MASTER THESIS

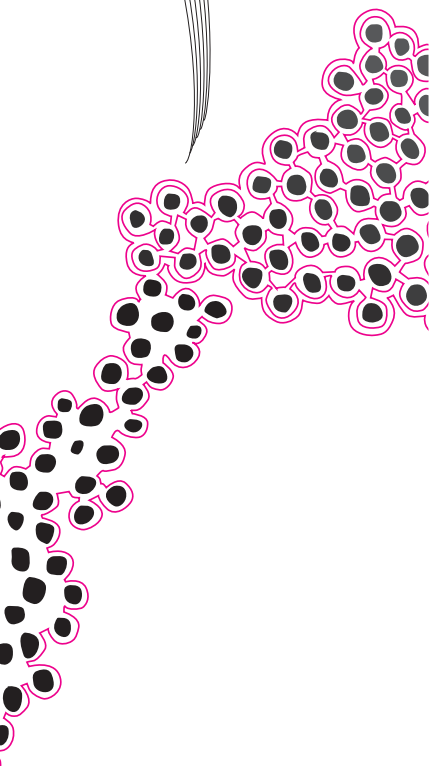
CO-SIMULATION BETWEEN CλASH AND TRADITIONAL HDLS

Author:
John Verheij

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
Ir. E. Molenkamp

August 19, 2016



UNIVERSITY OF TWENTE.

Abstract

CλaSH is a functional *hardware description language* (HDL) developed at the CAES group of the University of Twente. CλaSH borrows both the syntax and semantics from the general-purpose functional programming language Haskell, meaning that circuit designers can define their circuits with regular Haskell syntax.

CλaSH contains a compiler for compiling circuits to traditional *hardware description languages*, like VHDL, Verilog, and SystemVerilog. Currently, compiling to traditional HDLs is one-way, meaning that CλaSH has no simulation options with the traditional HDLs.

Co-simulation could be used to simulate designs which are defined in multiple languages. With co-simulation it should be possible to use CλaSH as a verification language (test-bench) for traditional HDLs. Furthermore, circuits defined in traditional HDLs, can be used and simulated within CλaSH.

In this thesis, research is done on the co-simulation of CλaSH and traditional HDLs. Traditional *hardware description languages* are standardized and include an interface to communicate with *foreign* languages. This interface can be used to include foreign functions, or to make verification and co-simulation possible.

Because CλaSH also has possibilities to communicate with foreign languages, through Haskell *foreign function interface* (FFI), it is possible to set up co-simulation. The *Verilog Procedural Interface* (VPI), as defined in the IEEE 1364 standard, is used to set-up the communication and to control a Verilog simulator. An implementation is made, as will be described in this thesis, to show the practical feasibility of co-simulation of CλaSH and Verilog[1].

The *VHDL Procedural Interface* (VHPI), as defined in the IEEE 1067 standard, is less popular compared with the VPI. Furthermore, not every VHDL simulator gives support for the VHPI. For example, ModelSim and QuestaSim use a different interface. The VHPI is set up in the same way as the VPI. The expectation is that co-simulation through the VPHI can be implemented in a comparable way.
GHDL, an open-source VHDL simulator, does however give support for the VPI and this interface could be used to set-up co-simulation between CλaSH and VHDL.

[1]The VPI can also be used to define co-simulation with SystemVerilog.

The co-simulation supports both combinational and synchronous sequential designs. A combinational circuit does not contain memory elements and the output can be seen as a pure function of the present input. This is in contrast to a synchronous sequential design, in which the output also depends on the history of the input.

Within a synchronous sequential circuit, the changes in the state of the memory elements are synchronized by a clock signal. A clock is a periodic signal, in which every period is called a clock-cycle, as shown in Figure 1. A clock-cycle consists of multiple simulation steps. A simulation step can be associated with a certain time indication in the Verilog code, for example 1 nanosecond.

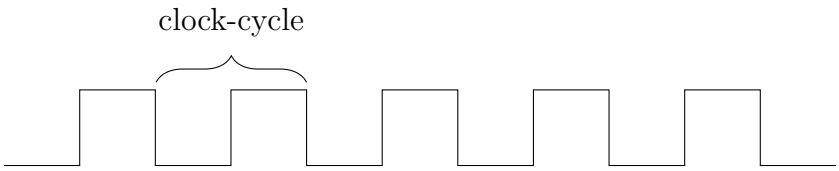


Figure 1: A clock signal

In typical *Register Transfer Level* (RTL) code, some events cause other events to occur in the same simulation step. For example, when a clock signal triggers, some signals may change in the same simulation step. To ensure race-free operations, HDLs must differentiate between such events with so-called 'delta' delays. However, CλaSH does not work with delta-delays and signals are defined per clock-cycle.

The defined co-simulation tries to seamlessly communicate with the traditional HDL and thus clock-cycles have the same length in both HDLs. A clock signal will change twice in one clock-signal. The recommendation is to define the clock-signals in the traditional HDL and only exchange the '*functional*' values.

Support for (feedback) loops is created with lazy-evaluation. The implementation uses the *IO-monad* to control and to communicate with the traditional HDL simulators. Lazy IO often has as disadvantage that releasing acquired resources is unpredictable. Foreign memory allocations are connected to the Haskell's *Garbage Collector* (GC). With the use of the GC, foreign functions are invoked when a resource is released, in which a particular co-simulation is finished and the traditional HDL simulator is closed.

With *Quasiquotation*, *Inline-Verilog* is made possible. By using a *Quasiquoter* it is possible to define a *Domain Specific Language* (DSL). With *Inline-Verilog* it is possible to embed Verilog modules in CλaSH or to create a wrapper in which sub-modules (defined in verilog files) are included.

Acknowledgements

After doing the *HBO* study '*Technische Informatica*' at Windesheim (Zwolle), I decided to do the Master study '*Embedded Systems*'. Input for this decision was conversations with my family, but also discussions with my piano teacher. Looking backwards, I am very happy for this decision and my knowledge, related to *Embedded Systems*, is very extended.

First of all I would like to thank Jan (Kuper) and Christiaan, for offering me this master thesis project and providing support. A year ago my knowledge about functional programming was very limited and I did not expect to learn so much in this field. I am very excited about CλaSH and the '*new*' possibilities on how to program FPGAs.

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

Joris, thanks for the conversations and the time we spend together during this master. I very liked our discussions, which also gave inspiration for this master thesis.

Guus and Rinse, for being my *day-time* room-mates and the conversations about holidays and technical things. The jokes we made together and the fun we had.

Harm, for the discussions on how to improve my implementation.

Bert Molenkamp, for always having time for checking & changing my study progress documents and the insights about VHDL & Verilog. And of course, for being part of my exam commission.

All members of the CAES group, thanks for the conversations and the time together during the breaks. This group also gave me a good impression about performing research and doing a PhD.

Finally I would like to thank my family and girlfriend for their support during my master and this thesis. Although we did not have much conversations about the technical part; I much appreciate the help with for example my car and all the other needed things.

John,

't Harde, August 2016

Acronyms

CλaSH	CAES Language for Synchronous Hardware
CAES	Computer Architecture for Embedded Systems
Cocotb	COroutine based COsimulation TestBench
DPI	Direct Programming Interface
DSL	Domain Specific Language
EDA	Electronic Design Automation
FFI	Foreign Function Interface
FLI	Foreign Language Interface
FPGA	Field programmable Gate Array
GHC	Glasgow Haskell Compiler
GHDL	G Hardware Design Language
HDL	Hardware Description Language
HVL	Hardware Verification Language
IVI	Icarus Verilog Interactive
MAC	Multiply ACcumulate
PLI	Program Language Interface
RTL	Register Transfer Level
TH	Template Haskell
VHDL	VHSIC Hardware Description Language
VHPI	VHDL Procedural Interface
VHSIC	Very High Speed Integrated Circuit
VPI	Verilog Procedural Interface

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

A *Field Programmable Gate Array* (FPGA) is a reprogrammable silicon chip, which can be configured with prebuilt logic blocks and programmable routing resources to implement custom hardware functionality. Unlike processors, FPGAs are truly parallel in nature and different processing operations can function deterministically and do not have to compete for resources [21].
A processor-based system often consists of several layers of abstraction, to perform scheduling tasks, and share resources for multiple processes. A processor core can only execute a few instructions at a time (instruction pipelining) and processor-based systems have continously the risk of time-critical tasks pre-empting each other [21].

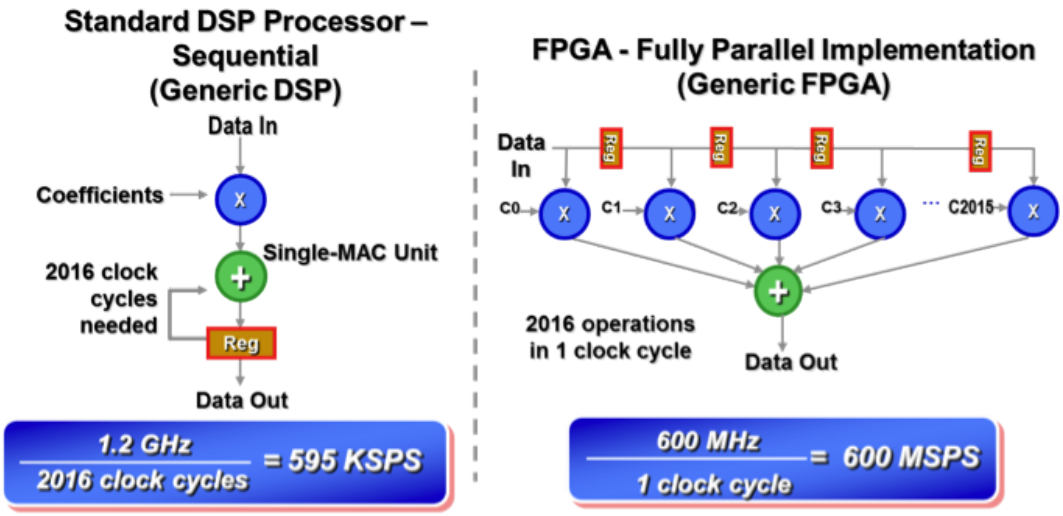


Figure 1.1: Sequential versus parallel implementation of a tap filter [21]

As visible in Figure 1.1, FPGAs can be used to speed up calculations, because of their parallel nature. Using such accelerators often leads to discussions of using FPGAs versus using *Graphics Processing Units* (GPUs). GPUs are mainly known for doing calculations related to 3D computer graphics, but GPUs are also used to accelerate scientific, analytics, engineering, consumer, and enterprise applications [22].

A paradigm shift is going on related to metric for procurements, system design, and application development. In the past *Floating-point Operations Per Second* (FLOPs) were used as measure for computer performance, which is now changing to FLOPs per watt. Energy costs become more and more important, and *performance at any cost* is no longer viable. The operational costs of supercomputer clusters become more dominant compared with the acquisition costs [24].

Research to device-energy-efficiency, related to application performance, showed that FPGAs are more energy efficient compared to GPUs, but FPGAs are considered hard to program [23]. Organisations, like Microsoft's Bing, are investing in FPGA-based codings, because of power efficiency. Microsoft Research touted that a *convolutional neural networks* (CNNs) implementation on a FPGA can achieve a three times better performance-to-power advantage compared to running on GPUs [24].

In the past, FPGA technology was only available for engineers with solid understanding of digital hardware design. With the rise of high-level tools and *Hardware Description Languages* (HDLs), new technologies became available to convert graphical block diagrams or even languages like C or Python into a digital hardware circuit.

CλaSH is a high-level functional HDL, which borrows both the syntax and semantics from the functional programming language Haskell. With this language, hardware can be described and compiled to traditional HDLs, like VHDL, Verilog, and SystemVerilog.

Co-simulation of CλaSH and traditional HDLs is not supported currently. With co-simulation, different subsystems can be simulated in a distributed manner. The execution of these subsystems, and the data-exchange between them, often happen in a black-box manner. This gives the user the idea of simulating, for example, two different languages in one system.

In this thesis, the issue of supporting co-simulation between CλaSH and traditional HDLs is addressed. The main focus will be on co-simulation with Verilog, because the *Verilog Procedural Interface* (VPI), part of the IEEE 1364 standard, is popular and widely accepted. For example with VHDL, simulators use different standards and in some cases licenses are needed.

Defining co-simulation between a functional and a traditional HDL is more complicated than defining co-simulation between two different traditional HDLs. In a functional HDL like CλaSH, there is no notion of delays and clocks are only used as annotations for signals. Signals are simulated as streams of values, in which transitions happen on the same moments. Furthermore, the co-simulation is only possible through the C-programming language, which does not support the same data types as in CλaSH and memory management has to be implemented manually. The two HDLs has to be synchronized and the lazy evaluation of signals has to be transformed to the appropriate simulation cycles.

1.1 Problem statement and approach

The ability to perform co-simulation with (traditional) HDLs is desirable within CλaSH. In other high-level HDLs, like *MyHDL* and *Cocotb*, co-simulation is possible to a certain level.
CλaSH does not have the functionality to perform co-simulation with, for example, VHDL or Verilog currently. This is considered as a shortcoming by some CλaSH users [25].

In this thesis, research is conducted to the *Verilog Procedural Interface* and the Haskell standard, to be able to support co-simulation. The research question central to this thesis will therefore be:

≫ *How can co-simulation with traditional HDLs be supported within CλaSH?*

This central research question gives rise to other questions and design choices, such as: which information has to be exchanged between CλaSH and a traditional HDL? Does the IEEE 1364 standard give possibilities for this communication and how can a simulation/simulator be controlled?

Furthermore, the possibilities and limitations of a (theoretical) co-simulation with CλaSH has to be identified. For example, traditional HDLs uses delta-delays to ensure race-free operations. CλaSH, on the other hand, uses a functional approach and support for (delta-)delays is very limited.

Finally, an implementation, using the VPI, will be made to show the feasibility of the co-simulation between CλaSH and Verilog.

1.2 Outline

In chapter 2, the background and related work with respect to co-simulation is presented. This chapter also gives a background on CλaSH. The *Foreign Function Interface* (FFI) and *QuasiQuotation*, both part of the Haskell language, are explained. Furthermore, related work, like *Cocotb* and *MyHDL*, is described. In the last part of this chapter, research is performed about the available Verilog and VHDL interfaces.

In chapter 3, the *Verilog Procedural Interface* (VPI) is presented. This interface is part of the IEEE 1364 standard and gives possibilities for communication with the C programming language. The needed information for defining co-simulation, like synchronizing with the simulator and data exchange, is explained in detail.
Chapter 3 and sections from chapter 2, the related work and the Verilog/VHDL interfaces, are described using literature study. The book [1] and the IEEE standards are used as the main resources for this research.

Chapter 4 shows how co-simulation between CλaSH and *Icarus Verilog*, an open-source Verilog simulator, could be implemented by using the VPI standard. Design choices are explained and additional information about Haskell's FFI and QuasiQuation possibilities are given.

In chapter 5, results of the implemented co-simulation are presentated and benchmarks are shown. Furthermore, the implementation is tested with other simulators, ModelSim and GHDL, to show the portability of the co-simulation. An implemented GFSK demodulator is compiled to Verilog and with the co-simulation compared to the CλaSH implementation.

In chapter 6, recommendations and ideas for future work are presented. One of the main points will be co-simulation with other HDLs and the feasibility of the needed implementations. Furthermore, ideas on how to give support for multiple clock domains are given. Recommendations with respect to *Inline Verilog* are made. *Inline Verilog* is one of the final goals of co-simulation, which gives support for embedding Verilog, as *Domain Specific Language*, in CλaSH.

Finally, in chapter 7, conclusions are drawn and a small evaluation of this master project is made.

2 Background

Hardware Description Languages (HDL) describe the structure and behavior of electronic circuits. The description of a circuit can be synthesised into a *netlist*, which can be *placed & routed* to produce the *mask set*, used to create an integrated circuit.

In this master thesis, the focus will be on a *sub-set* of the *Hardware Description Languages*, which is needed to define co-simulation between CλaSH and traditional HDLs. Traditional HDLs, like VHDL, Verilog, and SystemVerilog, are standardized. These standards describe interfaces to communicate with foreign languages, which can be used to define co-simulation. Standards are needed to increase the quality and also the compatibility to work with different simulators, like Icarus Verilog and Modelsim.

HDLs are also defined in *high level* languages like Python and C. Popular implementations are *MyHDL* and *Cocotb*, both written in Python. MyHDL is defined as an HDL and a *Hardware Verification Language* (HVL), having co-simulation possibilities with Icarus Verilog and Cver (both open-source Verilog-simulators). Cocotb is mainly defined as a HVL and has support for a wide range of simulators.

In 2011 it was stated that verification consumes approximately 75% of the design resources and time scheduling [34]. *Electronic Design Automation* (EDA) companies try to increase the verification productivity with foreign interfaces, which can be used for co-simulation.

The options for co-simulation depends heavily on the available interfaces. The *Verilog Procedural Interface* (VPI) is the commonly used and recommended interface for Verilog, mainly because it is standardized and supported by many Verilog tool vendors. The *Programming Language Interface* (PLI 1.0), the precursor of VPI, is also still commonly used because of its widely documented interface.

For VHDL there are fewer possibilities compared to Verilog. Vendors often provide functionality that is similar to the VPI, but these are mainly vendor specific interfaces. An example is the *Foreign Language Interface* (FLI) of Modelsim.
The recommended interface is the standardized *VHDL Procedural Interface* (VHPI), but this interface is not as popular as the VPI and there is less support from vendors [33].
The *Direct Programming Interface* (DPI) is available for SystemVerilog, but the VPI is also fully supported.

2.1 CλaSH

CAES Language for Synchronous Hardware (CλaSH) is a functional hardware description language defined within the *Computer Architecture for Embedded Systems* (CAES) group (University of Twente). CλaSH borrows both the syntax and semantics from Haskell, a general purpose functional programming language. Designers can define their circuits with regular Haskell syntax and use strong typing & higher order functions.

A functional programming language treats computation as the evaluation of mathematical functions. Combinational circuits can often be modelled as mathematical functions. Within CλaSH, functions are used to describe hardware. As in Haskell, a set of functions are provided in the CλaSH prelude library. With these functions, both combinational and synchronous sequential hardware can be designed.

An important type within CλaSH is the *Signal* type, an *infinite* stream of values used in a synchronous sequential circuit design. For example the *Mealy* machine, a classic machine model, uses the Signal types. The mealy machine is defined as the function *mealy* in the CλaSH prelude library.

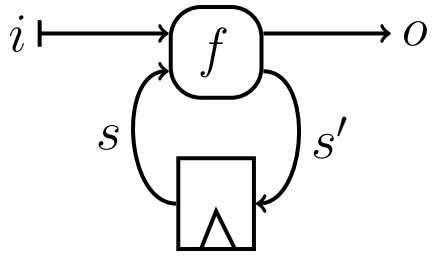


Figure 2.1: The mealy machine hardware representation

The input i and the output o are both signals. The signal i and the state s is fed into the function *f*, from which the updated state *s'* and signal o is defined:
This is also visible in the type definition of the *mealy* function.

$$mealy :: (s \to i \to (s, o)) \to s \to Signal\ i\ \to Signal\ o$$

The identifiers s, i, and o, used in the type definition of the *mealy* function, denote the types of input and output variables. This is in contrast to Figure 2.1, where they denote several variables. The first argument is a function, called *f*, which will be executed by the *mealy* function. The function f has as input a state with the type s and a input with the type i. Furthermore, f produces a tuple containing a updated state, with the same type s, and a output with the type o.
The second argument of the *mealy* function is given as initial state to the function *f*, after which f is mapped over the signal i to produce the signal o.

The function f is a combinational function, in which the output directly depends on the input without any memory elements. An example of such a function is the *Multiply Accumulate* (MAC) operation. Within the MAC operation, two values will be multiplied together and summed with the previous output value, as visualized in Figure 2.2. The CλaSH definition of the mealy machine with the MAC operation is shown in Listing 2.1.

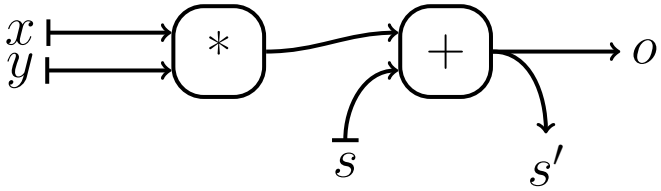


Figure 2.2: The multiply accumulate operation

```
topEntity :: t ~ Signed 16 => Signal (t, t) -> Signal t
topEntity      = mealy mac 0

mac s (x,y)    = (s', o)
    where
        s'     = x * y + s
        o      = s'
```

Listing 2.1: The *mealy* function with the MAC operation in CλaSH

The types for the function *topEntity* are defined explicitly. In this case every sample, inside the signals, is defined as a 16 bits wide signed value. Specifying the types with their sizes (bounds) is needed to make compilation to synthesizable code possible. Currently the CλaSH compiler supports the generation of VHDL, Verilog, and SystemVerilog code.

The function *mealy* is called a *higher-order* function, because it takes a function as an argument. Other examples of higher-order functions are: *map*, *zipWith*, *foldl*, *scanl*, and *mapAccumL*, as visualized in Appendix A.

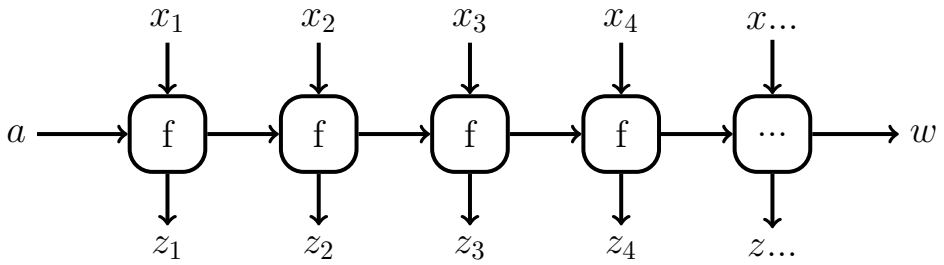


Figure 2.3: The *mapAccumL* function

The function *f*, used in the *mapAccumL* function, has as input an initial state a and the value x_i. It produces the output value z_i and an updated state *a'*, which is passed to the next f function.

Within CλaSH, the full Haskell environment can be used. This environment can be useful for extending the simulation possibilities in CλaSH (for example reading inputs from file), but for defining co-simulation it is actually a requirement. For example, the execution of C code, as will be explained in subsection 2.1.1, is needed to communicate with a (Verilog) simulator.

Haskell is a *pure* language, meaning that functions will always return the same outputs for the same inputs, without any side-effects. But it is possible to define *impure* functions with the usage of the so called *monads*. Monads allow a programmer to define computations using sequential building blocks. For example, the *Maybe* monad represent computations which may fail and the *List* monad represent computations with multiple values.

To define co-simulation, the IO monad becomes important. This monad makes (low-level) IO operations possible. Many of the possible functions in the IO monad are not *pure*, because they depend on the *external world* (e.g. operation system or other processes). The *do notation* is often used to combine multiple (IO) statements, as show in Listing 2.2.

```
main :: IO ()
main = do
        putStr "What is your name?  "
        getLine >>= putStrLn . (P.++) "Hello "
```

Listing 2.2: The function *main* with the *do notation*

The function *putStr* will write *"What is your name? "* to the standard output, after which the function *getLine* is used to read a line from the standard input. The function *getLine* return an *IO String*, but with the pipe forward operator (>>=) the encapsulated string is forwarded to the *putStrLn* function. The (.) operator denotes function composition. The string *"Hello "* will be concatenated with the input string, after which they are written to the standard output.

```
λ> main
What is your name?  John
Hello John
```

Listing 2.3: The execution of the *main* function

To define a *Domain Specific Language* (DSL) in Haskell, the *Q* (abbreviation of Quotation) monad can be used. This monad is part of Template Haskell, as will be explained in subsection 2.1.2. A DSL could be interesting to include Verilog or VHDL code in the CλaSH code.

2.1.1 Foreign Function Interface

To define co-simulation, CλaSH has to perform IO operations, which can be done using the IO monad. Examples of IO operations are the creation of processes (e.g. the execution of the Verilog/VHDL simulator) and defining pipes to these processes. The Haskell package *process* contains libraries for dealing with these system processes, and gives support for IO operations [12]. This package and its dependency, the *unix* package, internally use the *Foreign Function Interface*. For example, the function *createProcess* is used to spawn an external process and the function *createPipe* is used to create a pipe for interprocess communication.

The *process* package does not provide enough functionality to control and communicate with a (traditional) HDL-simulator. As will be explained in section 2.3, the control and communication will not be done directly with the simulator, but through an interface; which is compiled to object code and loaded into the simulator at runtime.

The *Foreign Function Interface* (FFI) is an extension to the Haskell language and can be used to communicate with *'foreign'* languages (often C). The FFI is introduced by the 'Haskell 2010' revision and from then on further improvements are made [14].

The interface contains two concepts: the possibility to import and to export function(s). As example, to import a C function *exp* into Haskell, with as input and output argument a *double*, the following code can be defined:

```
foreign import ccall "exp" c_exp :: CDouble → CDouble
```

Listing 2.4: The imported Haskell-function *c_exp*

In this case the function *c_exp* acts like a normal Haskell function, but actually it calls the C function *exp* and returns the result of that function.
As a remark, in older versions of GHC (6.8.3 and earlier) it was needed to include the header file (in this case 'math.h').

Only C code, which is compiled to an object file, can be imported [13]. Furthermore, the option *Position Independent Code* (PIC) must be given to the C compiler. Code compiled with *-fPIC* is suitable for inclusion in a library, because the library must be relocated from its preferred location in memory to another address. The created object file must be compiled into a shared library, after which it can be loaded into the interpretor.

```
gcc -O2 -fPIC -c -Wall -o ffi.o ffi.c
gcc -O2 -shared -o libffi.so ffi.o
clash --interactive MyDesign.hs -lffi -L/tmp/ffi
```

Listing 2.5: The commands to compile C code into a shared library

The export functionality looks similar; instead of using the keyword *import*, the keyword *export* will be used. Listing 2.6 shows an example of an exported Haskell function.

```
triple :: Num a ⇒ a → a
triple = (∗3)

foreign export ccall triple :: CInt → CInt
```

Listing 2.6: The exported Haskell function *triple*

Although the function *triple* will look like a normal C function in the C code, it is a binding to the Haskell function. The Haskell runtime environment must be initialized (with a call to *hs_init*) and released (with a call to *hs_exit*), in order to use the function *triple*.

