewed literature there were some interesting ideas that could prove useful for future work. For instance, introducing our implementation of function offloading to the distributed programming environment of Cloud Haskell can be potentially very interesting to achieve more efficient distributed systems.

In the paper of Teich [1] on the review HW/SW co-design, he mentioned that additional methodologies are required for these types of co-designs. Haskell, in combination with CλaSH, allows for many of these co-design methodologies, such as:

- Design space explorations, because reasoning about HW/SW co-designs in Haskell is easier due to its close relation to hardware architectures.
- Co-simulation, which is possible through the interactive compiler environment of Haskell combined with HDL simulators (e.g. Modelsim) [4]
- Co-compilation or co-synthesis. In this thesis it is shown that co-compilation can be partly achieved by using compile-time partitioning and modifications with a Core plugin.

Therefore, we can conclude that, with more research and tool development, an extensive methodological approach to HW/SW co-design in Haskell is possible.

During the thesis there were some issues encountered that took a considerable amount of time. One major issue was the extensive amount of time put into the ARM cross-compiler, which later on turned out to be impossible in its current implementation. Ideally, this cross-compiler would still be desired over compiling Haskell on the ARM itself, as was discussed already in Appendix B. In the following sections the design, implementation, and result phases of this thesis are discussed and associated conclusions and recommendations are given.

7.1 Design and Implementation

The design phase introduced several interesting observations and issues. The first topic of the design space exploration proposed an alternative platform that could allow for the full automatic

use scenario. This alternative is a desktop computer with an FPGA board connection through a PCIe bus. It is expected that we can port our proof of concept SoCKit implementation onto this alternative platform without too much effort, because we can also use the Xillybus IP core for the PCIe bus connection in order to configure it similarly to our current FIFO buffer based configuration. Besides that this alternative platform should be capable to fully automate the process of function offloading, it can also be used to achieve a higher performance with the more powerful hardware as was shown by the software-only benchmark results in chapter 6.

The Xillybus IP core was mainly chosen to reduce time spent on the implementation of the desired interconnect configuration between the FPGA and CPU. However, the Xillybus IP core also imposes the following disadvantages:

- Paid license required for commercial use.
- Interconnect data width is by default restricted to 32 bits.
- Xillybus IP core project is restricted to Quartus II version 13.0sp1.
- IP core is only (re)configurable on the Xillybus website, which prevents automated generation of an advanced co-design.
- The IP core has an unnecessarily large footprint on the FPGA due to the additional functionalities it provides but are unused in the proof of concept.

Due to these restrictions it might be interesting as future work, to replace the Xillybus IP core with a self-designed version in the future. Additionally, the combination of the Xillybus IP core and our implementation of the FPGA architecture does not allow for parallel execution of offloaded functions. Therefore, it would be interesting for the newly designed interconnect to integrate a method of generating a FIFO buffer pair for each offloaded function automatically, which was also a proposed solution in subsubsection 3.2.2.2 to allow multiple offloaded functions for pipelined applications.

The implementation of the protocol used to communicate between the CPU and FPGA can be optimized with respect to the data widths. Mainly the header, function identifier and data length words can be combined into a single 32-bits word. Furthermore, error correction of the messages might be desired for future co-design platforms that include a less reliable interconnection between the FPGA and CPU.

The FPGA architecture has been designed to be automatically generated with a vector of offloadable functions as input argument, but if only a single function should be offloaded, then a less complex architecture might be desired (i.e. the (de)multiplexing hardware and possibly even the protocol is unnecessary). This can be used to slightly reduce the control overhead and to achieve a smaller hardware design and a better performance. Subsequently, the software partition can also be slightly simplified for a reduced control overhead.

The Core plugin implementation uses intermediate evaluation of Haskell modules to generate the replacement expression for the original offloadable functions. This is not the proper solution, but creating the replacement function from only the context of the original Core module can be considered as being too complex and in certain ways even impossible.

A restriction introduced by our Core plugin implementation is that all offloadable functions should be in the same Haskell module. If this restriction is not adhered to, and multiple instances are invoked of the Core plugin, then the automatically generated hardware partition top-entity file will be overwritten and the offloadable function identifiers will be duplicated. As stated in subsection 4.3.2, this can be solved in future work by sharing the required data in-between the invoked Core plugin instances.