Loading the code, as shown in Listing 2.6, in the interactive interpreter without any C code will give the error: *"Illegal foreign declaration: requires unregisterised..."* . GHCi fails to link, because object-files are missing. By default, GHCi only generates byte-code. The flag *-fobject -code* can be used to have GHCi generate object-code instead of byte-code [15].

Besides using *objects*, like integers, it is also possible to use *pointers*. Within Haskell, a pointer will have the type *Ptr a*. The type *a* is often an instance of the class *Storable*, which provides marshalling operations. The FFI library also contains marshalling functions, for example to marshall Haskell lists to and from C arrays.

The *Ptr a* is used as pointer to a foreign object. To use pointers to functions, the type *FunPtr a* can be used. This pointer is callable from the foreign code, but with a *dynamic* stub the FunPtr can be converted to a corresponding Haskell function.

```
type Func a = Ptr a -> IO ()
foreign import ccall "&free" p_free :: FunPtr (Func a)
foreign import ccall "dynamic" c_dynamic :: FunPtr (Func a) → (Func a)
```

Listing 2.7: The imported FunPtr *p_free*

Function pointers can be used to connect *finalizers* to *object* pointers. Normally foreign objects are not managed by the Haskell storage manager (garbage collector) and the memory management has to be performed manually. By creating a *ForeignPtr*, the finalizer will be executed after the last reference to the foreign object is dropped.

```
type FinalizerPtr a = FunPtr (Ptr a → IO ())
newForeignPtr :: FinalizerPtr a → Ptr a → IO (ForeignPtr a)
```

Listing 2.8: The *newForeignPtr* function

In the co-simulation implementation, as will be explained in subsection 4.2.3, C functions will be imported and finalizers will be used to interact and control a Verilog-simulator.

2.1.2 Template Haskell & QuasiQuotation

Template Haskell (TH) is a *Glasgow Haskell Compiler* (GHC) extension, which can be used in two main areas of application: compile-time meta programming and embedding domain specific languages [56][57].

Template Haskell provides features to convert between concrete syntax (normal Haskell code) and *Abstract Syntax Trees* (AST). Haskell values can be spliced into an AST and be manipulated at compile time. The abstract syntax tree can be spliced back into the concrete syntax [56][57].

TH programs are built inside the *Quotation* monad Q. Within Haskell, state monads like Q and IO are used to perform operations which can have side effects. Within the abstract syntax tree, a program is described using algebraic data types. The TH library provides the following algebraic data types: *Exp*, *Pat*, *Dec*, and *Type*, which can be used to represent expressions, patterns, declarations, and types, respectively. A possible operation within the Q monad is the name-generation operation, as shown in Listing 2.9 [53].

```
f = $(do
  nm1 ← newName "x"
  nm2 ← newName "y"
  return $ LamE [VarP nm1, VarP nm2] $ UInfixE (VarE nm1)
                                               (VarE $ mkName "+") (VarE nm2))
```

Listing 2.9: A *Template Haskell* example

The function *f* performs an addition, which is implemented in Template Haskell. The functions *newName* and *mkName* functions are used to generate names. The name, as generated with the function *mkName*, can be captured; this in contrast to the function *newName*. Capturing is needed for the '+' operator, to make the summation of the two arguments possible. Finally, the abstract syntax tree is spliced back into concrete syntax using Template Haskell's splice operator $.

LamE, *VarE*, and *UInfixE* are used as expression constructors. *LamE* can be used as a lambda expression, *VarE* for defining a variable, and *UInfixE* for performing an operation. *VarP* is used as pattern constructor [53].

Constructor	Expression
VarE name	x
UInfixE *Exp Exp Exp*	$x + y$
LamE [*Pat*] *Exp*	$\backslash p1\ p2 \rightarrow e$
TupE [*Exp*]	$(e1,\ e2)$

Table 2.1: Template Haskell's *Exp* constructors [53]

A *QuasiQuoter* is essentially a function which takes a string to an *Abstract Syntax Tree* (AST) [54]. *QuasiQuotation* is often used to write *Domain Specific Languages* (DSL). A parser, defined as *QuasiQuoter*, will transform a string into an AST during compile time.

Template Haskell defines the *QuasiQuoter* data type. This data type is actually a record with four *QuasiQuators*, defined for the four algebraic data types, as shown in Listing 2.10.

```
data QuasiQuoter = QuasiQuoter {
  -- | Quasi-quoter for expressions, invoked by quotes like lhs = $[q|...]
  quoteExp  :: String → Q Exp,
  -- | Quasi-quoter for patterns, invoked by quotes like f $[q|...] = rhs
  quotePat  :: String → Q Pat,
  -- | Quasi-quoter for types, invoked by quotes like f :: $[q|...]
  quoteType :: String → Q Type,
  -- | Quasi-quoter for declarations, invoked by top-level quotes
  quoteDec  :: String → Q [Dec]
  }
```

Listing 2.10: The data type *QuasiQuoter* [54]

The most trivial example is to immediately lift a string in the Q monad, as shown in Listing 2.11. The function *ex*, as shown in Listing 2.12, shows the execution of this QuasiQuoter.

```
qq :: QuasiQuoter
qq = QuasiQuoter { quoteExp = stringE }
```

Listing 2.11: An example QuasiQuoter [52]

```
{-# LANGUAGE QuasiQuotes #-}

ex :: String
ex = [qq|Hello|]

ex' :: String
ex' = $(quoteExp qq "Hello")
```

Listing 2.12: Execution of the example QuasiQuoter [52]

The expression quotation, as shown in the function *ex*, is written in the so called *Oxford brackets* and automatically spliced in the concrete syntax. This function is equivalent to the function *ex'*, in which the result of the function *quoteExp* is spliced back with Template Haskell's splice operator $. The syntax used in the function *ex* is somewhat more convenient and Haskell will automatically pick the right parser for the context [52].

More advanced QuasiQuoters support meta-variables. The convention is to use the keyword $ to splice Haskell variables into a quotation. Constructors like *VarE* and functions like *mkName*, as shown in Listing 2.9, can be used to parse the meta-variables.

2.2 Related Work

Co-simulation with the standardized *Hardware Description Languages* is also implemented in related work, as will be shown in this chapter. In some cases, simulators are using co-simulation to add, for example, a graphical viewer. This is the case with GHDL, a VHDL simulator, which uses *Icarus Verilog Interactive* (IVI) as graphical viewer [40]. Matlab also defines co-simulation possibilities with ModelSim/QuastaSim and Incisive (Cadence) [43].

Looking at open-source co-simulation possibilities, it becomes visible that most of the implementations are defunct and not under development any more. Examples of these projects are: *PyHVL* (Python) [44], *Ruby-VPI* (Ruby) [45], *Jove* (Java) [46], and the implementation of *Andre Pool* [18]. However, *MyHDL* and *Cocotb* are still under development as will be described in the following two sections.
Most of the projects use the *Verilog Procedural Interface*, which is supported by multiple simulators, for defining co-simulation with Verilog and SystemVerilog.

The implementation of *Andre Pool*, with the title '*Using ModelSim Foreign Language Interface for C*', only uses the *Foreign Language Interface* (FLI) of ModelSim, as visualized in Figure 2.4 [18].

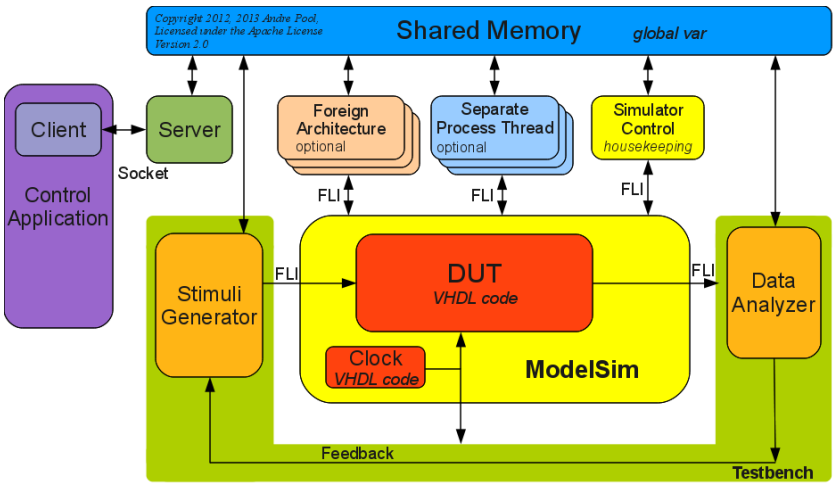


Figure 2.4: Co-simulation between C and VHDL using the FLI [18]

The red parts denotes a clock and the *Design Under Test* (DUT), both implemented in Verilog. However, the *TestBench*, consisting of a *Stimuli Generator* and a *Data Analyzer*, is implemented in C and connected through the FLI with the Verilog code.
The VHDL can also execute C code, defined as *Foreign Architectures* and *Separate Process Threads*, which are also connected through the FLI.
The other blocks are used to control the simulation and to do the memory management.

2.2.1 MyHDL

MyHDL is a hardware description and verification language written in Python. This package is free and open-source. The main goal of MyHDL is modelling and simulation. To some limitations, verification (using co-simulation) and conversion to Verilog and VHDL can be performed [47].

A key concept in MyHDL, but also in other Python-based HDLs, are generators. A generator is an object which contains one or more *yield* calls, and can be called iteratively with its *next* method. The MyHDLs manual gives the following example [47]:

```
def generator():
  for i in range(3):
    yield i
```

Listing 2.13: An example generator [47]

```
>>> g = generator()
>>> g.next()
0
>>> g.next()
1
>>> g.next()
2
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
```

Listing 2.14: Executing the example generator [47]

The yield values can be made sensitive to values or time, and are used as general sensitivity lists. This can be extended by the use of decorators, a special keyword in front of a function. With this keyword a function can be transformed into a generator. Generators are used to model concurrency. The following examples shows this in more detail.

```
def ClkDriver(clk):
  halfPeriod = delay(10)

  @always(halfPeriod)
  def driveClk():
    clk.next = not clk

  return driveClk
```

Listing 2.15: A clock driver example [47]

```
def HelloWorld(clk):

  @always(clk.posedge)
  def sayHello():
    print "%s Hello World!" % now()

  return sayHello
```

Listing 2.16: A hello world example using the clock driver [47]

```
clk = Signal(0)
sim = Simulation(ClkDriver(clk), HelloWorld(clk))
sim.run(50)
```

Listing 2.17: Executing the hello world example [47]

The decorator *@always* makes the functions sensitive to certain inputs. The function *ClkDriver* is made sensitive to a period of 10 time steps and the function *HelloWorld* is made sensitive to the positive edge of a clock.

Co-simulation between MyHDL and Verilog is defined using the VPI. Only two Verilog simulators are currently supported: Icarus Verilog and Cver. The *regs* and *nets*, which will be used within the co-simulation, must be registered with the functions *$from_myhdl* and *$to_myhdl*, as visualized in Listing 2.18 [48].

```
module dut_bin2gray;
    reg [`width-1:0] B;
    wire [`width-1:0] G;

    initial begin
        $from_myhdl(B);
        $to_myhdl(G);
    end

    ...
endmodule
```

Listing 2.18: Registration of signals in Verilog [47]

A *Cosimulation* object must be created in the Python code. This object will compile the verilog sources and execute the Verilog-simulator. Furthermore, the Python signals will be connected to the Verilog signals, as visible in Listing 2.19.

```
def bin2gray(B, G):
    # compile of Verilog files
    os.system(cmd)
    # start simulator and connect signals
    return Cosimulation("vvp -m ./myhdl.vpi bin2gray", B=B, G=G)
```

Listing 2.19: Cosimulation object in Python [47]

The co-simulation possibilities in MyHDL have some limitations [48]. First of all, both languages have to register the signals which will be used in the co-simulation. A more desirable approach would be to only register signals at the Python side and automatically recognize the ports of the top-level module in the Verilog-simulator.

Furthermore, only *passive* Verilog code can be used in the co-simulation, meaning that the Verilog code cannot contain any delay statements. One of the consequences is that clocks must be defined at the Python side [48].

Delta-cycles are only preserved from the MyHDL simulator towards the Verilog-simulator, but not in the opposite direction. Delta-cycles are implemented by making the time granularity in the Verilog simulator a 1000 times smaller than in the MyHDL simulator; meaning that for each MyHDL time step, 1000 Verilog steps are available for MyHDL delta-cycles. The value of 1000 steps is used, because the need for only a few delta-cycles per time step is assumed. Only after performing the 1000 steps, signal changes are returned to the MyHDL simulator [48].
It is unclear why 1000 simulation steps are used, theoretically it seems that less simulation steps would already be sufficient.

Only co-simulation with Verilog is possible and the development for co-simulation with VHDL is on hold (currently). In the MyHDL manual three requirements are given to be able to support co-simulation with VHDL:

1. A procedural interface to the internals of the simulator is needed.
2. The procedural interface should be a widely used industry standard.
3. The VHDL-simulator should be an open-source simulator.

Only the VHPI standard matches this requirement as a procedural interface, but this interface is less popular (compared with the VPI). Furthermore, there is only one credible open-source VHDL simulator (GHDL) and it is unclear whether it has VHPI capabilities that are powerful enough to support co-simulation [48].

An ideal (theoretical) co-simulation implementation between CλaSH and Verilog should not have the limitations which are described in this section. CλaSH should be able to automatically iterate through the Verilog hierarchy and scan for available input and output ports. These ports should be automatically mapped to CλaSH signals. Furthermore, it should be possible to define delay statements (e.g reset and clock streams) in Verilog, and use CλaSH as a pure functional HDL.

2.2.2 Cocotb

COroutine based COsimulation TestBench (Cocotb) is an environment for verifying VHDL/Verilog RTL using Python. Cocotb uses the VPI for Verilog and VHPI/FLI for VHDL, and gives support for the following simulators [49]:

- Icarus Verilog
- Aldec Riviera-PRO
- Synopsys VCS (only Linux)
- Cadence Incisive (only Linux)
- Mentor Modelsim

A cocotb testbench does not require additional VHDL/Verilog code. The so called Design Under Test (DUT) is instantiated as the toplevel in the simulator without any wrapper. From the python code, cocotb can drive stimulus into the inputs of the DUT and monitor the outputs.
The test environment can be fully defined in Python and the tests are simply Python functions (*coroutines*). With the *yield* keyword (see Listing 2.13) the control of execution can be switched between the simulator and the Python code [49].

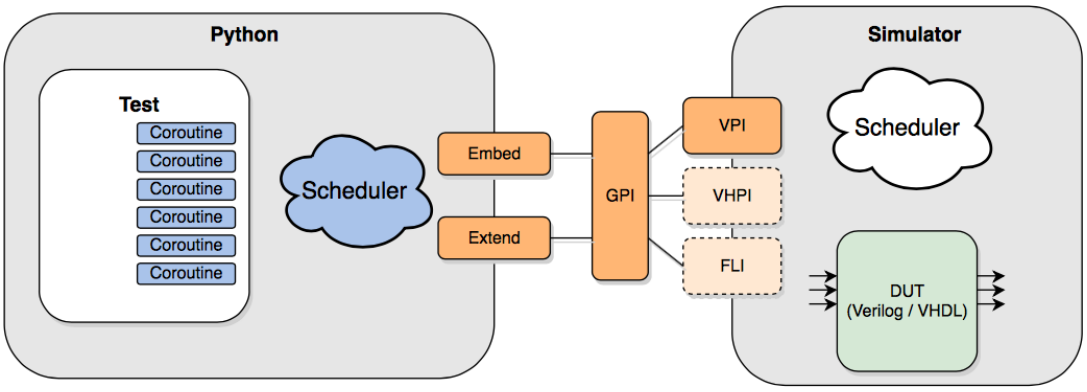


Figure 2.5: The Cocotb overview [49]

The GPI act as *General Purpose Interface*, to make interface independent commands possible at the Python Side. The VPI is used for a Verilog or SystemVerilog design, the VHPI is used for a VHDL design. ModelSim and QuestaSim have its own VHDL foreign interface, the FLI, which is also supported in Cocotb.
It seems that the keywords *Embed* and *Extend* indicate the parallel and sequential possibilities, but these two words are not explained.

Cocotb works with makefiles for specific simulators, such as Icarus Verilog or ModelSim, in which the verilog-sources and the name of the top-entity must be given. When Cocotb initialises, it finds the top-level instantiation in the simulator and creates a handle called *dut*. This handle can be used to access signals inside the design [49].

The usage of the handle *dut* is shown in the following examples. The signal *reset* is accessed inside the design with *dut.reset*. With the *yield* and *fork* commands sequential and parallel Cocotb routines can be executed.

```
@cocotb.coroutine
def reset_dut(reset_n, duration):
    reset_n <= 0
    yield Timer(duration)
    reset_n <= 1
    reset_n._log.debug("Reset complete")
```

Listing 2.20: Reset the DUT [49]

```
@cocotb.test()
def parallel_example(dut):
    reset_n = dut.reset

    # This will call reset_dut sequentially
    # Execution will block until reset_dut has completed
    yield reset_dut(reset_n, 500)
    dut._log.debug("After reset")

    # Call reset_dut in parallel with this coroutine
    reset_thread = cocotb.fork(reset_dut(reset_n, 500))

    yield Timer(250)
    dut._log.debug("During reset (reset_n = %s)" % reset_n.value)

    # Wait for the other thread to complete
    yield reset_thread.join()
    dut._log.debug("After reset")
```

Listing 2.21: A reset test [49]

Cocotb does not put any requirements on the VHDL or Verilog sources; delays can be put in both the HDL and Python. Furthermore, the VHDL and Verilog sources can be loaded in Cocotb without any changes. Cocotb automatically recognizes the available signals and creates handles to make them accessible in Python.
There is however one disadvantage: only one co-simulation can be performed at a time.

As will be shown in chapter 5, the implemented co-simulation between CλaSH and Verilog has the advantages of both MyHDL and Cocotb. CλaSH automatically recognizes the DUT and connects automatically CλaSH signals with Verilog ports. Multiple co-simulations in one CλaSH simulation can be performed using lazy evaluation, which is needed to independently define co-simulation in multiple functions. Furthermore, delay statements are allowed in the Verilog code.

2.3 Verilog & VHDL interfaces

The *Institute of Electrical and Electronics Engineers* (IEEE) is a leading standards development organization and in the area of *Hardware Description Languages* multiple standards are defined. VHDL is standardized in the IEEE 1076 standard, Verilog in the IEEE 1364 standard, and SystemVerilog in the IEEE 1800 standard. SystemVerilog was first seen as extension to the Verilog language, but since 2009 both languages are merged in the IEEE 1800 standard. SystemC, described as System-Level Modelling Language, can also be used as HDL and is standardized in the IEEE 1666 standard.

The history of interfaces to define communication between the (standardized) HDLs and Foreign Languages sees its origin in the 1980s. Verilog was created by Prabhu Goel and Phil Moorby around 1984, and at that time Verilog represented a tremendous productivity improvement for circuit designers. The Verilog PLI was first introduced in 1985 as part of a digital simulator called Verilog-XL, developed by Gateway Design automation; which later merged into Cadence Design Systems [1].

Around the same time, VHDL was developed commissioned by the U.S. Department of Defense (DoD), to document the behaviour of ASICs. In 1987 VHDL was standardized in the IEEE 1076-1987. Because of the increasing success of VHDL, Cadence decided to make Verilog available for open standardization. In 1995 it was standardized in the IEEE 1364-1995 standard [1].

Although Verilog already has foreign support from the beginning, VHDL was seen as a stand-alone language. Recognizing the Verilog PLI success, the IEEE 1067-1993 standard includes the foreign attribute. The foreign interface, VHPI, is included in the IEEE 1067-2008 standard, but at that time VHDL simulators had often defined their own interfaces.

In the 1990s, the Verilog PLI became very popular and succesfull. Mixed language simulators, primarily combining VHDL and Verilog, became common and a more generalized and lucid foreign interface with less overhead was desired. This interface, called the *Direct Programming Interface* (DPI), was defined in the SystemVerilog 3.1 standard, which was introduced in 2003. [19].

In 2001, the Verilog PLI was replaced by the *Verilog Procedural Interface* (VPI). In 2005, this interface was also included in the SystemVerilog standard [1][4].

In the following sections, information about the foreign interfaces will be given.

2.3.1 Verilog Procedural Interface

The *Verilog Procedural Interface* (VPI) gives possibilities to communicate with Verilog simulator(s) and is primarily intended for the C programming language. How to implement the VPI is explicitly defined in the *IEEE 1364* standard. This gives the advantages that every IEEE compliant Verilog simulator will use the same methods. In 1995 the IEEE chose to standardize the PLI 1.0 and 2.0 interfaces what resulted in VPI. PLI is an abbreviation for the *Program Language Interface* and mainly PLI 1.0 is still commonly used due to its widely documented function interface. In the IEEE 1364-2001 standard, the terms PLI 1.0 and PLI 2.0 do not exist, but they are referred as TF (task/function) and ACC (access) [1][4].

The *TF* library can only access the arguments of a system task or function. For simulators this is an advantage, because at compile time it is possible to exactly predict which information the PLI-application will access. For example, with a TF library it is not possible to traverse design hierarchy, analyze design structure, modify delays or access RTL code. In the beginning, the TF library was very small and contained a few C functions. Without any specification, the library evolved over the years and currently it contains more than 110 C functions. This give cause to inconsistency and redundancy. Furthermore, the library has problems with portability to different simulators and operation systems [1][4].

The *ACC* library is an extension of the TF library. In this library it is possible to access structural objects in a simulation data structure. This is done by using routines which can search for the structural objects, but it is limited to only accessing structural (netlist) based designs. The library cannot access all types of objects, like verilog procedures, continuous assignments, and memory arrays.
Performance is a limitation, because of the possible arbitrary access to objects. The simulator cannot predict what will be accessed and can thus not optimize the data structure. The ACC library suffers from many of the same problems as the TF library, like inconsistency and problems with the portability to different simulators [1][4].

The *VPI* library replaces the more than 220 C functions (that are complex, inconsistent and have portability problems) by 37 C functions in the IEEE 1364-2001 standard. The VPI provides full access to the RTL and the behavioral code. The functions are divided in three categories [1][4]:

- Locating objects
- Reading and modifying information about objects
- Utility routines for tasks (e.g. controlling simulation and file I/O)

These functions, as further explained in chapter 3, are used to define co-simulation between Verilog and high-level HDLs, like MyHDL and Cocotb.

2.3.2 VHDL Procedural Interface

The *VHDL Procedural Interface* (VHPI), as defined the IEEE 1076 standard, gives support for foreign communication with VHDL and is based on the *Verilog Procedural Interface*. Compared to VPI, VHPI has less support from vendors and some of them stick to their own interface, like ModelSim & QuestaSim with its *Foreign Language Interface* (FLI).

VHDL is a *Hardware Description Language* (HDL) and largely based on the programming language Ada. Originally, the associated mindset was that the language will mainly be used stand-alone, and would be self-contained. In the VHDL 93 revision the need for including non-VHDL code was recognized and resulted in the *Foreign* attribute.

```
package P is
  function F return INTEGER;
  attribute FOREIGN of F: function is "implementation-dependent info";
end package P;
```

Listing 2.22: The declaration of a foreign function subprogram [9]

The *Foreign* attribute was very limited in practice. This attribute gave possibilities for including non-VHDL sub-programs, but there was no support for traversing a VHDL hierarchy or synchronizing to simulation events. Looking at the capabilities of the Verilog PLI, vendors often provide mechanisms to communicate with foreign code. Most of these foreign functionalities are vendor specific and the capabilities vary across products. One of these interfaces is the *Foreign Language Interface*, as will be described in the next section.

In 2007, as an amendment to the VHDL 2002 standard, the *VHDL Procedural Interface* was introduced. The VHPI is an application-programming interface to VHDL simulators, which provides access to a VHDL model during its analysis, elaboration, and execution [10].

The VHPI consist of two aspects. The first aspect is an *information model* that represent the topology and the state of a VHDL model. This model is expressed in an object-orientated manner as a set of classes with relationships between them. A class is a data type that have data properties and sub-program operations.
The second aspect is a number of functions that can operate on the *information model* to access or affect the state of the VHDL model, and to control the VHDL simulator [10].

An *information model* is also referred as *Static VHDL Design Data*, which contains the behavioural and structural parts of the elaborated model. A VHPI application can traverse the hierarchy of this model, fetch the properties of VHDL objects, and navigate between these objects [10][11].

A VHPI application can access values of *Dynamic VHDL Objects*, such as signals and variables. These values can be fetched and modified using different formats as described in the IEEE standard [10][11].

Callbacks are used to create interaction with the simulation. A callback can be defined object related (e.g. executed when a value changes) or time related (connected to a simulation phase or executed after a number of simulation steps) [11].

The VHPI also provides *foreign* mechanisms to execute C code in a VHDL design. A VHPI application must be registered during simulation startup in order to work properly [11]. The following examples are copied from [11], to show a small VHPI example.

```
PLI_VOID startup() {
  vhpiForeignDataT foreignData = {vhpiProcF, "test.dll", "test_init",
                                  NULL, register_cb};
  vhpi_register_foreignf(&foreignData);
}

PLI_VOID (*vhpi_startup_routines[])() = { startup, 0L };
```

Listing 2.23: VHPI callback registration [11]

The function-pointer array *vphi_startup_routines* is the starting point of a VHPI application. The functions in this array will be executed during simulation startup and can be used to register callbacks or foreign functions.
The function *startup* registers a *foreign procedure*, which is connected to the function *register_cb* and can be executed in the VHDL code, as shown in Listing 2.23.

```
PLI_VOID register_cb ( const struct vhpiCbDataS* cb_p ) {
  vhpiCbDataT       cbDataAction;
  vhpiValueT        *Value;
  vhpiTimeT         *Time;

  vhpiHandleT hnd       = vhpi_handle_by_name("top.memory_var", NULL);
  vhpiHandleT hnditr    = vhpi_iterator(vhpiIndexedNames, hnd);

  Value = (vhpiValueT*) malloc ( sizeof(vhpiValueT) );
  ...
  Value->format         = vhpiStrVal;
  cbDataAction.value    = Value;
  cbDataAction.cb_rtn   = ValueChangeEvent;
  cbDataAction.reason   = vhpiCbValueChange;

  while ( vhpiHandleT hndByIdx = vhpi_scan (hnditr) ) {
     cbDataAction.obj   = hndByIdx;
     vhpi_register_cb (&cbDataAction, vhpiReturnCb);
  }
}
```

Listing 2.24: The *register_cb* function [11]

The function *register_cb*, as shown in Listing 2.24, iterates through the VHDL hierarchy (with the function *vhpi_handle_by_name*) to get a handle to the array *memory_var*. The *vhpiCbValueChange* callback is registered to every element of this array, with the format defined as *vphiStrVal*. When an element of *memory_var* changes, the function *ValueChangeEvent* will be executed, with as argument the changed value (defined as string).

The VHDL code, as shown in Listing 2.25, defines the C function *register_cb* as a foreign procedure with the name *callback_event*. This foreign procedure is executed at the beginning of the process *proc1*. Within the function *register_cb*, *vhpiCbValueChange* callbacks are connected to the array *memory_var*.
The other VHDL statements (in the process *proc1*) modify elements of the *memory_var* array after certain delays. Each time that an element is modified, the C function *ValueChangeEvent* will be executed.

```
library ieee;

entity top is
end;

architecture top of top is
  procedure callback_event;
     attribute foreign of callback_event: procedure is "VHPI test;
         test_init";

  type MEM is array ( NATURAL range <> ) of BIT_VECTOR( 7 downto  0 );
  signal memory_var : MEM( 3 downto 0 );

begin
  proc1: process
  begin
     callback_event;

     wait for 1 ps;     memory_var(0) <= "00111000";
     wait for 1 ps;     memory_var(2) <= "10010010";
     wait for 1 ps;     memory_var(3) <= "01101000";

     wait;
  end process;
end;
```

Listing 2.25: The VHDL code which is connected to the VPHI application [11]

2.3.3 Other Foreign Interfaces

The *Foreign Language Interface* (FLI) is an interface to provide procedural access to information within ModelSim and QuestaSim. The FLI consist of C functions, which can traverse the hierarchy of an HDL design, read and drive *VHDL* signals, and control (to some extent) the simulation [17][18].

The FLI has a comparable structure to the VPI and VHPI. When the simulator starts, it will load the libraries and initialize the functions which are connected through the FLI. The initialization needs an entry point in the foreign C model. This entry point can allocate memory, register callbacks to the simulator or signal events, and define C procedures which can be executed within the VHDL code [17].

Although the other mentioned interfaces are standardized, the FLI is not standardized and can only be used with ModelSim (with an SE license) or QuestaSim. Having a mixed-language simulation license, it is possible to use the FLI for Verilog, but then the code must be instantiated inside a VHDL top level design [17].

The *Direct Programming Interface* (DPI) is available for SystemVerilog. The DPI is standardized in the IEEE-1800 standard and provides mainly functionality for integrating SystemVerilog code with C code. Values, with a compatible type, can be passed directly between the two languages. Furthermore, C functions can call SystemVerilog functions & tasks and SystemVerilog can make concurrent calls to C functions.

With some simulators, such as the *Synopsys VCS simulator*, the C sources are compiled together with the SystemVerilog source-files. The SystemVerilog code is unaware that it is calling C code, and the C functions are unaware that they are called from SystemVerilog [19][20].

```
import "DPI" function real sin(real in);

always @(posedge clock) begin
    slope <= sin(angle);
end
```

Listing 2.26: The imported C function *sin* in SystemVerilog

The possibilities of the DPI are too limited to define co-simulation. With the DPI it is not possible to access the simulation data structure or to synchronize to simulation activity.

To give support for co-simulation it is desirable to traverse the design hierarchy and to automatically connect the two HDLs. Furthermore, callbacks are needed for synchronization and data exchange. The VPI and VHPI would be appropriate choices, because they both support these features and are standardized.

3 VPI

In section 2.3, an introduction is given to the foreign interfaces of the traditional *Hardware Description Languages*. The *Verilog Procedural Interface* will be further explained in this chapter. The VPI is one of the most popular foreign interfaces and, in contrast to the VHPI and FLI, literature is widely available. The VHPI is implemented in a comparable way, as will be seen in section 6.2, and the assumption is made that this research can be re-used to define co-simulation with VHDL.
Besides the IEEE standards, the books [1] and [2] will be used as main references for this chapter.

The IEEE Verilog standard encapsulates the Verilog foreign interfaces. In the IEEE 1364-2001 standard two older versions of the Verilog PLI standard were included, namely *PLI 1.0* (*TF/ACC*) and *PLI 2.0*. In the IEEE 1364-2005 standard, the two older versions are officially deprecated in favor of the newer VPI, as stated in the following quote:

"*IEEE Std 1364-2005 deprecates the Verilog PLI TF and ACC routines that were contained in previous versions of this standard. These routines were described in Clause 21 through Clause 25, Annex E, and Annex F. The text of these clauses and annexes have been removed from this version of the standard. The text of these deprecated clauses and annexes can be found in IEEE Std 1364-2001*" [4][5].

Some simulators, like ModelSim, follow the latest standards and using deprecated functions will result in, for example, null-pointers, as shown in the following quote:

"*# ** Warning: (vsim-8668) tf_nodeinfo has been deprecated by IEEE. Although still partially supported, memoryval_p will always be set to a a null pointer*" [16].

In the IEEE 1800-2005 standard, the VPI is extended to be supported with SystemVerilog; but the IEEE 1800-2005 standard referred to, and relied on, the IEEE 1364-2005 standard. In the IEEE 1800-2009 standard, both the IEEE 1364-2005 and the IEEE 1800-2005 standards were merged. Integration with Verilog-AMS was supported to ensure interoperability with other languages, such as VHDL and SystemC. The latest standard is the IEEE 1800-2012 standard, which mainly corrects errors and clarifies aspects of the language definition in IEEE 1800-2009 [6][7][8].

A 'VPI application' is a user-defined application which can be executed by a Verilog simulator[1]. The Verilog language must be used as the top-level design at the simulator side. Mixed language design provides possibilities for using VHDL with the same interface, but this lays outside the scope of this research.

The Verilog simulator will load and execute the VPI application. The simulator gets the pre-compiled VPI application as a start-up argument, in which it is possible to register functions and callbacks. The VPI application can interact with the simulation using these functions and callbacks.

The starting point for the VPI application is a special array, called *vlog_startup_routines*, which contains the names of the register functions. These register functions will be executed by the simulator before simulation time 0. Within a register function, *callbacks* and *system tasks/functions* can be registered.

A callback is often used to synchronize with simulation or value events. For example, when the simulation starts, the simulator will generate a simulation event. With the *cbStartOfSimulation* callback this event can be used in a VPI application. In section 3.2 these types of routines will be explained.

A *system task/function* is a C function which can be used within the Verilog code. To register this C function a *Calltf* and a *Compiletf* routine must be provided. In section 3.1 the registration and usage of C functions within Verilog code will be explained.

With the routine *vpi_control*, certain aspect of a Verilog-simulation can be controlled. Four operation constants are defined in the IEEE 1364 standard, as shown in Table 3.1, which can be provided to the routine *vpi_control*. Vendors can add additional flags specific for their simulator.

Flag	Description
vpiStop	execute the Verilog system task $stop
vpiFinish	execute the Verilog system task $finish
vpiReset	execute the Verilog system task $reset
vpiSetInteractiveScope	change a simulator's interactive debug scope

Table 3.1: VPI flags to control a simulation

[1]The focus will be on the Verilog-language, but the VPI can also be used with the SystemVerilog language

3.1 Compiletf & Calltf

The IEEE 1364 Verilog standard defines a number of system tasks and system functions that are built into all Verilog simulators. Examples of system tasks are $display, $stop, and $finish. The task $display is often used as print statement. The tasks $stop and $finish are used to control the simulation. The $stop command suspends the simulation and puts the simulator in interactive mode. The $finish command exits the simulation. Examples of standard system functions are $time and $random, to get the current time and to generate random numbers.
Simulation vendors can add proprietary system tasks and functions, which are specific to a simulator product. Besides the standard and vendor tasks/functions, the VPI provides ways to specify additional user-defined system tasks or functions, which are connected to corresponding, user-defined, C functions.

The names of the user-defined tasks/functions must begin with a dollar sign ($) and only the following characters are legal in the Verilog names: $a-z$ $A-Z$ $1-9$ _ $

The IEEE standard does allow overriding built-in system tasks and functions. For example, if a special random number generator is needed, it could be assigned to $random to override this system function.

System tasks can only be used as procedural programming statements and are thus only allowed in a Verilog *initial* procedure, an *always* procedure, a Verilog task, or a Verilog HDL function. System functions are used as expressions and can be called anywhere a logic value may be used. Furthermore, both the system task and function may have any number of arguments, including none.

A user-defined system task/function must be registered with the *vpi_register_systf* routine. The struct *s_vpi_systf_data* must be used to specify the system task/function, as shown in Table 3.2.

Name	Description
type	defines if the application is a system task or function
sysfunctype	defines the return type of a system function
tfname	specifies the name of the system task/function
calltf	specifies a pointer to the *calltf* routine
compiletf	specifies a pointer to the *compiletf* routine
sizetf	specifies a pointer to the *sizetf* routine
user_data	specifies a pointer to application-allocated information

Table 3.2: The components of the struct *s_vpi_systf_data*

Instead of defining one corresponding C function, the user can define up to three C functions used for a system task/function. These three C functions are named: *calltf*, *compiletf*, and *sizetf*. The *calltf* is the actual C function which is connected to the system task/function and will be executed at simulation run-time.

The *compiletf* and *sizetf* are optional functions, which are only executed once before the simulation starts. The main purpose of these two functions is to verify that a system task/function is being used correctly. The intent of the *compiletf* routine is to verify the correctness of the arguments of all the instances of a specific system task/function.
The *sizetf* routine is only used with system functions of which their return values are of the type *vpiSizedFunc* or *vpiSizedSignedFunc*. These types indicates that the function returns scalar or vector values[2]. These scalar or vector values have a user-specified number of bits and the simulator may need to know how many bits the function will return to correctly compile the Verilog statement.

```
always @(posedge clock)
  begin
    result <= $pow(x,y);
```

Listing 3.1: The user-defined pow-function

In Listing 3.1 the user-defined *$pow* function is used in a Verilog *always* procedure. In Appendix B the VPI routines, needed for this function, are shown.

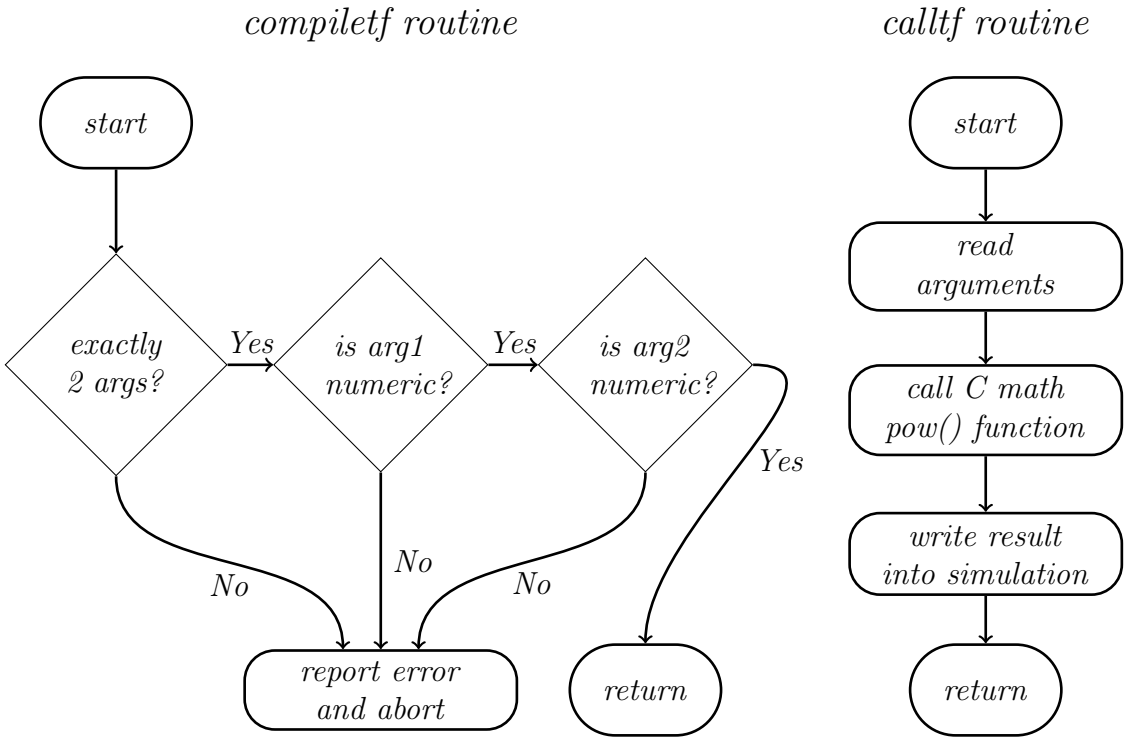


Figure 3.1: The schematic overview of the *$pow*-function [1]

[2]Functions can also return integer or real number values

3.2 Simulation Events & Callbacks

Besides using user-defined system tasks/functions, callbacks can be used to synchronize with the simulation or to indicate value changes. Furthermore, it is possible to use callbacks when the simulator encounters a user-defined system task/function.

A callback must be registered with the *vpi_register_cb* routine. The struct *s_cb_data* must be used to specify the callback, as shown in Table 3.3.

Name	Description
reason	indicates the callback reason
cb_rtn	specifies the name of the callback routine
obj	specifies the handle to the triggered object, if needed
time	indicates when the callback should occur, if needed
value	a pointer to an *s_vpi_value* structure, if needed
index	specifies the index of the changed memory word, if needed
user_data	specifies a pointer to application-allocated information

Table 3.3: The components of the struct *s_cb_data*

The *vpi_register_cb* routine returns a handle to the registered callback. The callbacks, which have not yet been transpired, are called scheduled callbacks. Using the handle, a scheduled callback can be removed at any time with the *vpi_remove_cb* routine. The handle will no longer be valid after the callback is removed. The removal request will be ignored when a transpired callback is tried to be removed. Even when a callback has expired, the handle must be freed with the *vpi_free_object* routine to avoid memory leaks.

Within the IEEE standard, a distinction is made between one-time (simulation action) callbacks and repeating (simulation action) callbacks. Examples of one-time callbacks are *cbEndOfCompile*, *cbStartOfSimulation*, and *cbEndOfSimulation*. These callbacks are directly related to the simulation, like the start of the simulation, and will only be executed once.
Repeating callbacks are for example *cbError*, *cbPLIError*, *cbTchkViolation*, and *cbSignal*. The first two callbacks are used to indicates a run-time error in the Verilog HDL or during the executing of a VPI application. The *cbTchkViolation* callback is used to indicate if a Verilog HDL timing check violation occurred. The *cbSignal* callback is used to respond when an operating system signal occurred.

Besides simulation action callbacks, simulation time-related callbacks can be used, which are all one-time and will not be repeated. The following callbacks are available, as shown in Table 3.4.

Name	Description
cbReadWriteSynch	callback after execution of all known events
cbReadOnlySynch	callback after execution of all events
cbNextSimTime	callback in the next simulation time step
cbStartOfSimTime	callback at a specific simulation time
cbAfterDelay	callback after a specified amount of time

Table 3.4: The VPI simulation time-related callbacks

The last three callbacks, *cbNextSimTime*, *cbStartOfSimTime*, and *cbAfterDelay*, are called before execution of any simulation events of the specified time step. The moment when the first two callbacks, *cbReadWriteSynch* and *cbReadOnlySynch*, are called (within a certain time step) is not known in advance and is simulator dependent.

The IEEE 1364 Verilog standard contains a generalized description of an event scheduler algorithm for Verilog simulators [4][5]. Within this description, four distinct regions of events, called slots, are defined within a simulation step. Within each slot, certain types of event are scheduled to be executed. A simplified overview of the slots is shown below [1].

Current Simulation Time:

- **Slot 1: Active Events**
  - Evaluate nonblocking assignments.
  - Evaluate and assign blocking assignments.
  - Evaluate and assign continuous assignments.
  - Evaluate the changes on primitive inputs & schedule changes to outputs.
  - Print outputs from scheduled $display and $write tasks.
  - Execute VPI calltf routines.
- **Slot 2: Nonblocking Assignment Update Events**
  - Assign nonblocking assignments.
- **Slot 3: Read-write Synchronization**
  - Call registered *cbReadWriteSynch* callbacks.
- **Slot 4: Read-only Synchronization**
  - Call registered *cbReadOnlySynch* callbacks.

The IEEE standard gives freedom on how to implement the internal event scheduling algorithm. This freedom is, for example, needed to create competitive algorithms, but sometimes it is also required for the behaviour of different types of hardware [1].

The events in *Slot 1* may be interleaved in any order. Furthermore, *Slot 2* and *Slot 3* can be reversed. For simulation results the affects of the event ordering should not matter, but this ambiguity may effect the results of a *calltf* routine, as shown in Listing 3.2.

```
always @(a or b)
  begin
    c = a + b;          /* blocking assignment */
    d <= a - b;         /* non-blocking assignment */
    $my_func(c, d, e);  /* calltf routine */
  end

assign e = ~c;          /* continuous assignment */
```

Listing 3.2: A possible uncertain event interleaving

The order in which the concurrent continuous assignment and the calltf routine are executed is simulator depended, and thus the argument '*e*' is ambiguous. A solution, for this ordering problem, can be to schedule a *cbReadWriteSynch* callback in the calltf routine *$my_func*. The *cbReadWriteSynch* callback will be scheduled after all active events have been processed (*Slot 3*). But by using a *cbReadWriteSynch* callback, the value of '*d*' will become ambiguous, because a simulator can reverse *Slot 2* and *Slot 3*.

The simulator will first execute all scheduled active events in *Slot 1*, after which it will proceed to *Slot 2*. The events in *Slot 2* will be moved to the active event lists and then these new events will be processed. While processing the events, additional active events can be scheduled. When all active events of *Slot 1* and *Slot 2* have been processed, the simulator will move to *Slot 3*. The new active events will be processed again and additional event can be scheduled. The loop through the three slots will continue until all these slots are completely empty, after which the simulation will proceed to *Slot 4*.
It is important to notice that is possible that *Slot 4* will never be reached, i.e. by defining zero-delay infinite loops in the Verilog code.

The definition of *Slot 4*, that it will be executed after all active events are processed, does affect the scheduling possibilities of the *cbReadOnlySynch* callback. In this callback only event for future simulation steps, and thus not for the current simulation step, can be scheduled. In the other callbacks, events for the current simulation step and for future simulation steps can be scheduled.
Slot 4 can thus be seen as the true end of a simulation step.

In the past there have been discussions about the interleaving and reordering ambiguity. A proposal was to add two additional callbacks: *cbBeforeNBA* and *cbAfterNBA*. The callback *cbBeforeNBA* should be executed after all scheduled active events have been processed, but before the updates of any scheduled nonblocking assignments. The callback *cbAfterNBA* would occur after all scheduled nonblocking assignments have been processed. Simulators can implement these two additional callbacks, because the Verilog standard allows to add simulator-specific callback-reasons.

The last major category of callback reasons are the simulation events, such as (logic) value changes and the execution of procedural statements. Simulation event-related callbacks can occur more than once and are automatically defined as repeated callbacks. Table 3.5 lists the available VPI simulation event-related callbacks.

Name	Description
cbValueChange	callback after a logic or strength value changes
cbStmt	callback before execution of a procedural statement
cbAssign	callback after execution of a procedural assign
cbDeassign	callback after execution of a procedural de-assign
cbForce	callback after a *force* has occurred
cbRelease	callback after a *release* has occurred
cbDisable	callback after execution of a procedural disable

Table 3.5: The VPI simulation event-related callbacks

In some simulators, the *cbValueChange* callback is used to generate *Value Change Dump* (VCD) files. VCD is an ASCII-based format for dumpfiles used in various (Verilog) simulators. The VCD format is described in the IEEE 1364-1995 standard and an extended VCD format is defined in the IEEE 1364-2001 standard. The original format gives support for logging the 4 logic values[3]. The extended format also gave support for the logging of strength values and the directionality of signals.

[3]The 4 logic values are described in section 3.4.

3.3 Traversing hierarchy

The Verilog hierarchy is defined with object diagrams and relationships between several diagrams. A Verilog design can be traversed starting from the top-modules to anywhere in the design. Table 3.6 shows the specifications as defined in the module-object-diagram.

Name	Description
vpiType	returns *vpiModule*
vpiTopModule	bool if the module is a top-level module
vpiCellInstance	bool if the module is tagged as a cell
vpiArray	bool if the module is part of an instance array
vpiProtected	bool if the module source is protected
vpiTimeUnit	the module time unit
vpiTimePrecision	the module time precision
vpiDefNetType	the default net type
vpiUnconnDrive	the unconnected port drive
vpiDefDelayMode	the delay mode of the module
vpiName	the instance name of the module
vpiFullName	the full hierarchical path name of the module
vpiDefName	the definition name of the module
vpiFile	the file name containing the module instance
vpiLineNo	the file line number containing the module instance
vpiDefFile	the file name containing the module definition
vpiDefLineNo	the file line number containing the module definition
vpiConfig	the library/cell names of the corresponding config statement
vpiLibrary	the name of the corresponding configuration library
vpiCell	the name of the corresponding configuration cell

Table 3.6: The VPI module-object properties

The module time unit and precision can be retrieved with the *vpiTimeUnit* and *vpiTimePrecision*. The simulation time unit and precision can be accessed by using a null-pointer as module-handle, . These time scale factors are represented as the magnitude of 1 second, which is the exponent of 1 second times 10^n (values from -15 untill 2 are allowed). The value '-15' will indicate 1 femtosecond and the value '2' will denote 100 seconds.

Besides the module specific constants, as shown in Table 3.6, other constants like *vpiModule*, *vpiPort*, *vpiNet*, *vpiReg*, *vpiVariables*, *vpiParameter*, *vpiPrimitive*, and *vpiModPath* can be used to access the internals of a module.

```
void print_module_names (vpiHandle handle)
{
  vpiHandle mod_itr, mod_handle;

  mod_itr = vpi_iterate(vpiModule, handle);
  if (mod_itr != NULL)
  {
    while ( (mod_handle = vpi_scan(mod_itr)) != NULL)
    {
      vpi_printf("%s\n", vpi_get_str(vpiFullName, mod_handle));

      print_module_names(mod_handle);
    }
  }
}
```

Listing 3.3: Iterating through the VPI modules

As shown in Listing 3.3, the routines *vpi_iterate* and *vpi_scan* can be used to traverse through a Verilog hierarchy. An iterator for the top-modules in the Verilog design will be given when a null-pointer is used as handle.

Instead of using the *vpiModule* constant, the *vpiModPath* can be used to iterate through all the available *module paths*. Within a *module path* it is possible to iterate through all the available *path terms* to obtain handles for the path input terminals, output terminals, and data terminals.

Handlers to the ports of a module can be obtained with the constant *vpiPort*. To automatically recognize the modules in a design with the available ports, the combination of *vpiModule* and *vpiPort* is very powerful. Within a port, specifications like name, direction, and bit-size can be retrieved and used to automatically map certain signals within a co-simulation design. Furthermore, using the *vpiNet*, *vpiReg*, and *vpiVariable* objects, the current values within a module can be retrieved, as will be explained in section 3.4.

```
void print_port_specs (vpiHandle mod_handle)
{
  vpiHandle port_itr, port_handle;

  port_itr = vpi_iterate(vpiPort, mod_handle);
  if (port_itr != NULL)
  {
    while ( (port_handle = vpi_scan(port_itr)) != NULL)
    {
      vpi_printf("%s - %d - %d\n", vpi_get_str(vpiName, port_handle),
        vpi_get(vpiDirection, port_handle), vpi_get(vpiSize, port_handle));
    }
  }
}
```

Listing 3.4: Iterating through the VPI ports

Within a module-port, besides using properties like *vpiName*, *vpiDirection*, and *vpiSize*, the properties *vpiLowConn* and *vpiHighConn* can be used to iterate through the connections of a port. The *lowconn* is the signal *inside* the module that is connected to the port. The *highconn* is the signal *outside* the module that is connected to the port. Both the *lowconn* and *highconn* are defined as an *expression*, because different types can be connected to a port. Within a module, the allowed types are *nets*, *regs*, and *variables*. An *expression* can be scalar or a vector. Furthermore, the *lowconn* can have different port-directions, referred as a mixed IO port.

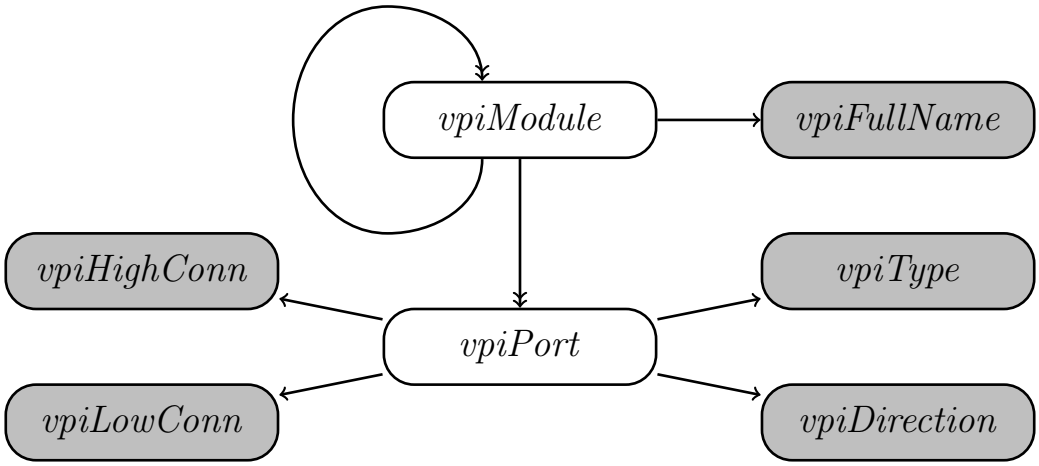


Figure 3.2: A part of the VPI object diagram

3.4 Reading & Modifying Values

The values of objects, with a value property in the VPI object diagrams, can be read or modified with VPI routines. Examples of these objects are: *net*, *reg*, *integer*, and *time* variable data types. Constants, like *parameter*, *specparam*, and *literals* are defined as read only.
To access the value of an object, a handle for the object must be obtained. The routines as shown in section 3.3 can be used to the obtain a handle.