An additional set restriction to the Core plugin is that merely pure functions can be automatically offloaded. As previously mentioned, the primary reason for this restriction is that it requires the initial state of the Mealy/Moore functions to be manually specified for the hardware partition and preferably the remaining software partition has to be modified. However, if desired, this restriction may be removed in future work by forcing the user to specify the initial state themselves in the hardware partition and using the offloadable function replacement that simply loops back the state data to the main Haskell program.

7.2 Results

The benchmark results of section 6.1 show that a non-pipelined co-design with a simple offloaded function will result in a lower throughput than an identical software-only design. Offloading a computational intensive function, however, can result in a co-design with a higher throughput than its software-only counterpart. However, using the offloaded functions in a pipelined fashion is a considerably more effective co-design approach due to the fact that the data transport latencies may be neglected. The results in subsection 6.1.1 show that a non-optimized 24th-order FIR-filter will perform up to 5.2 times better when offloaded. It may therefore be concluded that pipelined Haskell HW/SW co-designs are more efficient, however, pipelining is difficult to be automatically applied as it requires complex modifications to the software-only partition. Instead it is recommended to focus more on the manual approach of function offloading, such as forcing the user to manually create a multi-threaded co-design.

The benchmark results also show that the offloadable function type influences the co-design performance, which can be largely explained by the fact that calling an offloadable function includes four serialization steps. The current implementation of CλaSH is not optimized for simulation performance, i.e. no in-lining, and therefore it might be recommended as future work, to improve the serialization process in software by updating CλaSH or alternatively using the more restricting implementation of (lazy) byte-wise data packing as mentioned in chapter 3 instead of bit-wise.

7.3 Final recommendations

We have already seen some recommendations for future work in the previous sections of this chapter. In this section some final recommendations are given.

As briefly mentioned in section 2.2, extending the current dataflow implementation in CλaSH to support sequential *Signal*-based functions is desirable for function offloading. These *Signal*-based functions can be more complex than Mealy/Moore typed functions and are also more intuitive to use with normal Haskell programming due to its internal list-based design.

It is also recommended as future work to remove the *unsafePerformIO* method in the manual offloading implementation, such that a co-designed Haskell program explicitly shows that it uses the impure offloading functions with the *IO monad* and that it should use the associated design practices.

Lastly, it might also be interesting as future work to make this thesis' implementation of HW/SW co-design available for other common programming languages such as C/C++. This may be realized by creating a library to interface with offloaded Haskell functions similar to the current implementation.

A Higher-order Haskell functions

The table, in Figure A.1, shows the structural representations of the higher order functions *map*, *zipWith*, *foldl*, *scanl*, and *mapAccumL*. The figure originates from the slides of the course Embedded Computer Architectures 2 at the University of Twente.

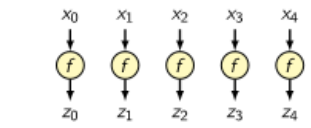
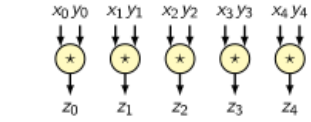
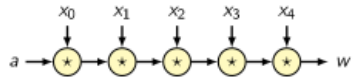
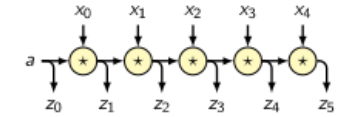
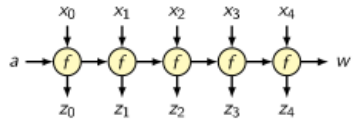
map		$f\ x \Rightarrow z$	$zs = map\ f\ xs$
zipWith		$x\ \star\ y \Rightarrow z$	$zs = zipWith\ (\star)\ xs\ ys$
foldl		$a\ \star\ x \Rightarrow a'$	$w = foldl\ (\star)\ a\ xs$
scanl		$a\ \star\ x \Rightarrow a'$ $z = a$	$zs = scanl\ (\star)\ a\ xs$
mapAccumL		$f\ a\ x \Rightarrow (a', z)$	$(w, zs) = mapAccumL\ f\ a\ xs$

Figure A.1: Higher-order functions in Haskell and their structural representation.