The Verilog standard supports 4 logic values: '0', '1', 'Z', and 'X'. Furthermore, there are ambiguous logic and strength values, like 'L' and 'H', which are only supported to a certain extent. The VPI routines provides several ways to automatically translate values between Verilog and C. The following C types are supported:

- A 32-bit C integer → a single integer is used to represent the Verilog value. The logic values 'Z' and 'X' are ignored.
- A C double → the real data type in Verilog is defined as double-precision floating point, which can directly be converted to C doubles. This type can also be used for scalar and vector values.
- A C string → Verilog scalar and vector logic values can be converted to a C character string. The value will be converted to the characters '0', '1', 'Z', and 'X'.
- A C constant → Verilog scalar values can be converted to a C integer constant. The following constants are supported: *vpi0*, *vpi1*, *vpiZ*, *vpiX*, *vpiZ*, *vpiH*, and *vpiL*.
- A C aval/bval structure → Verilog scalar and vector logic values can be converted to a C structure, which encodes each bit of a Verilog 4-state value to a pair of bits in C. An array of aval/bval pairs can be used to encode Verilog vectors of any size.
- A C strength structure → Verilog scalar and vector logic values can be converted to a C structure, in which the logic values are represented as the *C constants*. The strength levels are represented as a pair of 32-bit C integers.

The format used to translate values between Verilog and C can have an impact on the run-time performance of a VPI application. For Verilog scalars and vectors, using the *aval/bval pair* will result in the fastest run-time performance. The least efficient method is using the *C string*.

The *aval* value denotes a '0' or a '1' when the *bval* is a '0'. If the *bval* is a '1', the *aval* denotes a 'Z' (0) or a 'X' (1). An array of aval/bval pairs is denoted as the *vpiVectorVal* format.

The Verilog language does support any numbering convention. Any natural number can be used to indicate the starting and ending bit of a Verilog vector.

```
reg [39:0] data1;    /* LSB is bit 0  */
reg [0:39] data2;    /* LSB is bit 39 */
reg [40:1] data3;    /* LSB is bit 1  */
```

Listing 3.5: Valid vector declarations [1]

However, the Verilog numbering convention does not effect the *vpiVectorVal* format. The *Least Significant Bit* (LSB) of the Verilog vector will always be the first bit in the aval/bval array and the *Most Significant Bit* (MSB) will always be the last bit in the array, as shown in Figure 3.3.

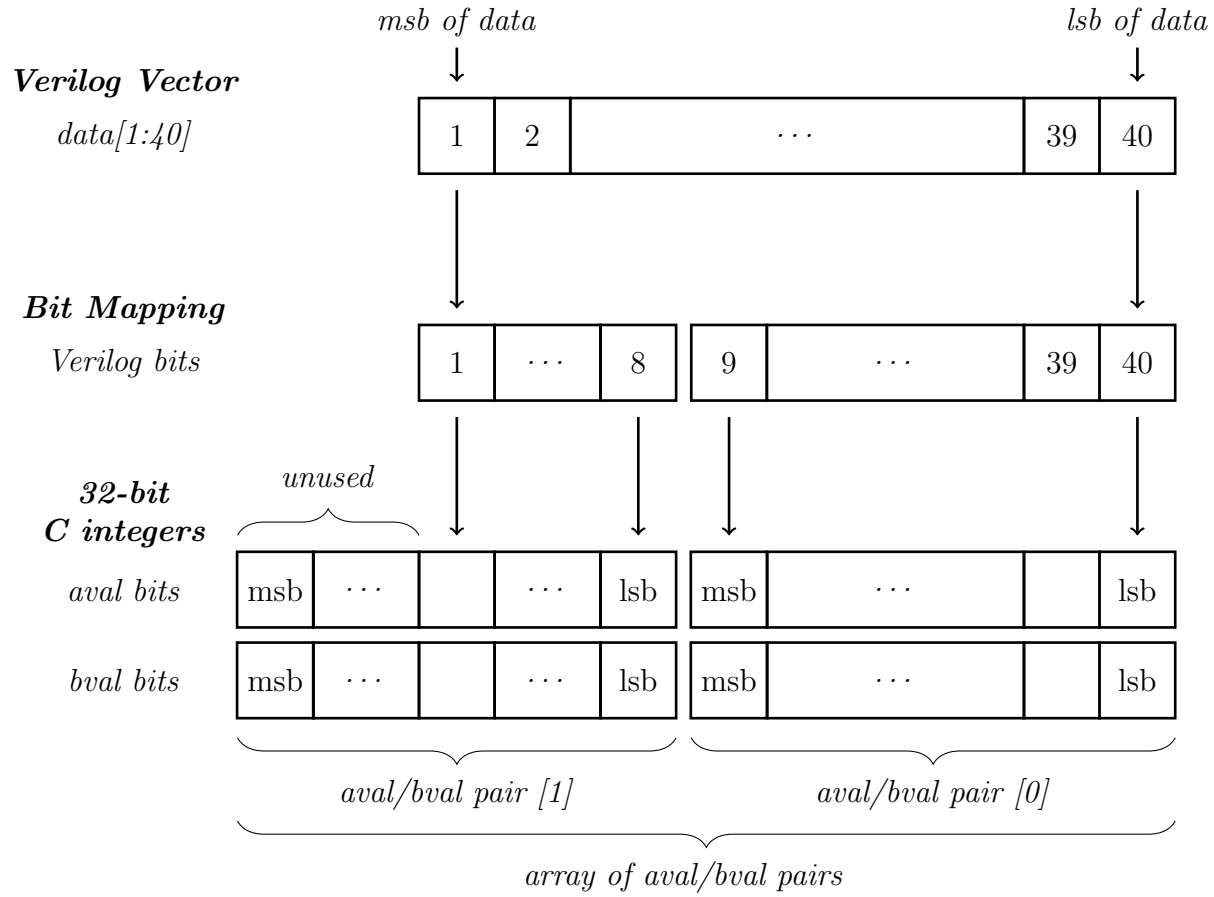


Figure 3.3: The conversion from a Verilog vector to a *vpiVectorVal* [1]

The value of an object can be retrieved with the *vpi_get_value* routine. This function has two arguments; the first one is the *handle* to the object and the second one is a pointer to the struct *s_vpi_value*, in which the format has to be set. The format *vpiVectorVal* is denoted as an array of the struct *s_vpi_vecval*, which contains two integers: *aval* and *bval*. The array of *s_vpi_vecval* structures is allocated by the simulator and is only guaranteed to be valid untill the next call to *vpi_get_value*.
The size of the *vpiVectorVal* can be determined with the VPI routine *vpi_get* and the *vpiSize* constant.

```
vector_size = vpi_get(vpiSize, vector_h);

vector_val.format = vpiVectorVal;
vpi_get_value(vector_h, &vector_val);
```

Listing 3.6: The VPI routines *vpi_get* & *vpi_get_value*

A value can be written into a Verilog simulation with the VPI *vpi_put_value* routine. In this case, the VPI application must allocate and maintain all needed storage elements. The *vpi_put_value* routine has four arguments, from which the first two are the same as the *vpi_get_value* routine. The third argument is a pointer to the struct *s_vpi_time*, in which a propagation delay value can be set. With a delay value of zero, the value will be written in the current simulation step. With a value larger than zero, the value will be written in future simulation steps. The fourth argument is a flag denoting the propagation delay. Examples of this flag are *vpiNoDelay* (using no delay), *vpiInertialDelay* (using the simulator's event scheduling), and *vpiPureTransportDelay* (using a transport delay).

```
vpi_time_s.type = vpiSimTime; /* relative to simulation time */
vpi_time_s.high = 0;          /* high & low are used together ... */
vpi_time_s.low  = 0;          /* ... as a 64 bit integer */

vector_val.format = vpiVectorVal;
vector.value.vector = ...
vpi_put_value(vector_h, &vector_val, &vpi_time_s ,vpiInertialDelay);
```

Listing 3.7: The VPI routine *vpi_put_value*

The *vpi_put_value* routine returns a *scheduled event handle*. With the flag *vpiScheduled* the simulator will indicate if the event is still scheduled or has already transpired. A scheduled event can be cancelled using the flag *vpiCancelFlag*.

```
event_h = vpi_put_value(...

/* check if event still scheduled */
if (vpi_get(vpiScheduled, event_h))
    vpi_put_value(event_h, NULL, NULL, vpiCancelFlag); /* cancel event */
```

Listing 3.8: The scheduled event

4 Implementation

In this chapter, the implementation of the co-simulation between CλaSH and '*Icarus Verilog*' will be discussed. Icarus Verilog is an open-source verilog simulator, which supports a large part of the VPI-standard. The implementation can be checked with other simulators in a later stage. As will be shown in chapter 5, the implementation of some VPI routines differ by using other simulators.

4.1 Overview

The Verilog module(s) with the appropriate input and output types have to be defined at user-level. The module(s) can be defined directly as Verilog code and/or as existing Verilog files. This information will be, without any parsing, forwarded to the Verilog simulator. Besides the definition of the module(s), the user has to define the name of the top-entity and to specify which Verilog simulator will be used for co-simulation.

As explained in section 3.4, the *vpiVectorVal* is the most efficient type for data exchange. For a synchronous sequential circuit, data exchange will occur every clock-cycle; making it an array of *vpiVectorVal*. Within CλaSH, the logic values 'X' and 'Z' will be ignored and the appropriate type will be the so called '*SignalStream*'.

$$type\ SignalStream\ =\ [[Int32]]$$

The same type can also be used for a combinational circuit, and then the outer list will have a length of 1. Before and after the co-simulation, every input and output value have to be converted from and to the user defined types, as will be explained in section 4.2.

The co-simulation uses the *Foreign Function Interface* (FFI). By using the FFI, C functions can be called in CλaSH. The C code will execute the operating system depended tasks, like starting the Verilog simulator and making the communication between the two processes possible. In subsection 4.2.3 this implementation will be explained in more detail.

The VPI implementation, which will be executed by the Verilog simulator, modifies the signals in the Verilog simulator and requests the simulator to schedule events (callbacks), as will be explained in section 4.3.

The implementation can be visualized, as shown in Figure 4.1.

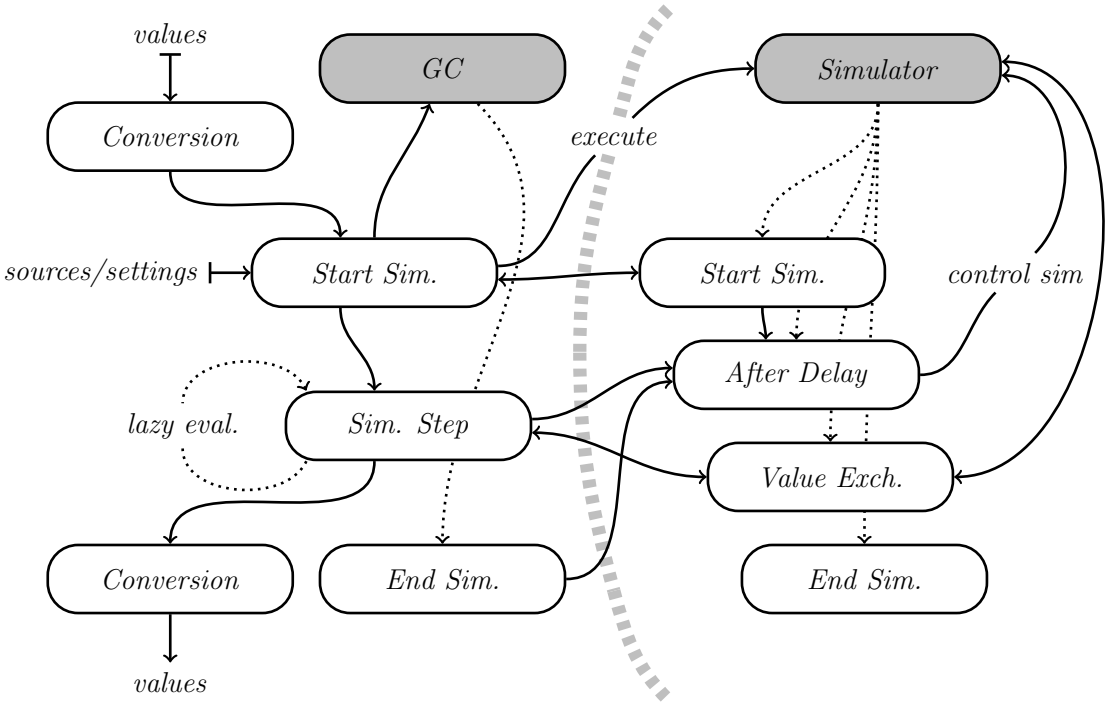


Figure 4.1: The schematic overview of the co-simulation implementation

The lightgray dashed line, which crosses the label '*execute*', can be seen as a boundary between the CλaSH and VPI implementation. At the CλaSH side, the values will first be converted to *SignalStreams*, after which the co-simulation is started.
The *Start Sim* function starts the Verilog simulator with the given Verilog modules. The Verilog simulator will compile the Verilog code and schedule the VPI *Start Sim* function. The VPI implementation will iterate through the top-module to search for available input and output ports. This information will be shared with the CλaSH implementation to check the correctness of the input and output types.

The higher-order Haskell function *mapAccumL* is used to perform the simulation steps. With the *mapAccumL*, the *Sim Step* function will be executed using lazy evaluation. Lazy evaluation is ideal for feedback loops, but has a disadvantage for *lazy IO*. With *lazy IO*, the moment to release the acquired resources is unknown. This problem of *lazy IO* is solved by using a *finalizer*, which is connected to the Haskell's *Garbage Collector* (GC). The finalizer will execute the *End Sim* function, which will finish the Verilog simulator and deallocate the allocated memory.

The VPI function *After Delay* requests the simulator to schedule new callbacks after a certain delay. The *Value Exch* function, defined with these callbacks, will modify signals inside the simulator. Furthermore, the *End Sim* function will free allocated memory.

Finally, the output values will be converted back to the user-defined types.

The example, as visible in Listing 4.1, shows how the co-simulation between CλaSH and Verilog is defined at user-level. The definition of the Verilog module *mult* is defined as a *QuasiQuoter*. A *QuasiQuoter* makes it possible to define custom and domain-specific syntax, in this case the Verilog syntax.

```
verilog_mult :: t ~ Signed 100 => Signal t -> Signal t -> Signal t
verilog_mult         = coSim source Icarus "mult"
    where
        source       = [verilog| module mult (Out, Left, Right);

                                 parameter data_width = 100;

                                 input  signed [0:data_width-1] Left;
                                 input  signed [0:data_width-1] Right;
                                 output signed [0:data_width-1] Out;

                                 assign Out = Left * Right;

                           endmodule |]
```

Listing 4.1: A co-simulation example between CλaSH and Verilog

The function *coSim* act as a *polyvariadic* function, meaning that the number and the types of the arguments are not fixed. However, the first three arguments are fixed.
The first argument is the source, in this case one Verilog module. As second argument, a Verilog simulator is defined, which will compile and simulate the Verilog code. The name of the top-module must be given as third argument, because no Verilog parser is defined in CλaSH.

Type definitions are needed to make the conversion to and from a *SignalStream*. The two inputs and one output are defined with the type *Signal t*, with '*t*' defined as *Signed 100*. The types has to match the input and output ports in the Verilog module. The verilog ports are iterated from left to right, giving *Left* as first input port and *Right* as second input port. The output port *Out* is connected to the output value.

```
λ> let a = fromList [1..]
λ> sampleN 13 $ verilog_mult a a
[1,4,9,16,25,36,49,64,81,100,121,144,169]
```

Listing 4.2: Execution of the *verilog_mult* function

The function *verilog_mult* is executed by using the same input values. As expected the function will calculate the square, as shown in Listing 4.2

4.2 CλaSH

The CλaSH implementation is written in two languages, namely Haskell and C. The conversion to and from the *SignalStreams* is fully made in Haskell and the co-simulation is implemented using the FFI, as will be explained in subsection 4.2.3.

Listing 4.1, as shown in the previous section, shows that the co-simulation implementation is accessible with the function *coSim*. This function has the following type:

$$(CoSim\ r)\ \Rightarrow\ CoSimSettings\ \rightarrow\ CoSimulator\ \rightarrow\ String\ \rightarrow\ r$$

The first argument is a '*tuple*' containing 6 values, as shown in Table 4.1 and Listing 4.3. The second argument indicates which Verilog simulator will be used and the third argument denotes the name of the top-entity.

	Description
1	An *Int*, which denotes the HDL; for example 1 = Verilog
2	An *Int*, which denotes the length of a clock period
3	A *Bool*, which denotes if a reset phase is used
4	A *String* containing the module which will be used as top-entity
5	A *List of Strings* containing file-sources, which can be used as sub-modules
6	A *Bool* used to enable/disable the *standard output*

Table 4.1: The components of the tuple *CoSimSettings*

```
type CoSimSettings = (Int, Int, Bool, String, [String], Bool)
```

Listing 4.3: The type CoSimSettings

The *CoSimSettings* is on purpose used as first argument, because it will be generated by a *QuasiQuoter*. In section 4.4 the usage of *QuasiQuotation* and the generation of the *CoSimSettings* will be further explained.
The second and third value in the *CoSimSettings* are used for a synchronous sequential circuit, as will be explained in section 4.5.

The other argument(s) and the output(s) are defined with the type 'r'. A constraint is added such that the type 'r' must be an instance of the class *CoSim*. In the following two sections this type will be explained, which will be used to make the conversion to and from *SignalStream* possible.

4.2.1 Type Conversion

The ANSI C standard only guarantees the minimum width of C data types. For example the C 'int' can be 16 bits, 32 bits or 64 bits wide, depending on the operating system. The VPI provides special data-types, such as PLI_INT32, with a guaranteed number of bits [1][4].

The user-defined types in CλaSH can exceed the 32-bits and thus the type *vpiVectorVal*, as explained in section 3.4, will be used. Within the *vpiVectorVal*, every value is defined as a vector containing the Verilog 4-state logic values: '0', '1', 'Z', and 'X'. CλaSH only works with the logic values '0' and '1' and thus the 'Z' and 'X' values will be ignored.

The first step is to convert a CλaSH type to a list of 32-bit integers. The function *bitSizeMaybe*, from the *Data.Bits* module, will be used to retrieve the number of bits in a given type. This function returns *Nothing* for types that do not have a fixed bit size. For example, the Haskell type *Integer* is an arbitrary precision type and can hold any number up to the limit of the machine's memory.

The number of bits has to be divided by 32 and rounded up to the nearest whole number, to define the length of the output list. This division can be defined using a right shift. Using a *mapAccumR*, the function *wordPack'* will slide over the user defined value to group the bits into 32-bits values, as shown in Listing 4.4.

```
wordPack     :: (Integral a, Bits a) ⇒ a → [Int32]
wordPack x
    | isJust size = snd $ L.mapAccumR wordPack' x [1 .. wordSize]
    | otherwise   = error "Value does not have a fixed bitsize"
    where
        size     = bitSizeMaybe x
        wordSize = 1 + shiftR (fromJust size - 1) 5

wordPack'    :: (Integral a, Bits a) ⇒ a → b → (a, Int32)
wordPack' x _    = (shiftR x 32, fromIntegral x)
```

Listing 4.4: Conversion to a list of 32-bit integers with the function *wordPack*

Within every iteration, the function *fromIntegral* will convert the given type to an *Int32*. If the given type consist of less than 32 bits, sign extension will be applied. If the given type is larger than 32 bits, bits will be cut off. In both cases this is fine, because the Verilog simulator will only take the needed bits.

The input type for the function *wordPack* has as constraint *Integral a* and *Bits a*. The class *Integral* can be seen as a whole-number class, but the CλaSH types *SFixed* and *UFixed* are fixed point types. To apply the *wordPack* function on fixed point types, the CλaSH types will first be converted to a vector of bits: *BitVector n*.

BitVector n is defined as an unsigned type, with the bits indices in descending order. Furthermore, product types, like tuples and vectors, can also be converted to this bit vector. The CλaSH function *pack* makes the conversion possible, as shown in Listing 4.5.

```
λ> pack (8 :: Unsigned 6)
00_1000
λ> wordPack it
[8]

λ> pack (3.75 :: SFixed 3 3)
01_1110
λ> wordPack it
[30]

λ> pack (34350000000 :: Signed 40)
0000_0111_1111_1111_0110_1011_0110_0111_1000_0000
λ> wordPack it
[7,-9738368]

λ> pack (5 :: Signed 8, 5 :: Unsigned 8)
0000_0101_0000_0101
λ> wordPack it
[1285]

λ> pack (((0:>1:>Nil) :> (2:>3:>Nil) :> Nil) :: Vec 2 (Vec 2 (Signed 4)))
0000_0001_0010_0011
λ> wordPack it
[291]
```

Listing 4.5: Conversion to *BitVector n* and [*Int32*]

To make the conversion from [Int32] to a CλaSH type, the combination of the CλaSH function *unpack* and the function *wordUnpack* can be used. The *wordUnpack* function converts the list of 32 bits integers to the *BitVector n*, after which the *pack* function makes the conversion to the CλaSH type possible.

The function *wordUnpack* iterates over the list with the higher order function *foldl* and applies a bitwise *OR* to set the bits in the CλaSH type. The function *fromIntegral* converts a *Int32* to the bit vector. In case of negative numbers and a bit size larger than 32 bits, sign extension can influence the end result, as shown in Listing 4.6

```
λ> pack (2^33 - 1 :: Signed 36)
0001_1111_1111_1111_1111_1111_1111_1111_1111

λ> wordPack it
[1,-1]

λ> pack (fromIntegral $ (it P.!! 1) :: Signed 36)
1111_1111_1111_1111_1111_1111_1111_1111_1111
```

Listing 4.6: Influence of sign extension

Using a bitwise *AND* with the value $2^{32} - 1$ (4294967295) will avoid the possible consequences of sign-extension. Listing 4.7 shows the implementation of the *wordUnpack* function.

```
wordUnpack  :: (Integral a, Bits a) ⇒ [Int32] → a
wordUnpack      = P.foldl wordUnpack' 0

wordUnpack' :: (Integral a, Bits a) ⇒ a → Int32 → a
wordUnpack' x y = (shiftL x 32) .|. (4294967295 .&. (fromIntegral y))
```

Listing 4.7: Conversion from a list of 32-bit integers with the function *wordUnpack*

As demonstrated in Listing 4.8, the functions *unpack* and *wordUnpack* can be used to to make the conversion from a list of 32 bit integers to a CλaSH type.

```
λ> wordPack $ pack (8589934591 :: Signed 36)                    -- 2^33 - 1
[1,-1]
λ> unpack $ wordUnpack it :: Signed 36
8589934591

λ> pack (((0:>1:>Nil) :> (2:>3:>Nil) :> Nil) :: Vec 2 (Vec 2 (Signed 4)))
0000_0001_0010_0011
λ> wordPack it
[291]
λ> unpack $ wordUnpack it :: Vec 2 (Vec 2 (Signed 4))
<<0,1>,<2,3>>
```

Listing 4.8: Conversion from [*Int32*]

4.2.2 Type Classes

The functions, as explained in subsection 4.2.1, can be used to convert a CλaSH type to and from a list of 32 bit Integers. However, only a single CλaSH value can be converted to [Int32]. For a sequential synchronous circuit, a stream of CλaSH values has to be converted to a 2D list of 32 bit Integers.
Haskell is a strongly typed language and it is not possible to create a function which supports both a single CλaSH type and a stream of CλaSH types. However, within a type-class, instances can be defined to give support for different types.

Instead of using the type [Int32], the type [[Int32]] will be used for both combinational and sequential synchronous circuits. In case of a combinational circuit the outer list will have a length of 1. The type [[Int32]] will be called *SignalStream* from now on.

Inside the class *CoSimType* two functions will be created: *toSignalStream* and *fromSignalStream*, as shown Listing 4.9.

```
class CoSimType t where

    toSignalStream   :: t → SignalStream
    fromSignalStream :: SignalStream → t
```

Listing 4.9: The class *CoSimType t*

The first supported type is the required type for the composition of the *wordPack* and *pack* functions. This required type will be called *CLaSHType*.

```
type CLaSHType a = (KnownNat (BitSize a), KnownNat (BitSize a + 1),
                    KnownNat (BitSize a + 2), BitPack a)
```

Listing 4.10: The required type for a CλaSH value

In the first instance, the functions as defined in the subsection 4.2.1 will be used to convert to and from a list of 32 bit Integers. Furtermore, an empty list will be appended to convert to a 2D list. The function *head* from Haskell's *Prelude*, will be used to convert a 2D to a 1D list, as shown in Listing 4.11.

```
instance {-# OVERLAPPABLE #-} CLaSHType a ⇒ CoSimType a where

    toSignalStream   = (:[]) . wordPack . pack
    fromSignalStream = unpack . wordUnpack . Prelude.head
```

Listing 4.11: The instance for a single CλaSH value

The CλaSH type *Signal* is used, as a stream of values, for a sequential synchronous circuit. With the functions *fromList* and *sample*, a *Signal* can be converted to and from a list. In the instance as shown in Listing 4.12, the '*conversion*' functions will be mapped over the stream of values.

```
instance {-# OVERLAPPING #-} CLaSHType a ⇒ CoSimType (Signal' clk a) where

    toSignalStream          = Prelude.map (wordPack . pack) . sample
    fromSignalStream        = fromList . Prelude.map (unpack . wordUnpack)
```

Listing 4.12: The instance for a CλaSH Signal

Other instances can be added easily. For example, Listing 4.13 shows the support for a list with *Integral* values.

```
instance {-# OVERLAPPING #-} (Integral a, Bits a) ⇒ CoSimType [a] where

    toSignalStream          = Prelude.map wordPack
    fromSignalStream        = Prelude.map wordUnpack
```

Listing 4.13: The instance for list with *Integer* values

Using the class *CoSimType*, a CλaSH type can be converted to a *SignalStream*. Multiple CλaSH types should be converted to a list of *SignalStreams*. The function *parseInput*, as shown in Listing 4.14, will convert a CλaSH type and add the resulting *SignalStream* to the given list of *SignalStreams*.

```
parseInput :: CoSimType t ⇒ [SignalStream] → t → [SignalStream]
parseInput xs x             = toSignalStream x : xs
```

Listing 4.14: The *parseInput* function

The function *parseOutput* will extract one *SignalStream* from a list of *SignalStream* and convert it to a CλaSH type, as shown in Listing 4.15. In this case a function 'f' will be given as argument, to convert the input *SignalStreams* to output *SignalStreams*. This conversion from input to output arguments will only be done as a given boolean has the value *False*. As will be explained in subsection 4.2.3, the function 'f' denotes the starting point of the co-simulation.

```
parseOutput :: CoSimType t ⇒ ([SignalStream] → [SignalStream]) → Bool
                           → [SignalStream] → ([SignalStream], t)
parseOutput f u xs
    | qs == []              = error "Simulator expects less output ports"
    | otherwise             = (ys, fromSignalStream y)
    where
        (y:ys)              = qs
        qs      | u         = xs
                | otherwise = f $ Prelude.reverse xs
```

Listing 4.15: The *parseOutput* function

For co-simulation it should be possible to have multiple input and output arguments. In the ideal case, the function *parseInput* should be mapped over all the input arguments. But defining a function with a variable number of arguments, consisting of different types is normally not allowed. A variable number of inputs can be defined in a list, but a list is *homogeneous* and thus every value must have the same type. *Heterogeneous* collections are available, for example tuples, but their length is fixed and always finite. For co-simulation, heterogeneous collections with a variable length are desirable to specify the input and output ports.

In the literature different solutions are suggested to define the input and output arguments as heterogeneous collections with a variable length. Examples are the use of *HList* [36], *algebraic* or *universal* (dynamic) types. These approaches have disadvantages, like often type-switching or limited support for adding new types.

A more desirable approach is the use of another type class to define a *poly-variadic* function. A *variadic* function is a function of *indefinite arity*, meaning that the number of arguments is variable. *Poly* indicates that arguments with different types can be applied.

The class *CoSim* is defined to create the *poly-variadic* function *coSim'* which makes the conversion of the multiple arguments to and from a *SignalStream* possible.

```
class CoSim r where

  coSim' :: CoSimSettings' → Bool → [SignalStream] → r
```

Listing 4.16: The class *CoSim r*

The type *CoSimSettings'* denotes the settings needed for starting the co-simulation, as will be explained in the subsection 4.2.3. The *Bool* indicates when the co-simulation should be started. At the beginning, the value will be *False*. After all input arguments are collected, the co-simulation will start and the value will change to *True*.
The list of *SignalStreams* will be used to collect all the input arguments. After the co-simulation is started, this list will be used to save all the output *SignalStreams*.
The type '*r*' denotes the types of the input and output values. Like the *CoSimType* class, instances will be created to give support for multiple input and output types.

A *poly-variadic* function can be created because of the definition of a Haskell function. All Haskell functions are considered curried: "*all functions in Haskell take just one argument*" [37]. Arrow notation associates to the *right*, as shown in the following example:

$$f :: a \to b \to c \to d \quad \text{is the same as} \quad f :: a \to (b \to (c \to d))$$

However, function application associates to the *left*:

$$f\ a\ b\ c\ d \quad \text{is the same as} \quad (((f\ a)\ b)\ c)\ d$$

Partial application can be used to define the *CoSim* instance for collecting all the input arguments. This instance will recursively convert all the input arguments to a *SignalStream*, as shown in Listing 4.17.

```
instance {-# OVERLAPPING #-} (CoSimType t, CoSim r) ⇒ CoSim (t → r) where

    coSim' s u xs           = coSim' s u . parseInput xs
```

Listing 4.17: The *CoSim* instance for collecting all the input arguments

Multiple output arguments are defined as a tuple, in which the right side is defined as a recursive tuple. Within this instance, the output *SignalStreams* are recursively converted to the user-defined CλaSH types.

```
instance {-# OVERLAPPING #-} (CoSimType t, CoSim r) ⇒ CoSim (t, r) where

    coSim' s u xs           = (y', y'')
        where
            (ys, y')        = parseOutput (coSimStart s) u xs
            y''             = coSim' s True ys
```

Listing 4.18: The *CoSim* instance for converting multiple outputs values

As visible in Listing 4.18, the function *coSimStart*, partial applied with the *CoSimSettings'*, is given as argument to the function *parseOutput*. The function *coSimStart* is the starting point of the co-simulation, as will be explained in subsection 4.2.3.

The last instance will convert one output value, as shown in Listing 4.19. This instance can be seen as the stopping point of the recursive conversions of the input and output values. It is important to notice that this instance will be executed exactly once. The co-simulation can thus only be performed with at least one output value.

```
instance {-# OVERLAPPABLE #-} CoSimType t ⇒ CoSim t where

    coSim' s u xs
        | ys == []          = y'
        | otherwise         = error "Simulator expects more output ports"
        where
            (ys, y')        = parseOutput (coSimStart s) u xs
```

Listing 4.19: The *CoSim* instance for converting one output value

The type conversion process can be shown graphically, as visible in Figure 4.2.

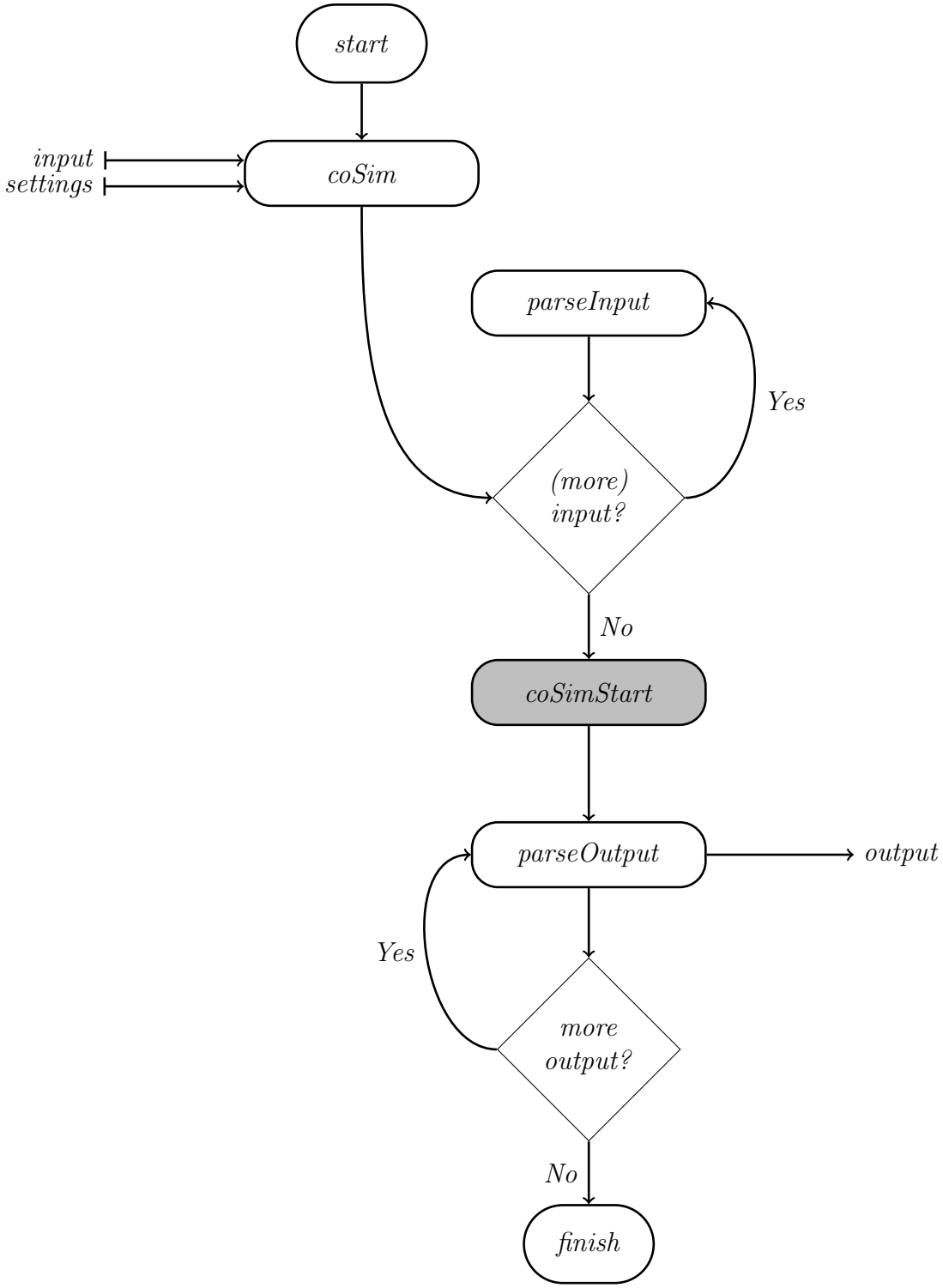


Figure 4.2: The schematic overview of the input and output conversions

4.2.3 Foreign Function Interface

The *Foreign Function Interface* (FFI), as explained in subsection 2.1.1, will be used to execute C-functions in CλaSH. The FFI is an extension to the Haskell standard to communicate with foreign languages. Foreign imports will be made and the imported functions can be executed. It is also possible to inline the FFI using *Template Haskell* (TH) and then it is not needed to implicitly define the foreign imports. For example *inline-c*, a package to embed C-code in Haskell, uses this principal.
This implementation will not uses packages like *inline-c* and stick with the FFI. One of the main reasons is that *inline-c* will create files for every C-function and these files will be compiled/linked on the fly. This will create overhead, because the C-functions, used in the co-simulation, will not be edited on user-level and can be statically compiled and linked before running any simulation.

As a recap, the user will call the *coSim* function, in which the *CoSimSettings*, the *CoSimulator*, and the name of the top-entity will be put together in a tuple, called *CoSimSettings'*. This tuple, together with a boolean flag, and an empty list will be given to the *coSim'* function. As explained in subsection 4.2.2, the function *coSim'* will recursively collect all the input arguments, after which the function *coSimStart* will be executed.

```
coSim :: (CoSim r) ⇒ CoSimSettings → CoSimulator → String → r
coSim settings sim top      = coSim' (settings, sim, top) False []
```

Listing 4.20: The *coSim* function

The function *coSimStart*, as shown in Listing 4.22 and Figure 4.3, will start the Verilog simulator and map the input values to the appropriate simulation steps. A part of this function is implemented in C. The C code will execute the operating system dependent code, like starting the Verilog simulator and setting up the communication between CλaSH and the Verilog simulator. On a Linux system, executing a process can be accomplished with a fork. Shared memory, fifos, or sockets are often used to communicate between processes.

To call the C code, foreign imports have to be made. It is a common idiom to expose the imported C functions with the prefix "c_", to distinguish these function from the higher level functions in Haskell, as shown in Listing 4.21. It is important to notice that with the function '*simEnd*' only a pointer to this function is imported. A pointer to a function is in Haskell defined as *FunPtr* (Function Pointer). This pointer will be used to connect to Haskell's *Garbage Collector* (GC).

```
foreign import ccall "simStart" c_simStart :: Ptr CInt → CString →
                                              Ptr CString → IO (Ptr a)
foreign import ccall "&simEnd"  c_simEnd   :: FunPtr (Ptr a → IO ())
```

Listing 4.21: The foreign import of the *c_simStart* and *c_simEnd* functions

The co-simulation will be executed using lazy evaluation. Using lazy evaluation has as drawback that it is unknown when the co-simulation will end. This moment has to be known to finish the Verilog simulator and close the communication. A solution is to execute the simulation strictly and define in advance (user-level) the number of clock-cycles. Strict evaluation makes the use of feedback loops impossible.

A better solution is to use Haskell's *Garbage Collector*. Normally, memory management between the Haskell code and the foreign code is separated explicitly. The Haskell storage manager takes care of the Haskell side, and the foreign side must be handled manually. The exception to this rule is when the foreign pointer is associated with a *finalizer*. In this case, the Haskell storage manager will detect (within the Haskell heap and stack) that there are no more references (at the Haskell side) pointing to this foreign pointer. When there are no more references, the finializer will be executed.

The exact moment when the *finalizer* is executed is unknown. In the documentation it is stated that this will be done after the last reference to the foreign object is dropped, but there is no guarantee of promptness. However the finalizer will be executed before the program exits [66]. To have more control on when the finalizer is exectued, garbage collection will be performed before starting the co-simulation within the function *coSimCleanUp*, as shown in Listing 4.22. Manually performing garbage collection will reduce memory usage of previous simulations and can close previous started Verilog simulators.

```
coSimStart :: CoSimSettings' → [SignalStream] → [SignalStream]
coSimStart settings xs = unsafePerformIO $ do

    -- garbage collection
    coSimCleanUp

    -- marshall c-types
    (rst, c_sPtr, c_topE, c_fPtrs) ← coSimMarshall settings xs

    -- start simulation
    c_coSimState'    ← c_simStart c_sPtr c_topE c_fPtrs
    when (c_coSimState' == nullPtr) $ error "Start co-simulation failed"

    -- add finilizer
    c_coSimState     ← newForeignPtr c_simEnd c_coSimState'

    -- perform simulation steps
    c_oLength        ← withForeignPtr c_coSimState c_outputLength
    (_, ys)          ← mapAccumLM coSimStep c_coSimState $ f rst xs

    -- transpose and return
    return $ transposeList c_oLength ys
    where
        f r | r          = ([]:) . L.transpose
            | otherwise  = L.transpose
```

Listing 4.22: The function *coSimStart*

The finalizer will invoke routines in the foreign languages to free the appropriate resources. As shown in Listing 4.22, the C function *c_simEnd* is used as finalizer. This finalizer is connected, with the function *newForeignPtr*, to the output of the *c_simStart* function: the struct *c_coSimState*. When there are no references to this struct, the function *c_simEnd* will be executed. This function will deallocate the used C memory and finish the Verilog simulator.

Before starting the *c_simStart* function, the needed arguments must be marshalled to the appropriate C types. The types are part of the FFI and the names starts with a '*C*' (like *CString* and *CInt*), to explicitly show their origin. The marshalling is implemented in the function *coSimMarshall*, as shown in Appendix C.

One of the easiest marshalling is between *Int* and *CInt*. Both types derive the *Integral* and *Num* class and thus converting in both directions can be done with the *fromIntegral* function. This function converts from any *Integral* type into any *Numeric* type.

```
fromIntegral :: (Num b, Integral a) ⇒ a → b
```

Listing 4.23: The *fromIntegral* function

The FFI makes an explicit distinction between a foreign array and a *CString*. The comparable type in Haskell is in both cases a List[1]. The *CString* type is an array with C-characters terminated by NULL. Furthermore, the marshalling consist of converting each Haskell character to C-encoding (normally Unicode, but also 8-bit characters, and Unicode variants like UTF-16 and UTF-32 are possible). Converting to a foreign CString can be done with the *newCString* function and reading a foreign CString can be done with the *peekCString* function.

```
type CString = Ptr CChar
newCString   ::  String → IO CString
peekCString  :: CString → IO String
```

Listing 4.24: String conversion

An foreign array is defined as a '*Ptr a*'[2]. The FFI contains functions for allocating and marshalling a foreign array. For example, the *mallocArray* function allocates storage for the given number of elements. The *peekArray* and *pokeArray* are functions to convert a list to and from a foreign array. The function *newArray* is a combination of the *malloc* and the *poke* functions.

```
mallocArray :: Storable a ⇒ Int → IO (Ptr a)
peekArray   :: Storable a ⇒ Int → Ptr a → IO [a]
pokeArray   :: Storable a ⇒ Ptr a → [a] → IO ()
newArray    :: Storable a ⇒ [a] → IO (Ptr a)
```

Listing 4.25: Array conversion

[1] A 'String' is defined as [Char]

[2] Other pointers, for example structs, are also defined as a *Ptr a*.

After marshalling, the function *c_simStart* can be called with the appropriate variables. The embedded Verilog module will be written to a file and additional Verilog files can be given as argument. Instead of using single files, it also possible to specify directories, as will be explained in section 4.4. The *c_simStart* function will iterate through these directories to get all the available Verilog modules.

Based on the given Verilog simulator, the instructions to compile the Verilog module(s) and to start the Verilog simulator will be defined. These instructions will be executed with a *fork* and the *execvp* command. The *execvp* command replaces the current process image with a new process image. Replacing the current process image is ideal for starting the Verilog simulator, because file descriptors remain open. Fifos will be used as communication medium between CλaSH and the Verilog simulator. The needed file descriptors for the VPI application, will be shared through environment variables.

When the sources are compiled, the Verilog simulator will be started with the VPI application, as will be described in section 4.3. The VPI application will automatically iterate through the top-module to get the available input and output ports.

When the ports of the top-entity are known, CλaSH and the VPI application will exchange the specifications needed for the co-simulation. These specifications include the number of input and output variables (ports), the bit-sizes of every port and optionally the length of a clock-cycle. If this information matches at both sides, the co-simulation can be performed.

The specifications needed for every simulation step will be saved in a struct, called *coSimState*. Within this struct the needed memory for every simulation step will be allocated. Listing 4.26 shows the content of this *coSimState*.

```
struct coSimState {
    struct CoSimComm *comm;   // communication with the Verilog simulator
    char strR[MAX_BUF+1];     // used within the communication
    int reset;                // no of reset cycles
    int noInput;              // no of input-ports
    int noOutput;             // no of output-ports
    int *inputSizes;          // size of every input-port
    int *outputSizes;         // size of every output-port
    int **input;              // allocated memory for the input-ports
    int **output;             // allocated memory for the output-ports
    int noFiles;              // no of files, which has to be deleted
    char **fileNames;         // file-paths, these files will be deleted
};
```

Listing 4.26: The struct coSimState

When the struct *coSimState* is created and connected to Haskell's Garbage Collector, the input values can be mapped onto every simulation step. However, the input values are still defined as a list of *SignalStreams*. This list has to be transposed, after which a value of every *SignalStream* can be written into the simulator in every simulation step, as shown in Figure 4.3.

The mapping to every simulation step is implemented with the function *coSimStep* and the higher order function *mapAccumLM*, a *Monadic* version of *mapAccumL*. The *mapAccumLM* is a lazy version of the *mapAccumLM* as defined in GHC's *MonadUtils*. The implementation, and the consequences, of the lazy *mapAccumLM* will be described in chapter 6. Furthermore, the *coSimStart* function uses the function *unsafePerformIO* to break out the *IO-monad*. The consequences of this function will also be described in chapter 6.

After performing the simulation steps, the output-values have to be transposed (again) to create the appropriate *SignalStreams*. Finally, the output *SignalStreams* will be converted to the user defined types, as described in subsection 4.2.2.

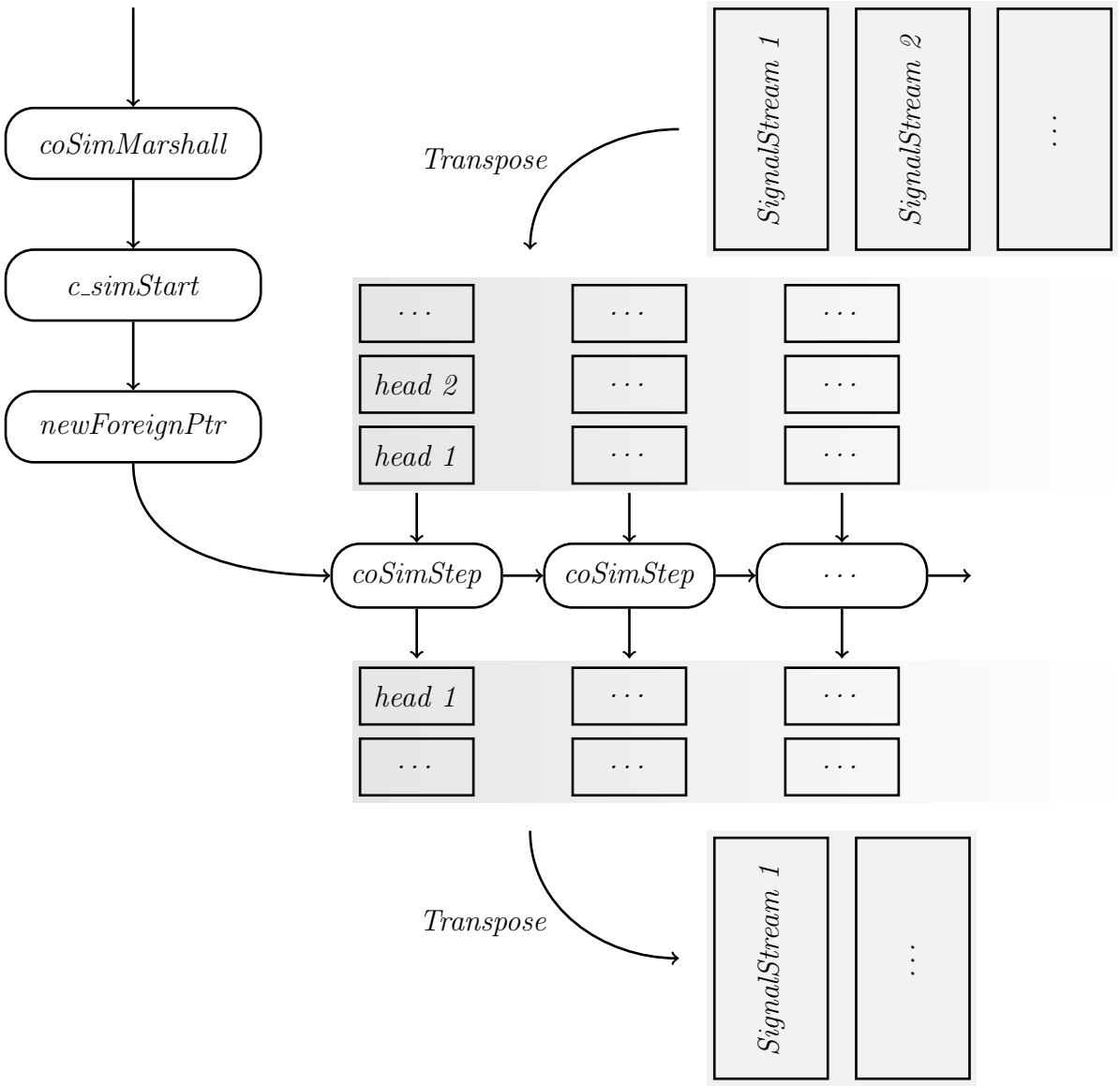


Figure 4.3: The schematic overview of the function *coSimStart*

The function *coSimStep* writes the input-values into the struct *coSimState*, performs a simulation step, and reads the output-values from the struct *coSimState*. The function has as input the foreign pointer to the *coSimState* and a list with *Int32* values. As output, a list with *Int32* and the same foreign pointer will be returned.
Although the memory address will be the same, the content of the *coSimState* will be different. Furthermore, it is important that the simulation steps are performed sequentially, because every step depends on the previous step. Using a function like *mapM* will gives the indication that the simulation-steps are independent. As workaround, the function *mapAccumLM* is used to make the simulation steps dependent.

The functions *coSimInput* and *coSimOutput* will respectively write and read the input and output values, as shown in Listing 4.27. The implementation of these two functions is shown in Appendix C.

```
foreign import ccall "simStep" c_simStep :: Ptr a → IO CInt

coSimStep :: ForeignPtr a → [[Int32]] → IO (ForeignPtr a, [[Int32]])
coSimStep state xs = do

    -- write input
    coSimInput state xs

    -- perform simulation step
    rv ← withForeignPtr state c_simStep
    when (rv /= 0) $ error "Error in co-simulation step"

    -- read output
    ys ← coSimOutput state

    -- touch state, to keep state alive
    touchForeignPtr state

    -- return output
    return (state, ys)
```

Listing 4.27: The function *coSimStep*

A part of the simulation step is implemented in the C function *simStep*. In this C function the input values will be sent to the Verilog Simulator and the output values will be retrieved from the Verilog simulator. However, as visible in the type definitions of the *c_simStep* and *coSimStep* functions, the type of the *coSimState* is different. The function *coSimStep* uses a *ForeignPtr* which is connected to the Garbage Collector, but the function *c_simStep* uses a normal pointer. A solution is to use the function *withForeignPtr*, which will apply the encapsulated pointer to the *c_simStep* function and also ensures that the foreign pointer is kept alive at least during the whole execution of this function.

According to the C standard, a return value of 0 denotes successful termination. If the return value of the *c_simStep* is not equal to zero, the simulation will be aborted.

```
withForeignPtr   :: ForeignPtr a → (Ptr a → IO b) → IO b
touchForeignPtr  :: ForeignPtr a → IO ()
```

Listing 4.28: Foreign pointer functions

The function *touchForeignPtr* is used to keep a foreign pointer alive. After calling the *c_simStep* function, pointers from the *coSimState* are used to read the output data. Using *normal* pointers, instead of the foreign pointers, can send a fake signal to the garbage collector and execute the finalizer. The *touchForeignPtr* is used to indicate that the *coSimState* is still in use.

The C function *c_simStep* will only forward the input values and retrieve the output values, as shown in Listing 4.29 and Listing 4.30. Notice that the order of the *Int32* values in the inputs and outputs is reversed. In CλaSH, the *Least Significant Bit* (LSB) is defined in the last *Int32* value, but in Verilog the LSB is defined in the first *Int32* value. However, the bits in an *Int32* value are defined in the same order.