B SoCKit development kit

This appendix features as a technical overview of the SoCKit development kit. It also contains additional background information that is used in the thesis.

There are multiple SoC solutions available on which this thesis project can be realized. For this thesis, a particular SoC platform was already chosen as a potential candidate, namely the SoCKit development board (revision C). It is a board specifically designed for System-on-Chip (SoC) application and is created by Arrow Electronics[30] and produced by TerasIC Inc [31]. It contains a Cyclone V SoC, as seen in Figure 3.1 [32], which features an Altera FPGA (5CSXFC6D6F31C8ES) and a Hard Processing System (HPS) containing a Dual-core Cortex-A9 ARM (Advanced RISC Machine) processor and several peripherals. This ARM processor is capable to run and also compile Haskell code as shall be described in section B.3. An overview of the SoCKit board can be seen in Figure B.1. Due to its numerous peripherals and interfaces, which are either accessible by the FPGA (e.g. with provided IP blocks) or directly from the HPS, this board can be deployed for multiple solutions.

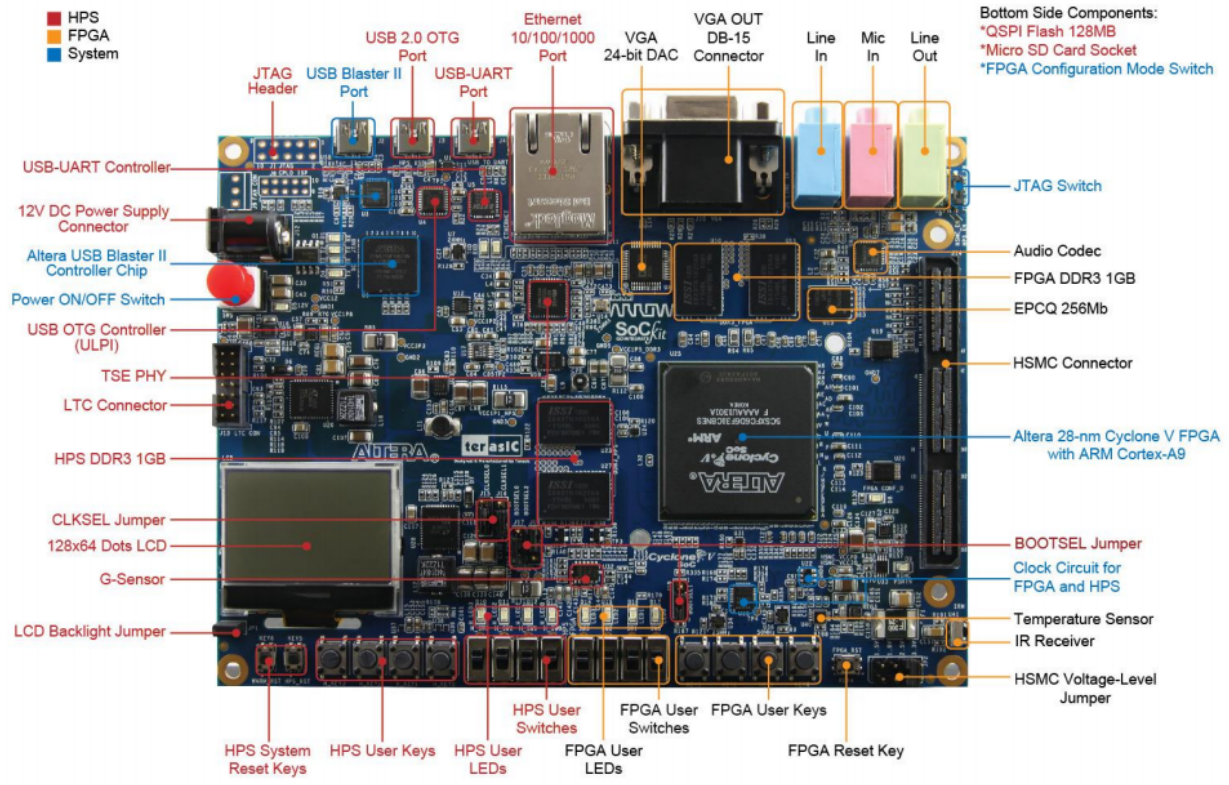


Figure B.1: SoCKit development board overview [31]

The FPGA contains 42K Adaptive Logic Modules (ALMs), 5662 Kbit embedded memory, 6 fractional Phase-locked loops (PLLs) and has access to 1GB DDR3 SDRAM. The CPU runs by default at 925MHz, has 64KB of scratch RAM and also has 1GB of DDR3 SDRAM available. The development board can be used as a standalone desktop computer with its onboard VGA video output and USB input, but in order to keep more of the system resources free it is interfaced with by means of a remote secure shell (SSH) network connection. In summary, the specifications of the SoCKit are more than sufficient to create the proof of concept for this thesis assignment.

B.1 SoC Interconnect

The HPS and FPGA fabric are connected through a set of third generation Advanced Microcontroller Bus Architecture (AMBA) bus bridges with the Advanced eXtensible Interface (AXI) protocol by ARM Ltd[33]. This bus architecture is an open-standard specifically designed for on-chip interconnections of functional blocks in SoC designs. An overview of the interconnect within the Cyclone V SoC is displayed in Figure B.2. It features three types of bridges, namely:

- HPS-to-FPGA bridge. Bus with HPS as master and configurable bus width of 32, 64 or 128 bits. Used for high bandwidth transactions.
- FPGA-to-HPS bridge. Bus with FPGA as master and configurable bus width of 32, 64 or 128 bits. Used for high bandwidth transactions.
- Lightweight HPS-to-FPGA 32 bits bridge. Used for low latency register access.

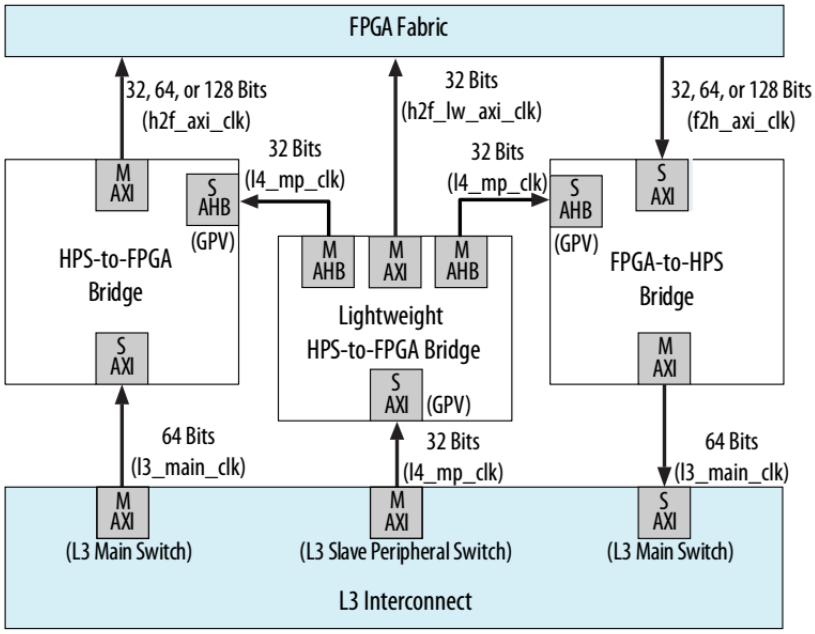


Figure B.2: Cyclone V HPS-FPGA Interconnection overview

The default method to utilize this interconnection, provided that either an Altera supplied or a customized Linux image with a board support package is used, is by setting up a system with Altera's Qsys tool in Quartus and their SoC Embedded Design Suite (EDS) [35].

B.2 HPS operating system

The Cyclone V HPS has a specific boot flow in order for it to work. As shown in Figure B.3, it starts with a minimal configuration and loading of the preloader in the scratch RAM during the 'BootROM' stage. The 'preloader' stage initializes functions such as I/Os, pin multiplexing, PLLs and other peripherals. It will then initialize the SDRAM interface such that the bootloader can be stored there. The 'bootloader' stage sets up the operating system (OS) related environment. And finally the 'Linux' stage is reached, which the user can subsequently use to start applications.