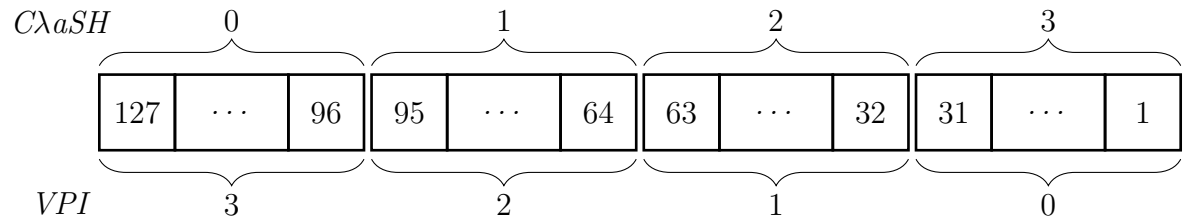


Figure 4.4: The ordering of the *Int32* values

```
for (i = 0; i<state→noInput; i++)
{
    for (j = state→inputSizes[i]-1; j≥0; j--)
    {
        // send 'aval' to simulator
        sprintf(state->strR, "%d", state→input[i][j]);
        if (writeMessage(state→strR,state→comm, 1) < 0) return -1;
    }
}
```

Listing 4.29: Sending the input values to the Verilog simulator

```
for (i = 0; i<state→noOutput; i++)
{
    for (j = state→outputSizes[i]-1; j≥0; j--)
    {
        // get 'aval' from simulator
        if (readMessage(state→strR, state→comm, 1) < 0) return -1;
        state->output[i][j] = atoi(state→strR);
    }
}
```

Listing 4.30: Retrieving the output values from the Verilog simulator

4.3 VPI

A VPI application, as explained in chapter 3, will be loaded in the Verilog simulator. The VPI implementation denotes the right side of Figure 4.1, which was given at the beginning of this chapter. As is explained in subsection 4.3.1 two simulation callbacks and three callbacks for every simulation cycle will be used to exchange data with CλaSH and thus to define co-simulation.

Inside the callback *cbStartOfSim* the Verilog-module(s) inside the simulator will be analysed and the specifications of the top-entity will be exchanged with CλaSH, as will be explained subsection 4.3.2. At the end of the *cbStartOfSim* callback, events for the first simulation cycle will be registered to make value-exchange possible. Reading and writing of values will be done once a clock-cycle, to support delay values in the Verilog code. The implementation of the value exchange will be explained in subsection 4.3.3.

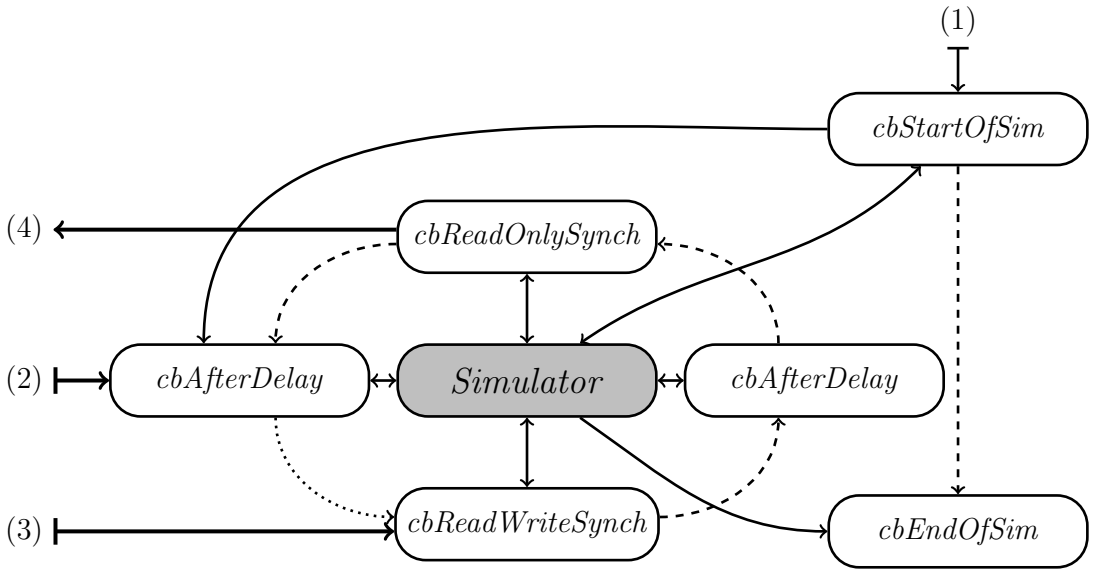


Figure 4.5: Overview of the VPI application

The VPI application will register the *cbStartOfSimulation* callback at the beginning of the execution (1). This callback will register the *cbEndOfSimulation* callback and directly execute the *cbAfterDelay* routine. The *cbAfterDelay* will wait untill CλaSH requests to perform the next clock-cycle (2), after which the *cbReadWriteSynch* will be registered. CλaSH will send values (3), which will be written into the simulation. The *cbReadOnlySynch* callback will be registered at the end of the clock cycle, in which values from the simulation will be read and sent to CλaSH (4). The *cbReadOnlySynch* will register the *cbAfterDelay* and the process repeats.
The *cbEndOfSimulation* routine will be executed when the simulation finished, in which the allocated memory will be deallocated.

4.3.1 Simulation Callbacks

The main starting point of the VPI implementation is the *vlog_startup_routines*. This is an array with function-pointers, which will be executed by the Verilog-simulator. By using the VPI, this function pointer array will always be the beginning of a VPI application.

```
/* array with function-pointers which will be loaded by the simulator */
void (*vlog_startup_routines[])(void) = {
    registerCallbacks,
    0
};
```

Listing 4.31: The vlog_startup_routines

In this case, only the function *registerCallbacks* is registered. Furthermore, the last item in the array is a '0' as required by the standard. The Verilog-Simulator does not know how many functions will be registered and will iterate through the array until it finds a *null-pointer*. The function *registerCallbacks*, as shown in Listing 4.32, registers the *cbStartOfSimulation* callback. This callback is a simulation action event, which will be raised when the simulation starts.

```
void registerCallbacks(void)
{
    registerCB(NULL, startOfSim, cbStartOfSimulation, -1);
}
```

Listing 4.32: Registration of the *cbStartOfSimulation* callback

The higher order function *registerCB* will register a callback in the Verilog simulator, as shown in Listing 4.33. This function has four input arguments: the struct *vpiState*, a callback function, the callback reason, and a delay.
The struct *vpiState* is used within every registered simulation step and will be created when the design is analysed, as will be explained in subsection 4.3.2.
The callback function is a function pointer, which will be executed by the Verilog simulator for the given callback reason.
The delay is used as the relative time measured from the current simulation time. A negative delay indicates no time registration.

The struct *s_cb_data* is needed to register a callback (cb), as explained in section 3.2. The field *time* is used as relative or absolute delay, depending on the callback. Icarus Verilog does allow registering callbacks with a negative time, after which the timing will be ignored. In this case, the time will be set to a NULL-pointer when a negative delay is given. Simulation-action callbacks, like *cbStartOfSimulation* and *cbEndOfSimulation* require a NULL-pointer. On the other hand, simulation-time-related callbacks require a time. Currently, the timing can only be set as a 32-bit value. If more than 4 billion simulation steps are needed, an extra argument can be added to set the *high* field.

```
typedef PLI_INT32 (*f_cb)(p_cb_data cb_data);

void registerCB(struct vpiState *state, f_cb f, PLI_INT32 reason, PLI_INT32
    time)
{
    // variables
    s_cb_data           cb_data_s;
    s_vpi_time          time_s;

    // time
    if (time < 0)
    {
        cb_data_s.time  = NULL;
    }
    else
    {
        cb_data_s.time  = &time_s;
        time_s.type     = vpiSimTime;
        time_s.high     = 0;
        time_s.low      = time;
    }

    // settings
    cb_data_s.reason    = reason;
    cb_data_s.cb_rtn    = f;
    cb_data_s.user_data = (PLI_BYTE8 *) state;

    // register
    vpi_free_object(vpi_register_cb(&cb_data_s));
}
```

Listing 4.33: The *registerCB* function

The *reason* and the *callback routine* (*cb_rtn*) are both user defined. The callback routine has a required type definition, as defined with the typedef *f_cb*.
The last used field of the *s_cb_data* is the *user_data*. The book [1] strongly discourage sharing data using global variables and instead recommend using the *user_data*. The struct *vpiState* is used as the user data and will be explained in the next section.

Using the function *vpi_register_cb* the callback is registered and a handler is returned. The handler can be used to change or remove the callback. In this case the handler will not be used and is explicitly freed with the function *vpi_free_object*.

4.3.2 Analysing Design

In the start-of-simulation callback the Verilog top-module will be analysed. The VPI application is executed as a child-process. Before the *fork*, fifos are created between CλaSH and the VPI application, as explained in subsection 4.2.3. Although open file-handles can still be used in a child-process, variables are not forwarded and thus the file-handles are unknown in the beginning. The function *getSharedComm* uses *environment variables* to retrieve the communication specific information.

```
PLI_INT32 startOfSim(p_cb_data __attribute__((__unused__)) cb_data)
{
    p_cb_data cb_data_p;
    struct vpiState *state;

    // init
    cb_data_p = malloc(sizeof(struct t_cb_data));
    state = createState();

    if (getSharedComm(&state->comm) < 0) return abortSim();

    // read seq settings
    if (exchangeSeq(state) < 0) return abortSimM(state);

    // get port specifications
    if (getModuleSpecs(NULL, state) < 0) return abortSimM(state);

    // exchange ports info
    if (exchangePortSpecs(state, 1) < 0) return abortSimM(state);
    if (exchangePortSpecs(state, 0) < 0) return abortSimM(state);

    // register end-of-sim
    registerCB(state, endOfSim, cbEndOfSimulation, -1);

    // register event for first simulation step
    cb_data_p->user_data = (PLI_BYTE8 *) state;
    synchStep(cb_data_p);

    // free cb-data
    free(cb_data_p);

    return 0;
}
```

Listing 4.34: The *cbStartOfSimulation* callback function

The function *exchangeSeq* is only used to exchange information needed for sequential synchronous circuits. CλaSH will send the length of a clock-cycle, specified as a number of simulation steps. This number will be used to register read and write callbacks for the data exchange. Furthermore, CλaSH will indicate if in the first clock-cycle the write phase has to be skipped, which is sometimes necessary for having a reset phase.

The top-entity will be analysed in the function *getModuleSpecs*. Icarus Verilog does not support iterating through the modules with, for example, the *vpiModPath*. The properties *vpiModule* and *vpiPort* will be used to automatically recognize the available input and output ports.

```
int getModuleSpecs(vpiHandle handle, struct vpiState *state)
{
    ...

    /* iterate through all modules in current scope */
    mod_itr = vpi_iterate(vpiModule, handle);
    if (mod_itr != NULL)
    {
        /* scan the modules and get a handle to each */
        while ( (mod_handle = vpi_scan(mod_itr)) != NULL)
        {
            /* get module name */
            tmp = vpi_get_str(vpiName, mod_handle);
            mod_name = (char*) malloc((strlen(tmp)+1) * sizeof(char));
            strcpy(mod_name, tmp);

            /* get the ports in this module */
            getPortSpecs(mod_handle, state, mod_name);

            free(tmpModuleName);
        }
    }

    ...
    return 0;
}
```

Listing 4.35: Traversing Verilog design

The VPI routines *vpi_iterate* and *vpi_scan* are also used to iterate over all available ports in a module, as shown in Listing 4.36. Furthermore, the routines *vpi_get* and *vpi_get_str* are used to retrieve port-specific information, like the name (*vpiName*), direction (*vpiDirection*), and bit-size (*vpiSize*).

```
int getPortSpecs(vpiHandle handle, struct vpiState *state, char *modName)
{
    ...

    // iterate through the ports
    itr = vpi_iterate(vpiPort, handle);
    if (itr != NULL)
    {
        while ((port_handle = vpi_scan(itr)) != NULL )
        {
            ...
```

Listing 4.36: Iterating through the module ports

After knowing all available ports in the VPI application, information about the input and output ports can be exchanged with CλaSH. The number of ports and the size of every port has to match with the information at the CλaSH side. If the information matches between the two processes, the VPI application will save a handle to every object connected to the ports. Examples of objects are wires (*vpiNet*) or registers (*vpiReg*). By saving a handle to every object, values can directly be read or written to an object, without the need of traversing the design again.

A handle to every object is obtained with the VPI routine *vpi_handle_by_name*, in which the *full-name* must be given. For example, a handle to the input-wire *left*, as shown in Listing 4.37 is retrieved with the full-name *mult.left*.

```
module mult (left,right,out);

input    [63:0] left;

...

endmodule
```

Listing 4.37: The Verilog module *mult*

The routine *vpi_handle_by_name* is actually a work-around and not recommended. The simulator has to traverse the hierarchy again to get the handle of an object. The most efficient way is to use the *vpiLowConn* and *vpiHighConn* properties to directly retrieve the port's connected object. Icarus Verilog does not support the *vpiLowConn* and *vpiHighConn* properties and thus the routine *vpi_handle_by_name* is used.

The handles to the port will be saved in the struct *vpiState*. This struct will also contain the communication information (file handles), the length of a clock-cycle (period), and the check if a reset-phase is used.

```
struct vpiState
{
    struct CoSimComm *comm;     // communication with CλaSH
    char strR[MAX_BUF+1];       // for receiving messages from CλaSH
    int period;                 // length of a clock-cycle
    int reset;                  // reset-phase
    int noInput;                // no of input-ports
    int noOutput;               // no of output-ports
    struct port *inputPorts;    // port specific information, like handles
    struct port *outputPorts;   // port specific information, like handles
    s_vpi_time time;            // time needed for the write callbacks
    s_vpi_value vector;         // vector needed for the write callbacks
};
```

Listing 4.38: The struct *vpiState*

The callback *cbEndOfSimulation* is used to deallocate the struct *vpiState*, as shown in Listing 4.39. The function *endOfSim* will be executed when the Verilog simulation finishes.

```
PLI_INT32 endOfSim(p_cb_data cb_data)
{
    struct vpiState *state;

    // dispose state
    state = (struct vpiState*) cb_data->user_data;
    disposeState(&state);

    return 0;
}
```

Listing 4.39: The *cbEndOfSimulation* callback function

Finally, the *startOfSim* function will register the first read or write callback (depending on the reset phase) using the function *synchStep*. As will shown in the next section, subsection 4.3.3, the function *synchStep* is also used for the *cbAfterDelay* callback. This callback can be used to execute a function after a certain (relative) delay.

```
PLI_INT32 synchStep(p_cb_data cb_data)
{
    struct vpiState *state;

    // init
    state = (struct vpiState*) cb_data->user_data;

    // if CLaSH send 'finish', finish simulation, else register new cbs
    if (readMessage(state->strR, state->comm, 1) < 0) return abortSim();
    if (strcmp("finish", state->strR) == 0) vpi_control(vpiFinish, 0);
    else
    {
        // register next event
        if (state->reset-- > 0)
            registerCB(state, registerRD, cbAfterDelay, state->period-1);
        else
            registerCB(state, readWriteSynch, cbReadWriteSynch, 0);
    }

    return 0;
}
```

Listing 4.40: The *synchStep* function

In case of a reset phase, the function *registerRD* with the callback *cbAfterDelay* will be registered. This function will be executed at the end of the clock-period, in which the callback *cbReadOnlySynch* is registered. The callback *cbReadOnlySynch* is used for reading values and the callback *cbReadWriteSynch* is used for writing values, as will be explained in the next section.

4.3.3 Reading & Writing Values

After analysing the desing, simulation-cycle related callbacks can be registered to be able to exchange values between CλaSH and the Verilog simulator. Three callbacks are used: *cbAfterDelay*, *cbReadWriteSynch*, and *cbReadOnlySynch*. As explained in section 3.2, these three callbacks occur at certain moments in a simulation time step.

The *cbAfterDelay* callback occurs once after a given delay and before execution of any simulation events in the specific time-step. Using a delay of '1', the callback will occur in the next time-step. CλaSH will indicate, using *lazy-evaluation*, when and if next time-steps are needed. The VPI application will wait in function *synchStep* until CλaSH indicates that a new time-step has to be performed or if the simulation has to finish.
If a new time-step is desired, a read or write callback will be registered, as shown in Listing 4.40. If CλaSH indicates that the simulation has to stop, the VPI application will use the routine *vpi_control*, with the *vpiFinish* flag, to finish the simulation.

In case of a write event, the *cbReadWriteSynch* callback will be registered with a delay of '0'. The function *readWriteSynch* is used as callback routine, in which values will be written in the Verilog simulator. All the values are written in the simulation at the same moment (*zero delay*), using the *vpiInertialDelay* flag to use the simulator's event scheduling. The *cbReadWriteSynch* callback can occur in *Slot 2* or in *Slot 3*, as explained in section 3.2, and thus the order of the *non-blocking assignments* and *co-simulation assignments* is unknown.

The *vpiVectorVal* format is used to exchange the values. With this format a value will be defined as a 4-state logic value with an encoded pair of 32-bit C integers. Within CλaSH , only the logic values *'0'* and *'1'* are used and thus the *bval* is set to '0'.

```
for (i = 0; i<port→width; i++)
{
    // read a value from CLaSH
    if (readMessage(state→strR, state→comm, 1) < 0) return abortSim();

    // set value in the vector
    port→vector[i].aval = atoi(state→strR);
    port→vector[i].bval = 0;
}

// write vector into simulator
state->vector.value.vector = port→vector;
vpi_put_value(port→handle, &state→vector, &state→time, vpiInertialDelay);
```

Listing 4.41: Writing a value into the simulator

A read event is postponed to the end of a clock-cycle with the *registerRD* function. In this function the *cbReadOnlySynch* callback will be registered, as shown in Listing 4.42. The *cbReadOnlySynch* occurs once in *Slot 4*, after execution of *all* events in a specific time-step. For a combinatorial circuit there is no need to postpone the read event, but in this way the implementation can be used for both *combinational* and *synchronous sequential* circuits. The only difference between both circuits, is that in case of a *combinational* circuit, the whole simulation will happen in one simulation period.

```
PLI_INT32 registerRD(p_cb_data cb_data)
{
    struct vpiState *state;

    // register cb
    state = (struct vpiState*) cb_data->user_data;
    registerCB(state, readOnlySynch, cbReadOnlySynch, 0);
    return 0;
}
```

Listing 4.42: The *registerRD* function

Instead of using the *vpi_put_value* routine, the *vpi_get_value* routine will be used to retrieve values from the simulator. The logic values *'X'* and *'Z'* are ignored and only the *'0'* and *'1'* values, grouped in 32-bits, are send to CλaSH.

```
vpi_get_value(port->handle, &state->vector);

for (i = 0; i<port->width; i++)
{
    // convert vector[i] to '0' or '1'
    val = state->vector.value.vector[i].aval;
    val &= ~state->vector.value.vector[i].bval;

    // send value to CLaSH
    sprintf(state->strR,"%d",val);
    if (writeMessage(state->strR, state->comm, 1) < 0) return abortSim();
}
```

Listing 4.43: Reading a value from the simulator

Instead of setting the *'X'* and *'Z'* values to *'0'*, the *bval* values could be ignored. The *'Z'* will then be represented with a *'0'* and *'X'* with a *'1'*.

4.4 Inline Verilog

As explained in subsection 2.1.2, *Template Haskell* and *QuasiQuotation* can be used to define a *Domain Specific Language* (DSL). A *QuasiQuoter* is defined to embed Verilog code in CλaSH code. Only the algebraic data type *Exp* is used, because the embedded Verilog module will only be used as expression.

The function *createQuasiQuoter*, as shown in Listing 4.44, creates a *QuasiQuoter*. As argument, a function with the type *String* → *Q Exp* must be given, which is used as the quote expression.

```
createQuasiQuoter :: (String → Q Exp) → QuasiQuoter
createQuasiQuoter f = QuasiQuoter
                          {quoteExp  = f
                          ,quotePat  = undefined
                          ,quoteType = undefined
                          ,quoteDec  = undefined}
```

Listing 4.44: The function *createQuasiQuoter*

Currently, only one *QuasiQuoter*, called *verilog*, is created to embedded a Verilog module in CλaSH. As shown in Listing 4.45, the function *createQuasiQuoter* is called with as argument the partial applied *inlineCoSim* function. The given argument (the value '1') denotes the Verilog language. The HDL is specified to make (future) *QuasiQuoter* implementations for VHDL and SystemVerilog possible.

```
verilog :: QuasiQuoter
verilog             = createQuasiQuoter $ inlineCoSim 1
```

Listing 4.45: The Verilog *QuasiQuoter*

The function *inlineCoSim* creates the *CoSimSettings*. These settings are used to start the co-simulation. Only the HDL and the top-entity are given as argument. In case of Verilog, the HDL will be set to '1' and the top-entity is the embedded Verilog module (defined as String). The other settings are set to default values. The period will be set to '20' and the reset phase will be disabled. These two settings will be further explained in section 4.5.
Furthermore, extra sub-modules, given as files, can be added. Sub-modules can be added using the path to the files, or by defining the directory which contains the source files. Initial, no sub-modules are defined and the list is thus empty.
The last setting can be used to enable or disable the standard output of the (Verilog) simulator. The default setting enables the standard output.

```
inlinlineCoSim :: Int → String → Q Exp
inlineCoSim hdl s   = liftM TupE $ sequence [q_hdl, q_period, q_reset,
   q_data, q_list, q_stdOut]
    where
        q_hdl       = lift hdl
        q_period    = lift (20 :: Int)
        q_reset     = lift False
        q_data      = lift s
        q_list      = lift ([] :: [String])
        q_stdOut    = lift True
```

Listing 4.46: The generation of the CoSimSettings

With the function *lift*, the separate values of the *CoSimSettings* are lifted into the Q monad. The function *sequence* is used to transform the [*Q Exp*] to *Q* [*Exp*].
The function *liftM* gives the [*Exp*] as argument to the expression constructor *TupE*. The constructor transforms the list to a tuple, after which the tuple is returned as expression.

The example in Listing 4.1, at the beginning of this chapter, already shows the usage of the *QuasiQuoter*. The Verilog module must be defined inside *Oxford brackets* with the verilog keyword.

[*verilog*| *module ... endmodule* |]

Currently, the embedded Verilog module is only forwarded to the Verilog simulator and the *QuasiQuoter* does not parse the Verilog code. The simulator will compile the Verilog code and indicate if the code is correctly defined. However, the recommendation is given to define a Verilog parser in CλaSH, as will be described in section 6.4. Defining a Verilog parser gives as main advantage that the correctness of the Verilog code will be checked at compile time.

Functions are defined to update the generated *CoSimSettings*, as shown in Listing 4.47. The functions *coSimEnableStdOut* and *coSimDisableStdOut* are used to enable or disable the standard output of the Verilog simulator.
A list of with source-paths can be given to the function *coSimWithFiles* to add sub-modules.
The function *coSimSeq* can be used, besides adding additional sub-modules, to update the clock period and reset phase, as will be described in the next section.

```
coSimEnableStdOut  :: CoSimSettings → CoSimSettings
coSimDisableStdOut :: CoSimSettings → CoSimSettings
coSimWithFiles     :: CoSimSettings → [String] → CoSimSettings
coSimSeq           :: CoSimSettings → (Int,Bool) → [String] →
                                                    CoSimSettings
```

Listing 4.47: The function definitions for updating the *CoSimSettings*

4.5 Clock Cycles

A *Synchronous Sequential Circuit* design is based on streams of values, called *Signals*. In the CλaSH tutorial the following statement about *Signals* is given: *"A Signal is an (infinite) list of samples, where the samples correspond to the values of the Signal at discrete, consecutive, ticks of the clock"* [3].

The co-simulation implementation works with streams of *Int32* values and actually there is no distinction between clocks, resets and ordinary signals. For a *Synchronous Sequential Circuit* this can actually be problematic, because a clock-stream and a signal-stream are not the same. A clock consist normally of low and high periods. The lengths of these periods define the speed of the clock. From functional point of view, a clock stream has at least twice as much samples compared with the samples in a CλaSH signal (a '0' and a '1' in one clock-cycle).

In the CλaSH documentation the following statement is given: *"The periods of the clocks are however dimension-less, they do not refer to any explicit time-scale (e.g. nano-seconds). The reason for the lack of an explicit time-scale is that the CaSH compiler would not be able guarantee that the circuit can run at the specified frequency. The clock periods are just there to indicate relative frequency differences between two different clocks"* [3].

Within the co-simulation, every CλaSH signal is transformed to and from a list of *Int32* values, without any notation of the clock annotations. This has as consequence, that only the usage of one clock domain is supported. Ideas exist to improve the clock (and reset) modelling in future versions of CλaSH [35]. Examples of these ideas are the use of gated clocks, specifying the reset behaviour, and specifying the clock and reset pin names.
Because of the possible updates within CλaSH, different clock-domains will not be supported at this moment. The signals can be synchronized on user-level, with the function *unsafeSynchronizer*, and the support for different clock-domains can be added in a future stage.

```
λ> let x = fromList [0 .. ]
λ> let f = unsafeSynchronizer systemClock clk500
λ> let y = register' clk500 0 $ f x
λ> sampleN 20 x
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
λ> sampleN 20 y
[0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9]
```

Listing 4.48: Conversion of clock-domains with the function *unsafeSynchronizer*

As described in section 4.3, the co-simulation will modify the values in the Verilog simulator once every clock-cycle. The exact moments of the write and read synchronization moments, inside a clock-cycle, depends on the specifications of the clock signal.

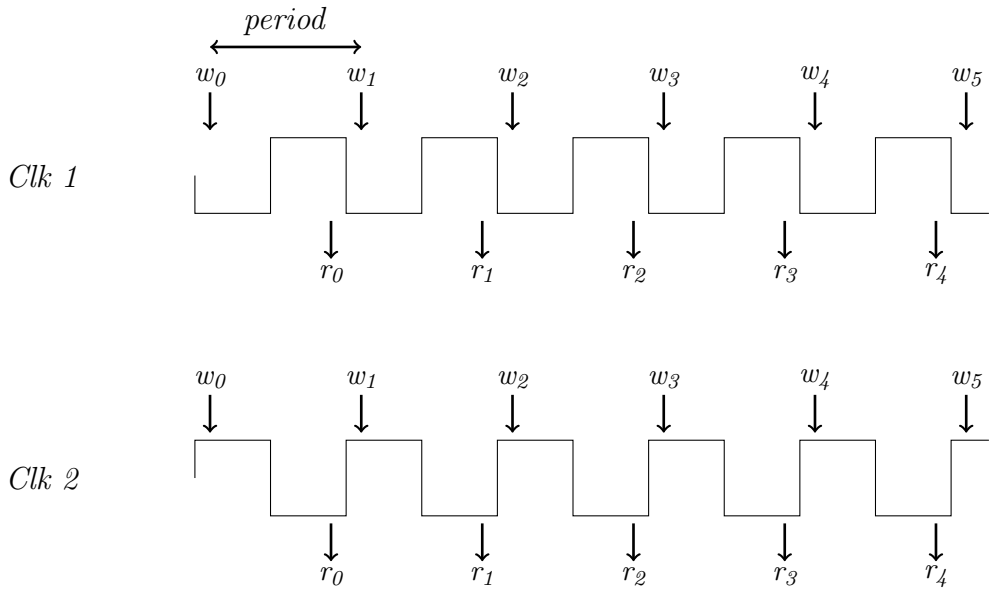


Figure 4.6: Clock signal definitions

An often followed approach within Verilog/VHDL is to use the rising *or* falling edge of a clock to trigger the design. CλaSH generates Verilog code, which uses the *rising* edge of a clock. A co-simulation approach could be to write the input data after the *falling* edge in the Verilog simulator. The data will then be stable before the *rising* edge, after which it can be used in the Verilog circuit. The output data could be read before the next *falling* edge of the clock. This approach would use a clock stream like *Clk 1*, as shown in Figure 4.6.

The FIR filter example, as will be demonstrated in subsection 5.2.2, shows that CλaSH needs a clock signal like *Clk 2*, as shown in Figure 4.6. A circuit often consists of combinational and synchronous sequential parts. The combinational part denotes the computation for one sample inside a CλaSH signal and the rising edge of the clock is used to make the transition to the next sample. The co-simulation will thus write the input data after the *rising* edge and reads the output data before the next *rising* edge.

The number of simulation steps needed between two write moments, called *period*, can be modified in the *CoSimSettings* with the function *coSimSeq*. The same number of simulation steps is used between two read moments. The length of the *period* must be equal to the length of a *clock-cycle* to make the co-simulation stable. The *CoSimSettings* uses 20 simulation steps as default period value. This number is actually chosen randomly; if the clock-cycle also consist of 20 simulation steps, this default period can be used.

Furthermore, a reset phase can be used, which will ignore the first write moment (w_0). By using a reset phase, which can be set with the function *coSimSeq*, the co-simulation will start with a read moment. Starting with a read moment can be useful for circuit with feedback loops.
The FIR filter example, demonstrated in subsection 5.2.2, defines the clock and reset signal in the Verilog code. Furthermore, the function *coSimSeq* is used to disable the reset phase and to define the correct period.

The recommended and most efficient approach is to define the clock and reset signal in Verilog. Listing 4.49 shows how both signals can be defined in the Verilog code. Initial, both signals are set to '1'. The reset goes to '0', for one simulation step, to reset the design at the beginning of the simulation. The clock is negated every five simulation steps to generate the clock cycles.

```
reg clk = 1;
reg rst = 1;

initial begin
    #1 rst = 0;
    #2 rst = 1;
end

always begin
    #5 clk = ~ clk;
end
```

Listing 4.49: A clock and reset signal defined in Verilog

The clock and reset signal can also be defined in CλaSH. To correctly write a clock signal into a Verilog simulator, three times as much write moments has to be scheduled. Furthermore, the input signals has to be tripled to match the *speed* of the clock.
A reset is often performed to reset the design at the beginning of a simulation. Listing 4.49 shows 3 reset transitions at the beginning of the first clock-cycle. To use an equivalent reset signal in CλaSH, 2 extra write moments has to be scheduled in one clock-cycle. After the first clock-cycle, the reset signal will stay the same and the additional read and write moments will cause much overhead.

This master thesis does not show any example in which the clock and reset signals are defined in CλaSH. The first reason is the overhead, as described in the previous paragraph. Furthermore, CλaSH is defined as a functional *Hardware Description Language* without any notion of delays and clock edges. The co-simulation should follow this CλaSH perspective, meaning that clock and reset signals should be defined in the Verilog code.

5 Results

Using the *Verilog Procedural Interface* as described in chapter 3 and the implementation as elaborated in chapter 4, co-simulation between CλaSH and Verilog is made possible. *Icarus Verilog* is used as Verilog-simulator and in this chapter co-simulation examples will be shown. Furthermore, the implementation will be tested with two other simulators: *ModelSim* and *GHDL*.

5.1 Simulators

The *Verilog Procedural Interface* is standardized in the IEEE 1364 and IEEE 1800 standards. The co-simulation is defined with VPI routines which are defined in both the IEEE 1364-2001 and IEEE 1364-2005 standards. Theoretically, the implemented VPI application should have the same behaviour by using other Verilog simulators. The implementation is created with *Icarus Verilog* and in this chapter ModelSim is used as reference simulator to see if the behaviour is indeed the same.

GHDL, an open-source VHDL simulator, does support the VPI. The VPI was supported to use *Icarus Verilog Interactive* (IVI) as graphical developer aid. Currently, the VPI support is extended to make co-simulation with *Cocotb* possible. Using the *Verilog Procedural Interface* with a VHDL simulator would be ideal, because the same interface can be reused to support co-simulation with VHDL.

It is also possible to test the implemented co-simulation with other Verilog simulators, like the *Cadence Incisive Enterprise Simulator* and *Synopsys VCS*. However, the goal of this master thesis is not to support as many Verilog simulators as possible.
Furthermore, many simulators are closed-source and require a license, this is seen as drawback to use these simulators personally.

5.1.1 Icarus Verilog

The implementation, as described in chapter 4, is created to define co-simulation between CλaSH and Icarus Verilog. Icarus Verilog supports all the needed VPI-parts to define co-simulation. Certain properties, like *vpiLowConn* would be desired to more efficiently iterate through the modules and to get the handles to the input and output ports.

Icarus Verilog version 11 (or higher) is needed to support co-simulation, which must be build from source (currently). The current available binaries for Icarus Verilog do not support enough VPI functionality. For example, the callback *cbReadOnlySynch* is not supported. Although there are workarounds, like reading in the *cbReadWriteSynch* callback, these are not desirable. The *cbReadOnlySynch* is defined as the true end of a simulation step; but after a *cbReadWriteSynch*, new simulation events can be scheduled.

Icarus Verilog supports flushing the *Standard Output*, in contrast to ModelSim, what makes debug-statements useful. Furthermore, the *-N* flag can be given to exit the simulator when the simulation is finished. Normally, the simulator will go in interactive mode, what is not desirable for co-simulation.

Icarus Verilog supports simulation with multiple top-modules. For co-simulation this is not desirable. For example, using pre-defined Altera modules (as sub-modules) has as consequence that many input and output ports will be recognized.
Other simulators, like ModelSim, only give support for one top-module and the name of the top-entity has to be given in advance. To use the same implementation for multiple simulators, the design choice is made to only support one Verilog top-entity in a co-simulation.

As described in section 3.2, simulation events and callbacks can occur in different slots. The IEEE 1364 defines the *cbReadWriteSynch* callback in slot 3 and the *cbReadOnlySynch* callback in slot 4. The standard gives freedom on how to implement the internal event scheduling algorithm. By using sufficient simulation steps in one clock-cycle, the consequences of possible differences in scheduling algorithms will be avoided.

Although freedom is given for the scheduling algorithm, the definition of the callbacks should be the same. In this master thesis, the assumption is made that by registering the *cbReadWriteSynch* and *cbReadOnlySynch* callbacks, a *relative* time is used. For example, a *delay* of zero would indicate the current simulation step. In ModelSim this is the case (as expected), but Icarus Verilog uses an absolute time, as discussed on GitHub [69]. A *delay* of zero would then indicate the start of the simulation. However, this concept contradicts by specifying a time smaller than the current simulation time. Icarus Verilog decides to execute callbacks, registered with a time smaller than the current simulation time, in the current simulation step. Registration with a *delay* of 0 is used as *workaround* to define the same behaviour in Icarus Verilog and ModelSim.

5.1.2 ModelSim

ModelSim and *QuastaSim* are both hardware simulation tools from *Mentor Graphics*. ModelSim is primarily targeted at smaller ASIC and FPGA designs. QuastaSim has additional debug capabilities and is targeted at complex FPGA and SoC designs.
To test the co-simulation with CλaSH, the ModelSim-Altera Starter Edition software is used. This version has as limitation that 32 bit library files are required to run ModelSim [50]. This has mainly consequences for compiling and loading the VPI implementation. The VPI sources are compiled to 32-bits object files currently.

ModelSim supports all the needed VPI routines/properties and the same implementation, as used with Icarus Verilog, can be loaded to perform the co-simulation. However, ModelSim outputs much information to the console and this can be a disadvantage when running many (sequential) co-simulations. Command line flags can be used to execute ModelSim *quietly*, but is not possible to hide all the messages.
As described in section 4.4, the functions *coSimEnableStdOut* and *coSimDisableStdOut* can be used to enable or disable the standard output. The standard output is enabled as default setting, what can be useful to show warnings/errors when having syntax errors in the Verilog code. The standard output can be disabled to avoid having any output from the simulator.

5.1.3 GHDL

G Hardware Design Language[1] (GHDL) is an open-source VHDL simulator. Support for the *Verilog Procedural Interface* was created to support *Icarus Verilog Interactive* (IVI), an interactive and graphical front-end for simulating and debugging designs [40]. In GHDL version 0.34, the VPI functionality was extended to make co-simulation with Cocotb possible.

The VPI support is not sufficient currently. Needed VPI properties are missing, mainly for traversing the hierarchy. For example, a *vpiPort* iterator is needed to iterate over all the input and output signals. For every *vpiPort*, the property *vpiDirection* is needed to distinguish between input and output signals. But the *vpiPort* and *vpiDirection* are both unknown properties in GHDL.
Furthermore, it is desirable to use the property *vpiVectorVal* to exchange signals between CλaSH and GHDL. This property is needed to support values containing more than 32 bits, as described in section 3.4.

The owner of GHDL, *tgingold*, has confirmed that VPI properties are missing and he has added the enhancement label [42]. The co-simulation between CλaSH and GHDL can be tested when the missing VPI properties are implemented.

[1] The *G* has currently no meaning.

5.2 Examples and Benchmarks

In this section three co-simulation examples, a multiplier, a FIR filter, and a GFSK demodulator, are demonstrated. The implementations are first executed in CλaSH, after which the FIR filter and GFSK demodulator will be compiled to Verilog. The Verilog code needed for the multiplier will be defined manually to show that both options can be used. The co-simulation with the (generated) Verilog code will be defined and both simulations will be compared.

Besides comparing the outputs, the execution times will be measured. GHCi gives possibilities to measure the execution time in seconds. The multiplier and the FIR filter are small designs, and a smaller time scale is thus desired. The *Criterion* package [65] is used to measure the performance. This package uses a *least square regression* model to estimate the time needed for single a execution.

The needed execution time depends on external factors, like which operating systems is used and what are the hardware specifications. The measurements are performed on an Ubuntu 16.04 system, installed on a virtual machine. The performance indications are only used to compare a CλaSH simulation with the implemented co-simulation, and no conclusions are drawn about the absolute execution times.

5.2.1 Multiplier

The *multiplier* only consists of the elementary mathematical multiplication operation. Two signals will be multiplied together, resulting in one output signal. The multiplication is defined combinational and does not contain any memory elements. However, the output signal is used to register one of the input signals, for which a memory element is needed, as shown in Figure 5.1.

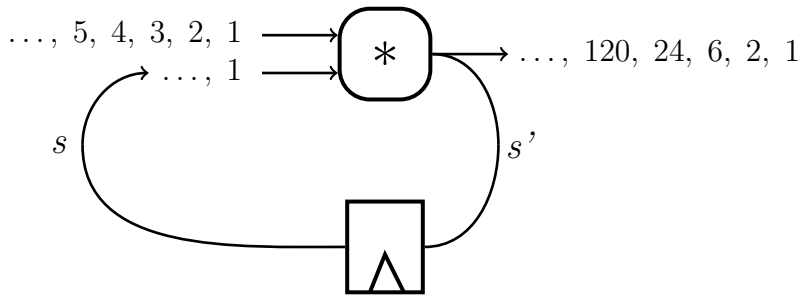


Figure 5.1: The *multiplier* example

The signals consist of 1000-bits signed values. In the CλaSH implementation, the multiplication operator is mapped over these values, as shown in Listing 5.1.

```
mult :: t ~ Signed 1000 => Signal t -> Signal t -> Signal t
mult x y = fmap (uncurry (*)) $ bundle (x, y)
```

Listing 5.1: The multiplier example defined in CλaSH

The two input signals are first combined with the function *bundle*, which creates a *Signal* (t, t). The operator (*) normally has two separate values as input. The function *uncurry* is used to convert this operator to a function on pairs, after which the *uncurried* operator is mapped over the *bundled* signal. Listing 5.2 shows the execution of this CλaSH implementation.

```
λ> let y = mult (fromList [1..]) $ register 1 y
λ> sampleN 13 y
[1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,6227020800]
```

Listing 5.2: The execution of the multiplier example in CλaSH

The Verilog implementation is already used in section 4.1 to show an example of the implemented co-simulation. The continues assignment statement multiplies the two signed input values to define the output value. The Verilog module is embedded with a *QuasiQuoter*, as shown in Listing 5.3.

```
verilog_mult :: t ~ Signed 1000 => Signal t -> Signal t -> Signal t
verilog_mult        = coSim source Icarus "mult"
    where source    = [verilog| module mult (Out, Left, Right);

                                parameter data_width = 1000;
                                input  signed [0:data_width-1] Left;
                                input  signed [0:data_width-1] Right;
                                output signed [0:data_width-1] Out;

                                assign Out = Left * Right;

                            endmodule |]
```

Listing 5.3: The multiplier example defined with co-simulation

The co-simulation is first performed with Icarus Verilog, as shown in Listing 5.4. The same implementation is also executed with ModelSim, as shown in Listing 5.5. Both co-simulations execute correctly and show the same output as the CλaSH implementation. However, using ModelSim has as disadvantage that much information is shown in the console.

```
λ> let y = mult (fromList [1..]) $ register 1 y
λ> sampleN 13 y
[1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,6227020800]
```

Listing 5.4: The execution of the multiplier example using Icarus Verilog

```
** Warning: (vlib-34) Library already exists at "work".
Start time: 23:26:56 on Aug 9,2016
vlog ./clashCoSim-3bqyeZ
Model Technology ModelSim ALTERA vlog 10.4d Compiler 2015.12 Dec 30 2015
-- Compiling module mult

Top level modules: mult
End time: 23:26:56 on Aug 16,2016, Elapsed time: 0:00:00
Errors: 0, Warnings: 0
:Reading pref.tcl

# 10.4d
# vsim -c -quiet -do "onfinish exit; run -all" mult -pli "CoSimVPI.sl"
# Start time: 23:26:57 on Aug 16,2016
# onfinish exit
#  run -all

[1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,6227020800]
# End time: 23:26:57 on Aug 9,2016, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
```

Listing 5.5: The execution of the multiplier example using ModelSim

Table 5.1 shows the needed execution times for the *Multiplier* design. The co-simulation needs approximately the same time as the CλaSH implementation. Furthermore, it is visible that both simulators, Icarus Verilog and ModelSim, executes the design in a comparable amount of time.

No of multiplications	CλaSH	Icarus	ModelSim
1	293.1 ns	307.3 ns	350.2 ns
10	646.8 ns	745.8 ns	717.1 ns
100	3.923 μs	4.195 μs	4.131 μs
1000	38.38 μs	38.03 μs	47.38 μs
10000	360.8 μs	420.5 μs	425.3 μs
100000	3.443 ms	3.543 ms	5.405 ms

Table 5.1: The execution times for simulating the *Multiplier*

5.2.2 FIR filter

The next example is the *FIR* filter as shown on the CλaSH website [3]. The CλaSH implementation is copied in Listing 5.6. The filter works with 16-bit signed values and uses 4 coefficients: '2', '3', '-2', and '8'. A window, with the same size as the coefficients, slides over the input values. The four values inside this window and the 4 coefficients will be multiplied together, after which the outputs are summed.

```
dotp :: SaturatingNum a ⇒ Vec (n + 1) a → Vec (n + 1) a → a
dotp as bs      = fold boundedPlus (zipWith boundedMult as bs)

fir :: (Default a, KnownNat n, SaturatingNum a) ⇒
                              Vec (n + 1) (Signal a) → Signal a → Signal a
fir coeffs x_t  = y_t
     where y_t  = dotp coeffs $ window x_t

topEntity :: Signal (Signed 16) → Signal (Signed 16)
topEntity       = fir (2:>3:>(-2):>8:>Nil)
```

Listing 5.6: The *FIR* filter example defined in CλaSH [3]

The output of the *FIR* filter is directly connected to the input, as visualized in Figure 5.2. The registers are only used as *window* to remember the last 3 inputs values.

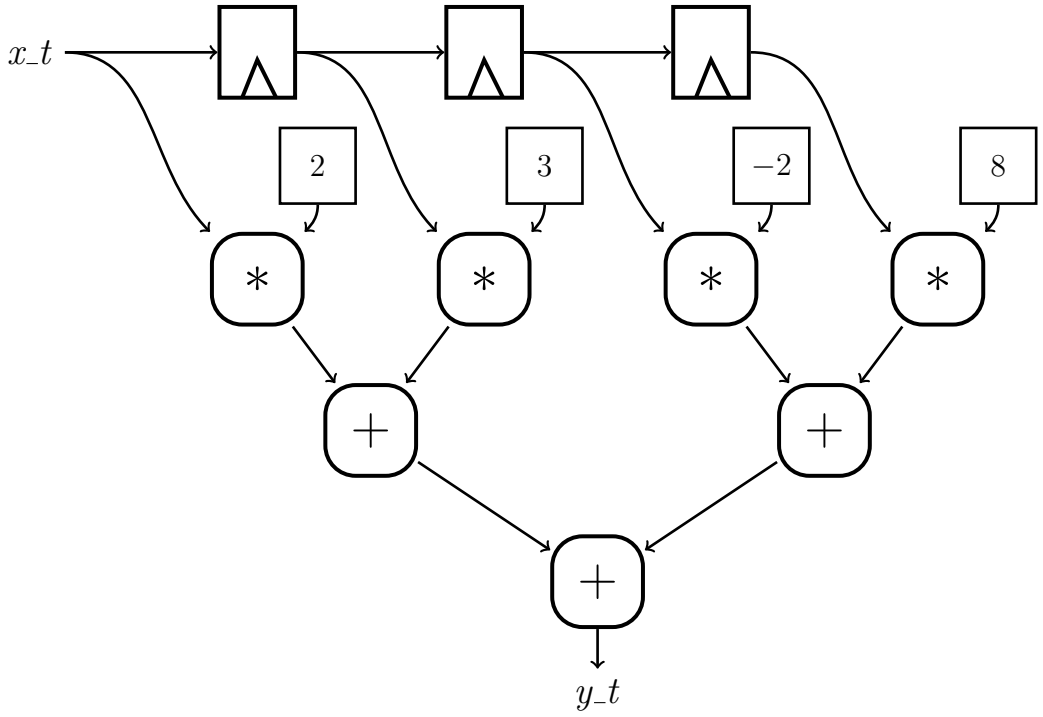


Figure 5.2: The *FIR* filter example

An infinite list, with incrementing values, is used as input signal. Executing the *FIR* filter example with this input signal, gives the output as shown in Listing 5.7.

```
λ> sampleN 13 $ topEntity $ fromList [1..]
[2,7,10,21,32,43,54,65,76,87,98,109,120]
```

Listing 5.7: The execution of the *FIR* filter example in CλaSH

The CλaSH implementation can be compiled to Verilog with the *:Verilog* command. The compiler puts the generated Verilog files in the director *./verilog/Main*. This directory is included with the *coSimSeq* function, as shown in Listing 5.8.
The Verilog implementation uses the *rising* edge of the clock to *remember* the last three values. As explained in section 4.5, the co-simulation will write the input values *after* the *rising* edge and read the output values *before* the *next* rising edge. Furthermore, a reset is needed to set all the registers to zero at the beginning of the co-simulation.

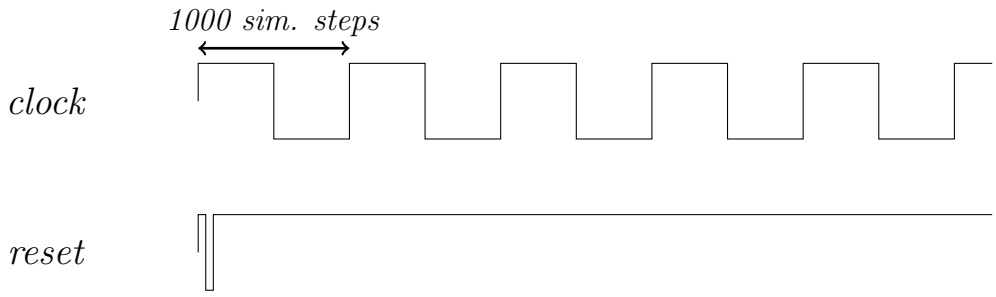


Figure 5.3: The *clock* and *reset* used in the *FIR* filter example (co-simulation)

```
verilog_fir :: t ~ Signed 16 ⇒ Signal t → Signal t
verilog_fir      = coSim source Icarus "fir"
    where source = coSimSeq [verilog| module fir (i,o);

                                      reg clk = 1, rst_n = 1;
                                      parameter data_width = 16;
                                      input  signed [data_width-1:0] i;
                                      output signed [data_width-1:0] o;

                                      initial begin
                                         #1 rst_n = 0;
                                         #2 rst_n = 1;
                                      end
                                      always begin
                                         #500 clk = ~ clk;
                                      end

                                      Main_topEntity_0 dm(i,clk,rst_n,o);

                                endmodule |] (1000,False) ["./verilog/Main"]
```

Listing 5.8: The *FIR* filter example defined with co-simulation

The *FIR* filter, as implemented in Listing 5.8, gives the same output as the CλaSH implementation, as demonstrated in Listing 5.9.

```
λ> sampleN 13 $ verilog_fir $ fromList [1..]
[2,7,10,21,32,43,54,65,76,87,98,109,120]
```

Listing 5.9: The execution of the *FIR* filter example using co-simulation

The execution times are comparable with the previous example, the *Multiplier*, as shown in Table 5.2. This is probably because the simulation itself and the memory management contribute mostly to the needed simulation time. Furthermore, it is visible that the needed simulation time increases almost linear. Around 40 nano-seconds is needed for one clock-cycle, when using a larger number ($\geq$100) of clock-cycles.

No of clock-cycles	CλaSH	Icarus	ModelSim
1	293.5 ns	350.7 ns	310.9 ns
10	615.8 ns	693.1 ns	632.0 ns
100	3.780 μs	4.082 μs	3.972 μs
1000	59.62 μs	38.44 μs	40.46 μs
10000	387.0 μs	391.3 μs	375.1 μs
100000	3.578 ms	4.008 ms	4.086 ms

Table 5.2: The execution times for simulating the *FIR* filter

5.2.3 GFSK demodulator

In the last example, co-simulation will be performed with a *GFSK* demodulator. *Gaussian frequency shift keying* (GFSK) is a modulation method for digital communication. GFSK can be found in many standards like Bluetooth and DECT. Some Background on GFSK Modulation can be found in [70].

Within the course *Implementation of Digital Signal Processing* (191210950), a GFSK demodulator was implemented in Haskell and CλaSH, as a *Design Under Test* (DUT). The demodulator is divided in four parts: a *Mixer*, a *Low-Pass Filter*, a *Delay and Multiply* operator, and a *Slicer*, as visualized in Figure 5.5.
As part of the *Test Vector Controller* (TVC), a GFSK modulator and a channel were implemented in Haskell. Both the DUT and TVC are shown in Figure 5.4.
The most important criteria of the implementation was the *Bit Error Rates* (BERs) for different *Signal-To-Noise ratios* (SNRs) [71][72].

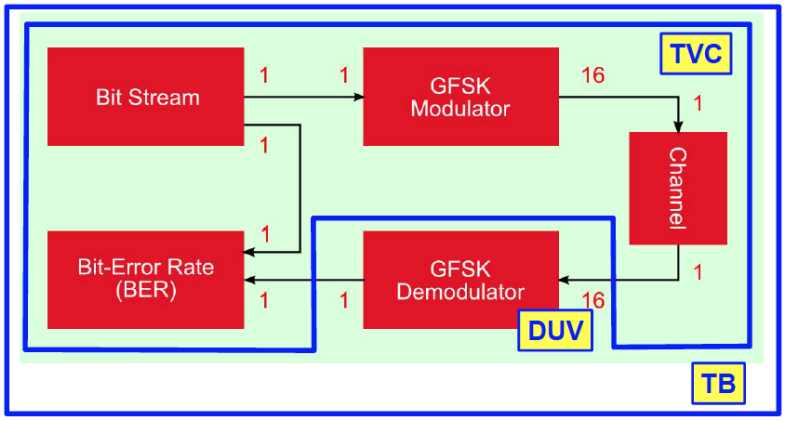


Figure 5.4: The schematic overview of the GFSK design testbench [71]

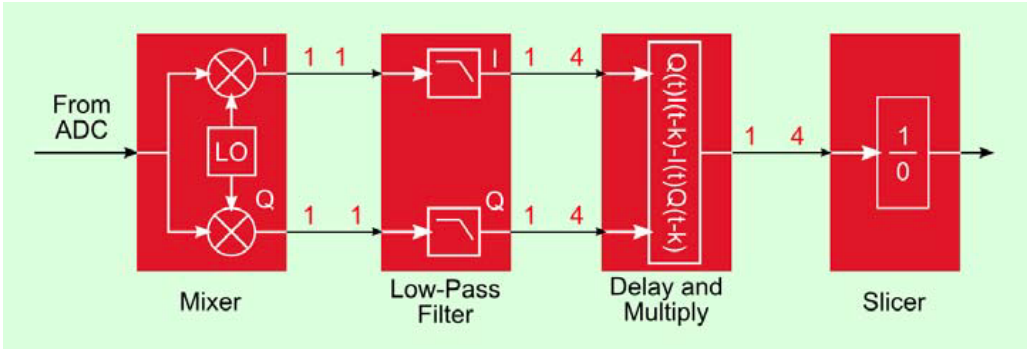


Figure 5.5: The schematic overview of the GFSK demodulator [71]

The first part of the demodulator, the *Mixer*, is used to eliminate the radio frequency. The input signal is multiplied with an unmodulated sine and cosine, to calculate the *in-phase* and *quadrature* components. The sine and cosine are calculated with a *cordic*, in which 8 iterations are performed to increase the accuracy.