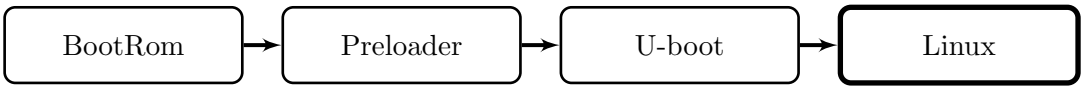


Figure B.3: Cyclone V HPS boot flow stages.

To increase performance and usability of the SoC system a SD-card of sufficient size and speed should be used[1]. The SoCKit itself comes with three basic and relatively dated Linux OS images, which for example don't contain a working package manager that could be used to install Haskell or other relevant packages. For this reason other options were investigated. There exist a number of third-party created Linux image, but there is also an option of building a custom image. In essence all SoCKit Linux images are to be created using a Golden System Reference Design (GSRD)[34] that utilizes Altera Quartus and its Qsys tool, the Yocto project[2] and Altera SoC Embedded Design Suite. With this approach it is possible to create a system based on a recent Linux distribution to get a working package manager.

B.3 Haskell on the ARM

This last section is about running Haskell programs on the SoCKit. The Haskell compiler, GHC, is also available for the ARM processor. However, the ARM processor on the SoCKit is not very efficient for compiling Haskell due to its rather small memory and low clock speed. For this reason a cross-compiler was investigated and built to compile programs on a more capable host computer. Building a cross-compiler [3] [37] consist of the first two stages as shown in Table B.1. Stage 0 is the GHC compiler installed on the host computer itself, which is a version installed from the GHC download page. Then the libraries used for building the stage 1 are downloaded and built using the stage 0 compiler. Then the actual cross-compiler at stage 1 is built, which targets the ARM processor.

	Stage 0	**libs boot**	**Stage 1**	**libs install**	**Stage 2**
Built on	—	host	host	host	host
Runs on	host	host	host	target	target
Targets	host	—	target	—	target

Table B.1: Stages involved in the building process for a GHC compiler [37]. Host is the development PC and target is the ARM processor.

[1] For this thesis a Kingston 32GB Micro SDHC Class 10 card was used.

[2] An open source project that provides templates, tools and methods for creating a custom Linux-based system for embedded products.

[3] An exemplary project of how this is done for the Raspberry Pi [36]

Cross-platform Haskell compiler building is still in a development phase and therefore some limitations and the occasional bugs are available. The most relevant restrictions are that in the stage 1 cross-compiler there can be no dynamic loading and no Template Haskell (TH) due to missing libraries, which are only available on the target platform [38]. Dynamic loading is required for CλaSH to work, due to for example a typechecker plugin that is used. TH is an extension on the GHC compiler, which is used to execute Haskell code at compile time. As we will show in a later chapter, the ability to run TH is paramount as it is used to automatically offload functions. There is at least one solution [39] to circumvent the TH problem by using an external process to run TH code, but this was deemed as too much work for this thesis project. So the only option is to have the GHC compiler on the ARM processor. This can be done by cross-compiling GHC itself to be a native GHC compiler on the ARM target processor, as can be seen in the last two columns of Table B.1. On some Linux distribution for the SoCKit platform, such as the upgraded Ubuntu image used in this implementation, there are package managers available which can install GHC (version 7.10.3) from the repository. This way, also any additional GHC packages can be installed, such as CλaSH. However, this process, and any future Haskell compilations, will take a considerable amount of time, due to the low performance of the ARM processor. A solution for this last mentioned problem would be to run the SoCKit image in a virtual machine on the host computer with the QEMU machine emulator. Mainly this allows for more memory than the standard 1GB on the SoCKit, but it also may provide a boost in processor clock frequency.

Bibliography

[1] J. Teich, "Hardware/Software Codesign: The Past, the Present, and Predicting the Future", Proceedings of the IEEE, vol.100, Issue: Special Centennial Issue, pp. 1411-1430, May 13 2012

[2] A.A. Jerraya, W. Wolf, "Hardware/Software Interface Codesign for Embedded Systems", Computer, vol.38, no. 2, pp. 63-69, February 2005, doi:10.1109/MC.2005.61

[3] Intel newsroom webpage detailing their view on FPGAs being a critical part of their growth strategy. Last seen 29-8-2016. https://newsroom.intel.com/editorials/intels-fpga-future-here-to-stay/

[4] J.G.J. Verheij, "Co-simulation between CλaSH and traditional HDLs", Master's thesis, University of Twente, August 2016 http://essay.utwente.nl/70777/

[5] W. Meeus, K. van Beeck, T. Goedemé, J. Meel, D. Stroobandt, "An overview of today's high-level synthesis tools", Design Automation for Embedded Systems, September 2012, Volume 16, Issue 3, pp. 31-51.

[6] The Haskell website. Last seen 28-4-2016. https://www.haskell.org/

[7] The Haskell 2010 Language Report. Last seen 28-4-2016. https://www.haskell.org/onlinereport/haskell2010/

[8] G.J.M. Smit, J. Kuper, C.P.R. Baaij. A mathematical approach towards hardware design. University of Twente, Enschede, The Netherlands.

[9] The CλaSH website. Last seen 28-4-2016. www.clash-lang.org

[10] Baaij, C.P.R. (2015) Digital Circuits in CλaSH : Functional Specifications and Type-Directed Synthesis. PhD thesis, University of Twente, Enschede, The Netherlands, January 2015.

[11] Educational website with information on a practical example of the GFSK demodulator and FIR filter. Last seen 29-9-2016. http://wwwhome.ewi.utwente.nl/~gerezsh/vlsidsp/index.html

[12] Haskell wiki page on annotations. Last seen 29-9-2016. https://ghc.haskell.org/trac/ghc/wiki/Annotations

[13] M. Sheeran, muFP, a Language for VLSI Design. Proceedings of the ACM Symposium on LISP and Functional Programming, 1984.

[14] J. O'Donnell, Generating Netlists from Executable Circuit Specifications in a Pure Functional Language. In Functional Programming Glasgow, Springer-Verlag Workshops in Computing, pages 178-194, 1993.

[15] P. Bjesse , K. Claessen, M. Sheeran and S. Singh, Lava: Hardware Description in Haskell. Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, 1998.

[16] A. Mycroft, R. Sharp, Hardware/Software Co-Design using Functional Languages.

[17] Getting Started with Hardware-Software Codesign Workflow for Altera SoC Platform. Last seen 29-8-2016. http://www.mathworks.com/examples/matlab-hdl-coder/mw/hdlcoder_product-hdlcoder_ip_core_tutorial_alterasoc-getting-started-with-hardware-software-codesign-workflow-for-altera-soc-platform

[18] Hardware-Software Codesign Workflow Examples. Last seen on 29-8-2016. https://www.mathworks.com/examples/matlab-hdl-coder/category/hardware-software-codesign-workflow-examples

[19] FPGA coprocessing for C/C++ programmers. Tutorial written in 2013. Last seen 29-8-2016. http://xillybus.com/tutorials/vivado-hls-c-fpga-howto-1

[20] J. Epstein, A. Black, and S.P. Jones, Haskell Symposium, Towards Haskell in the Cloud, Tokyo, Sept 2011.

[21] Cloud Haskell webpage on the Haskell wiki. Last seen 29-9-2016. https://wiki.haskell.org/Cloud_Haskell

[22] J. Armstrong, R. Virding, C. Wikström, and M. Williams, Concurrent programming in Erlang, 1993.

[23] George H. Mealy. A Method to Synthesizing Sequential Circuits. Bell Systems Technical Journal, pages 1045-1079, 1955.

[24] https://hackage.haskell.org/package/clash-prelude-0.10.3/docs/CLaSH-Prelude.html

[25] The official CλaSH tutorial page. Last seen 29-6-2016. https://hackage.haskell.org/package/clash-prelude/docs/CLaSH-Tutorial.html

[26] Baaij, C.P.R. (2009) CλaSH : from Haskell to hardware. MSc thesis, University of Twente, Enschede, The Netherlands, December 2009.

[27] Document describing the Glasgow Haskell Compiler. Dated March 16, 2012. http://community.haskell.org/~simonmar/papers/aos.pdf

[28] Haskell wiki page on the Core Type. Last seen 29-9-2016. https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/CoreSynType

[29] Haskell Core plugin user guide page. Last seen 29-9-2016. https://downloads.haskell.org/~ghc/7.10.3/docs/html/users_guide/compiler-plugins.html

[30] Arrow SoCKit development board website. Last seen on 29-8-16. https://www.arrow.com/en/products/sockit/arrow-development-tools

[31] Terasic webpage for the SoCKit Development Board. Last seen on 29-8-16. http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=816

[32] Cyclone V device specifications. Last seen on 29-8-16. https://cloud.altera.com/ds/part/5CSXFC6D6F31C8ES/

[33] Cyclone V Hard Processor System Technical Reference Manual. Last seen 28-4-2016. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_5v4.pdf

[34] Website for the SoCKit Golden System Reference Design (GSRD) documentation and sources. Last seen 28-4-2016. https://rocketboards.org/foswiki/view/Documentation/GSRD

[35] Terasic webpage for the DE1-SoC Development Board. See CD zip for the default linux images and the standard HW/SW co-design demo application. Last seen on 29-8-16. http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=836

[36] Github page on how to build a cross-compiler for a Raspberry Pi, which is quite similar to the SoCKit ARM processor. Last seen on 29-8-2016. https://github.com/ku-fpg/raspberry-pi/wiki/GHC-Cross-Compiler-for-Raspberry-Pi

[37] Haskell webpage on GHC for the ARM architecture. Last seen 29-8-2016. https://wiki.haskell.org/ARM

[38] Haskell webpage on why there can't be Template Haskell in a cross-compiler. Last seen 29-8-2016. https://ghc.haskell.org/trac/ghc/wiki/TemplateHaskell/CrossCompilation

[39] Github project for running Template Haskell code on a external process. Last seen 29-8-2016. https://github.com/ghcjs/ghcjs/wiki/Porting-GHCJS-Template-Haskell-to-GHC

[40] Xillybus homepage. Last seen 29-9-2016. http://xillybus.com/

[41] Xillybus licensing formats webpage. Last seen 29-9-2016. http://xillybus.com/licensing

[42] Haskell hackage page on Unsafe IO operations. Last seen 29-9-2016. https://hackage.haskell.org/package/base-4.9.0.0/docs/System-IO-Unsafe.html

[43] Online Xillybus IP core factory. Last seen 29-9-2016. http://xillybus.com/custom-ip-factory

[44] Error correction code in SoC FPGA-based memory systems. White paper. Last seen 29-9-2016. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01179-ecc-embedded.pdf

[45] Xillybus Linux host programming guide. Last seen 28-4-2016. http://xillybus.com/doc

[46] Bitpack package hackage page of the Clash compiler. Last seen 29-9-2016. https://hackage.haskell.org/package/clash-prelude-0.10.14/docs/CLaSH-Class-BitPack.html

[47] Altera Dual clock FIFO IP Cores User Guide. Last seen 29-9-2016. https://www.altera.com/en_US/pdfs/literature/ug/ug_fifo.pdf

[48] Source documentation webpage for the dataflow module in CλaSH . Last seen 28-4-2016. http://hackage.haskell.org/package/clash-prelude-0.10.7/docs/CLaSH-Prelude-DataFlow.html#t:DataFlow

[49] Dell webpage of the Inspiron 15 7559 laptop used in the testbenches. Last seen 29-9-2016. http://www.dell.com/en-us/shop/productdetails/inspiron-15-7559-laptop

[50] Homepage of the Criterion Haskell benchmarking library. Last seen 29-9-2016. http://www.serpentine.com/criterion/

[51] Haskell wiki page for a basic approach on timing IO computations. Last seen 29-9-2016. https://wiki.haskell.org/Timing_computations

[52] Guide to install Xillinux and use it with the demo FPGA bundle. Last seen 29-9-2016. http://xillybus.com/xillinux

[53] The permanent GitHub repository with the code and in-depth standalone user manual of the proof of concept of this thesis. https://github.com/jjvanvossen/Haskell-FunctionOffloading