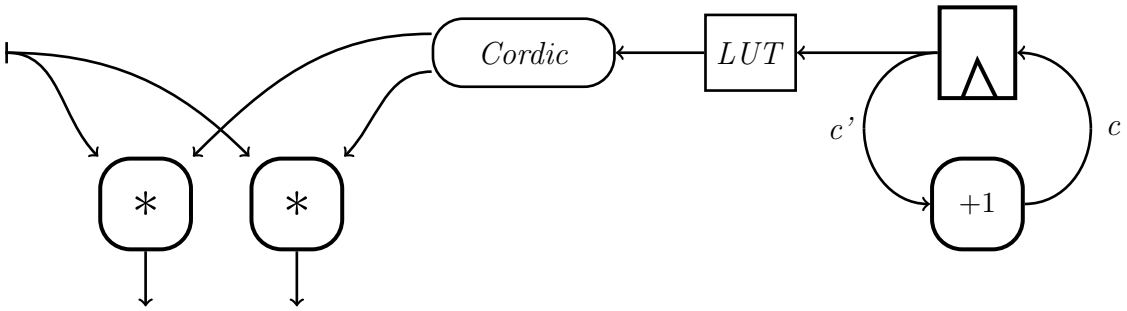


Figure 5.6: The *Mixer*

The second part is an FIR filter, used as a low-pass filter. Two filters are used to filter out any higher frequencies in both the in-phase and quadrature components. Although, more coefficients are used, a simular implementation as the previous example could be defined, as decribed in subsection 5.2.2. However, the FIR filter is defined as a multiplierless design [70][74], to create a more optimized design.

The third part, the *Delay and Multiply* operation, is a classical technique for FM demodulation [70]. Figure 5.7 shows the implementation of this operation.

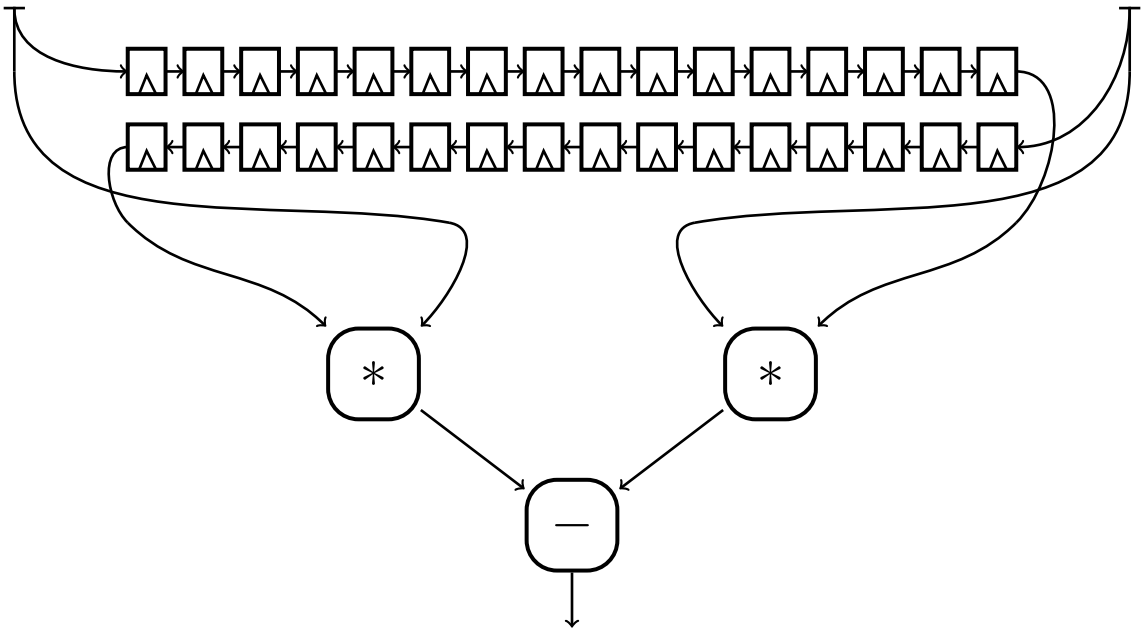


Figure 5.7: The *Delay and Multiply* operation

The *Slicer* is the last part of the GFSK demodulator. Four samples of a symbol are added to make this part more robust in presence of noise. The sign of the sum is returned as a single bit, as shown in Figure 5.8.

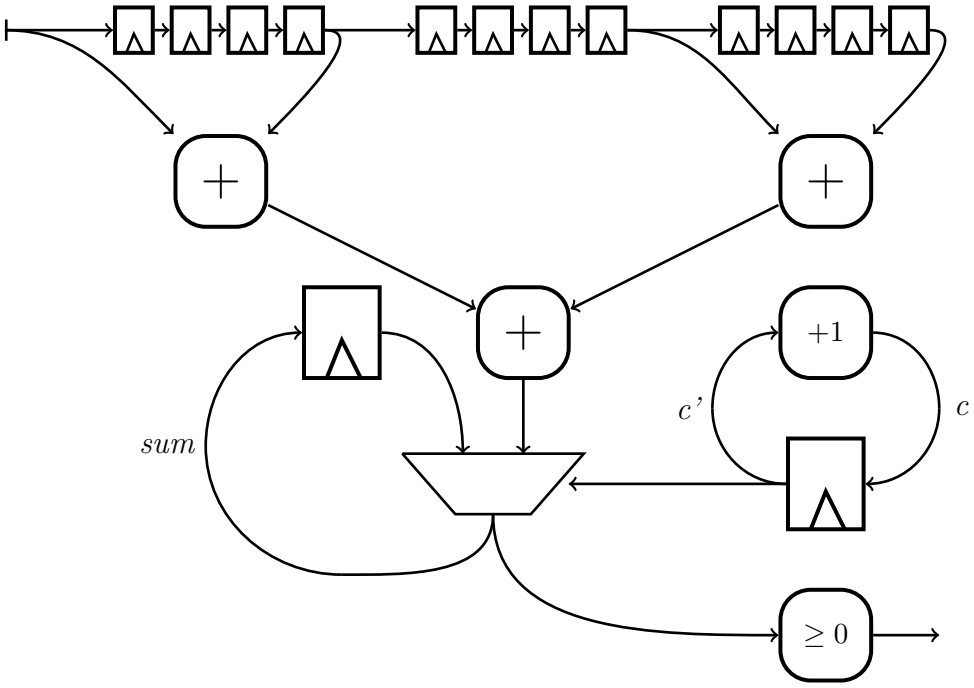


Figure 5.8: The *Slicer*

The goal was not to explain the inner workings of the GSFK implementation, but to show that this example is far more complex compared with the previous two examples. This is also visible in the needed simulation time, as shown in Table 5.3. The down-sampler is not included in the demodulator, because the simulation works with only one clock-domain. This has as consequence that 16 clock-cycles are needed for every output-bit.

No clock-cycles	CλaSH	Icarus	ModelSim
16	28.46 ms	1.404 s	3.891 s
160	127.2 ms	1.844 s	4.651 s
1600	1.475 s	9.194 s	15.99 s
16000	10.48 s	59.11 s	x

Table 5.3: The execution times for simulating the *GFSK* demodulator

In the two previous examples, it was visible that the needed simulation time for the co-simulation was approximately the same compared with the CλaSH implementation. In this example it becomes visible that CλaSH needs far less simulation time.
An explanation could be the lazy evaluation. Although lazy evaluation is used, no clock-cycles will be skipped in the co-simulation. However, the output of the demodulator is down-sampled, what allows the CλaSH implementation to skip certain '*clock-cycles*'.
Another reason is the output type. The demodulator returns a single bit, but the co-simulation works with 32-bit integers.

It was not possible to perform all the simulation with ModelSim. In last test-case an assertion failed, as shown in Listing 5.10, and the Verilog simulator crashes. This is probably due to the fact that *Criterion* does not perform one co-simulation, but a few dozens to estimate the execution time. When a simulation is finished, ModelSim goes to debug mode. This has as consequence that multiple ModelSim processes can run in the background, while performing the benchmarks.

```
__libc_malloc: Assertion `!victim || chunk_is_mmapped (mem2chunk (victim))
   || ar_ptr == arena_for_chunk (mem2chunk (victim))' failed.
```

Listing 5.10: The failed assertion in the *malloc* source

The co-simulation implementation is comparable with the previous co-simulation example. The clock and reset patterns are defined in the Verilog code. In this example, a clock-cycle consist of 64 simulation steps, as shown in Listing 5.11.

```
verilog_gfsk :: t ~ (SFixed 5 3) => Signal' Clk64 t -> Signal' Clk64 Bit
verilog_gfsk      = coSim (coSimDisableStdOut source) Icarus "gfsk"
    where source = coSimSeq [verilog| module gfsk (i,o);

                                      reg clk = 1, rst_n = 1;
                                      input signed [7:0] i;
                                      output o;

                                      initial begin
                                         #1 rst_n = 0;
                                         #2 rst_n = 1;
                                      end
                                      always begin
                                         #32 clk = ~ clk;
                                      end

                                      Main_topEntity_0 dm(i,clk,rst_n,o);

                                endmodule|] (64,False) ["./verilog/Main"]
```

Listing 5.11: The *GFSK* demodulator example defined with co-simulation

The *GFSK* test-bench is shown in Listing 5.12. The input source (*xs*) is modulated (*xs'*), after which *Additive White Gaussian Noise* (AWGN) is added, using an SNR value of 13 (*ys*). Besides adding noise, *Analog to Digital Conversion* (ADC) is simulated inside the channel. The resulting signal is used as input for the demodulator. The output of the demodulator is zipped with the wanted output *xs*, after which a *Bit Error Rate* (BER) test could be applied.
As shown in Listing 5.13, the *GFSK* demodulator is in both implementations be able to recover the original input source.

```
idsp :: ([Double] → [Bit]) → Int →s [(Int,Bit)]
idsp f n          = Prelude.take n $ Prelude.zip xs $ Prelude.drop 5 ys
    where
        ys        = f $ channel xs' awgn 13
        xs        = sourceWanted
        xs'       = signal_modulation 1 10 xs
        awgn      = createNoise sourceNoise

gsfk_tb           = idsp $ fromCLaSH . demodulator . toCLaSH
verilog_gfsk_tb   = idsp $ fromCLaSH . verilog_gfsk . toCLaSH
```

Listing 5.12: The *GFSK* test-bench

```
λ> Prelude.map fst $ gsfk_tb 50
[1,0,0,0,0,0,0,0,0,1,0,0,0,1,1,0,1,0,1,1,1,0,1,1,1
,1,1,1,0,1,0,1,1,0,1,1,0,1,1,0,0,1,0,0,0,1,0,1,1,1]

λ> Prelude.map snd $ gsfk_tb 50
[1,0,0,0,0,0,0,0,0,1,0,0,0,1,1,0,1,0,1,1,1,0,1,1,1
,1,1,1,0,1,0,1,1,0,1,1,0,1,1,0,0,1,0,0,0,1,0,1,1,1]

λ> Prelude.map fst $ verilog_gfsk_tb 50
[1,0,0,0,0,0,0,0,0,1,0,0,0,1,1,0,1,0,1,1,1,0,1,1,1
,1,1,1,0,1,0,1,1,0,1,1,0,1,1,0,0,1,0,0,0,1,0,1,1,1]

λ> Prelude.map snd $ verilog_gfsk_tb 50
[1,0,0,0,0,0,0,0,0,1,0,0,0,1,1,0,1,0,1,1,1,0,1,1,1
,1,1,1,0,1,0,1,1,0,1,1,0,1,1,0,0,1,0,0,0,1,0,1,1,1]
```

Listing 5.13: The execution of the *GFSK* test-benches

6 Recommendations

In chapter 3, the possibilities with *VPI* are explained. With this interface, co-simulation can be made possible between CλaSH and Verilog, as demonstrated in chapter 5. In this chapter, the implemented co-simulation is discussed and recommendations for future work are made.

6.1 IO monad

In section 2.1 it was stated that although Haskell is a *pure* language, *impure* functions can be defined. In the defined co-simulation, the IO-monad is heavily used to control and communicate with the Verilog simulator. Examples of side effects are the creation of files and the execution of processes. On the other hand, by using the same Verilog modules with the same input and specifications, the co-simulation should always return the same output.

Normally *pure* and *impure* can not be combined, but *System.IO.Unsafe* provides a '*back door*' into the IO monad, allowing to perform IO computation at any time [39]. The function *unsafePerformIO* with the type '$IO\ a \rightarrow a$' is used to embed the *impure* in *pure* functions. But this function has to be used carefully, because *unsafePerformIO* is not type-safe and when used inlined, the IO operations can be performed more than once [39].

ByteString, a dependency for CλaSH, also gives warnings for the use of such unsafe back doors: *"If you think you know what you are doing, use 'unsafePerformIO'. If you are sure you know what you are doing, use 'unsafeDupablePerformIO'. If you enjoy sharing an address space with a malevolent agent of chaos, try 'accursedUnutterablePerformIO'"* [38].

Within the function *unsafePerformIO* there is a check that the IO may only be performed by a single thread, which is omitted for the function *unsafeDupablePerformIO*. By omitting this check, there is a possibility that the IO action may be performed multiple times (for example on a multiprocessor) or that one of the duplicated IO actions only runs partially [39].

The examples, as shown in section 5.2, are performed single-threaded. The recommendation is to check the consequences of using the co-simulation in a multi-threaded fashion. Furthermore, the consequences of compiling the Haskell modules to object files, by using the optimize options, are unknown. The assumption is made that GHC will not inline the content used in the *unsafePerformIO* functions, but no research is conducted to validate this statement.

As explained in section 4.2, the function *coSimStart* uses a *mapAccumLM* to perform the simulation steps. This higher order function could be implemented as shown in Listing 6.1. However, this function is not lazy and it builds all the chain in memory until it returns the result.

```
mapAccumLM f s (x:xs) = do
    (s1, x') <- f s x
    (s2, xs') <- mapAccumLM f s1 xs
    return (s2, x' : xs')
```

Listing 6.1: A possible implementation of the *mapAccumLM*

Currently the function *mapAccumLM* is implemented with the *unsafePerformIO* function, as shown in Listing 6.2. Performing IO operations lazy results in unpredictable resource management. Although this is solved by using Haskell's Garbage Collector, the recommendation is to use streams or other data-types to avoid the usage of the *unsafePerformIO* function.

```
mapAccumLM :: (acc -> x -> IO (acc, y)) -> acc -> [x] -> IO (acc, [y])
mapAccumLM f s xs = return $ Data.List.mapAccumL f' s xs
    where f' = \a bs -> unsafePerformIO $ f a bs
```

Listing 6.2: The implemented *mapAccumLM* function

6.2 Co-Simulation with VHDL

The main focus, within this thesis, is about using the VPI to define co-simulation between CλaSH and Verilog. The co-simulation can also be used with SystemVerilog, but normally the VPI does not support co-simulation with VHDL. An obvious recommendation is to support co-simulation with VHDL.
In chapter 5 the implemented co-simulation is tested with GDHL, an open-source VHDL simulator. GHDL uses as foreign interface the VPI, mainly to use the *Icarus Verilog Interactive* (IVI), a graphical developer aid. In the current development version (0.34dev), GHDL has extended the VPI implementation to support co-simulation with *Cocotb*. Unfortunately, not all the needed VPI properties are implemented and thus co-simulation with GHDL is currently not possible.

The implementation at the CλaSH side does not have any connection with VPI properties, and only default C types are passed between CλaSH and the Verilog simulator. Only supporting default C types, gives possibilities for supporting other interfaces, like VHPI. The *VHDL Procedural Interface* is the recommended interface to support co-simulation with VHDL. ModelSim and QuastaSim do not give support for VHPI, but use its own interface: the *Foreign Language Interface* (FLI). Looking at related work, *Cocotb* gives support for both the VHPI and FLI to enable co-simulation with VHDL for a wide range of simulators.

The VHPI has a similar structure as the VPI. In many cases, the VPI routines given in chapter 3, have an equivalent VHPI implementation with comparable names, as shown in Table 6.1.

VPI routines	VHPI routines
vlog_startup_routines	vhpi_startup_routines
vpi_control	vhpi_control
vpi_printf	vhpi_printf
vpi_register_cb	vhpi_register_cb
vpi_remove_cb	vhpi_remove_cb
vpi_handle	vhpi_handle
vpi_iterate	vhpi_iterate
vpi_scan	vhpi_scan
vpi_free_object	vhpi_free_object
vpi_get	vhpi_get
vpi_get_value	vhpi_get_value
vpi_put_value	vhpi_put_value

Table 6.1: Comparable VPI and VHPI routines

Besides equivalent routines, the VHPI also has equivalent callbacks, as shown in Table 6.2, and properties as shown in Table 6.3.

VPI callbacks	VHPI callbacks
cbStartOfSimulation	vhpiCbStartOfSimulation
cbEndOfSimulation	vhpiCbEndOfSimulation
cbNextSimStep	vhpiCbNextSimStep
cbAfterDelay	vhpiCbAfterDelay
cbReadWriteSynch	vhpiCbEndOfProcesses
cbReadOnlySynch	vhpiCbLastKnownDeltaCycle
cbValueChange	vhpiCbValueChange

Table 6.2: Comparable VPI and VHPI callbacks

Although the properties *vpiBinStrVal* and *vphiBinStrVal* have comparable names and are used for the same purpose, they are not exactly the same. Verilog uses 4-state logic values, but VHDL uses 9-state logic values. Besides the '0', '1', 'Z', and 'X' values, VHPI also uses 'U' (uninitialized), 'W' (weak unknown), 'L' (weak 0), 'H' (weak 1), and the *don't care* value.
For exchanging values with Verilog, the *vpiVectorVal* was used as type format, which contains an array of aval/bval pairs. Within VHDL this is not possible and another way must be found to exchange the data efficiently.

VPI properties	VHPI properties
vpiFinish	vhpiFinish
vpiReset	vphiReset
vpiStop	vhpiStop
vpiBinStrVal	vphiBinStrVal
vpiTimeVal	vphiTimeVal

Table 6.3: Comparable VPI and VHPI properties

Besides differences in value exchange, traversing the hierarchy will be different. Although routines as *vhpi_iterate* and *vhpi_scan* are available, a property such as *vpiModule* is not available. Instead of having modules, VHDL uses *entities* and *architectures*. Research has to be done to these parts of the VHPI. The main questions will be: how can traversing through a VHDL hierarchy be implemented? And how to automatically recognize the available signals for co-simulation?

6.3 Multiple clock-domains

the support for *clock signals*, needed for sequential synchronous circuits, is discussed in section 4.5. In the implemented co-simulation only two settings can be given: the length of a clock-cycle and a check if a reset phase is used. The length of a clock cycle is used to read and write values in the correct simulation steps. Normally, the write phase will be performed at the beginning of the period, and the read phase happens at the end of the given period. If a reset phase is used, the first write phase will be skipped and a read phase is the first moment of data exchange.

The recommendation is to define the *clock* and the *reset* in the Verilog code. Both signals can be seen as streams, but they have not the same pattern as the other input and output signals.
CλaSH annotates signals with a clock. These clocks are defined with a certain period and can be used to support multiple clock-domains. Currently, it is only possible to define one period in the co-simulation, and thus only one clock-domain can be used. An extension would be to define the period for every *SignalStream* to support multiple clock domains. The periods should automatically be derived from the clock annotations. The *greatest common divisor* (GCD) of the periods can then be used as the co-simulation speed.

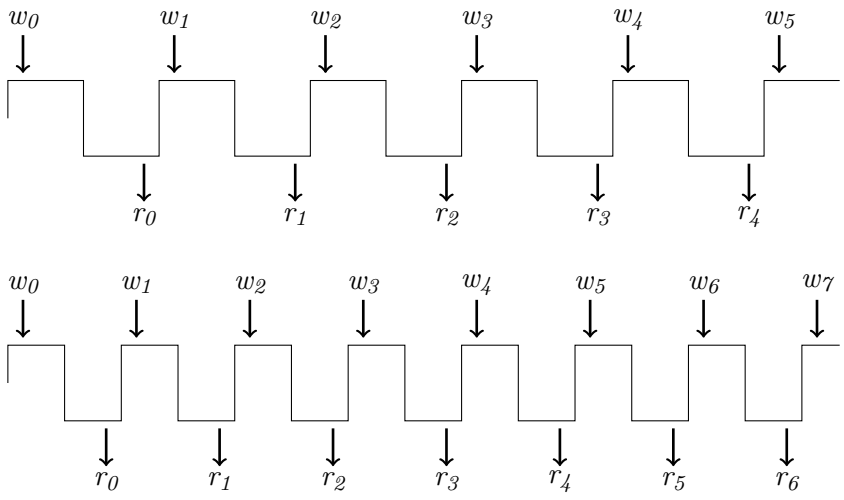


Figure 6.1: Two different clock signals

The clock signals, as shown in Figure 6.1, have not the same period and data exchange will happen in different simulation steps. Depending on the number of steps for both clock signals, the GCD can be calculated to define in which simulation steps data exchange must occur to support both clock-domains.

6.4 Inline Verilog

In subsection 2.1.2 the usage of *Template Haskell* and *QuasiQuotation* is explained. *QuasiQuoters* are used to define a *Domain Specific Language* (DSL). In chapter 4 and chapter 5 *QuasiQuoters* are used to embed Verilog in CλaSH. Currently this Verilog code is not analysed and only forwarded to the Verilog simulator. This has as advantage that CλaSH will not give limitations to the Verilog syntax.

Not analysing the Verilog code also has disadvantages. For example, the Verilog syntax will only be checked when starting the co-simulation and not at compile-time. Furthermore, *AntiQuotation* is not possible. Antiquotation can be used to splice Haskell entities (e.g. variables) in the embbeded Verilog code [51][52][53][55][56].

```
λ> let parseE = returnQ . VarE . mkName
λ> :t parseE
parseE :: String → Q Exp

λ> let x = 3

λ> $(parseE "x")
3
```

Listing 6.3: An *AntiQuotation* example

In the package *Inline-C*, *QuasiQuotation* and *Antiquotation* is used to call C libraries and embed inline C code in Haskell modules [58].

```
-- | 'readAndSum n' reads 'n' numbers from standard input and returns
-- their sum.
readAndSum :: CInt → IO CInt
readAndSum n = [C.block| int {
    // Read and sum n integers
    int i, sum = 0, tmp;
    for (i = 0; i < $(int n); i++) {
      scanf("%d", &tmp);
      sum += tmp;
    }
    return sum;
  } |]
```

Listing 6.4: An Inline-C example

The code between the *Oxford-brackets* is normal C code, except for the $(*int n*) in the for-loop. The Haskell argument *n* for the *readAndSum* function denotes the number of StdIn-reads and is spliced in the C code. The type of *n* is also given in this *AntiQuotation*.

HaskellR is a project bringing together a number of packages for statistical analysis and machine learning using R's comprehensive library support [59]. One of the parts in this project is the package *Inline-R*, used to embed inline R code in Haskell modules.

```
H> let x = 2 :: Double
H> let y = 4 :: Double
H> p [r| x_hs + y_hs |]
[1] 6
```

Listing 6.5: An Inline-R example

Instead of using the $ symbol, the suffix *_hs* is used to splice Haskell values in R [60]. In Listing 6.5, the variables x and y are spliced in R, after which they are added. The Haskell functions can be lifted in the *R Monad* and executed in a *QuasiQuotation*. Listing 6.6 shows the function f, which is used in R.

```
H> let f x = return (x + 1) :: R s Double
H> p [r| f_hs(1) |]
[1] 2
```

Listing 6.6: A spliced Haskell function in R

In order to use *AntiQuotation* for *Inline-Verilog* a parser must be created. A parser for Verilog exists [61], which can probably be reused. Furthermore, a unique keyword has to be defined to splice Haskell values into the DSL. In most literature the $ keyword is used to denote an *AntiQuotation*, but in the Verilog syntax this keyword denotes a foreign function call. A possible solution is use this keyword in combination with parenthesis. Listing 4.1 from chapter 4 can then be rewritten to:

```
verilog_mult :: (t ~ Signed 100) ⇒ Signal t → Signal t → Signal t
verilog_mult x y = coSim Icarus [verilog| assign Out = $(x) * $(y); |]
```

Listing 6.7: A co-simulation example with AntiQuotation

7 Conclusion

High-level *Hardware Description Languages* (HDLs) are becoming popular to increase the abstraction level and design productivity. These high-level HDLs often contain compilers to go to standardized HDLs, like VHDL, Verilog, and SystemVerilog. A desirable part of the high-level HDLs is co-simulation with the standardized HDL(s). Co-simulation gives possibilities for verification and simulating different HDLs within one system.

In chapter 2, background information related to co-simulation is provided. Standards are available to communicate with foreign code. For Verilog, the VPI standard is available, which gives support for data-exchange and to control the simulator. The VHDL standard for co-simulation, VHPI, is very comparable with the VPI and gives almost the same functionality. Drawbacks are the popularity of the VPHI standard, which is not as popular nor supported as the VPI standard. For example, ModelSim uses its own VHDL interface.

Within CλaSH, the FFI can be used to communicate with C. Furthermore, information is provided on how to define a *Domain Specific Language* (DSL) using *Template Haskell* and *QuasiQuotation*, which could be used to embed Verilog code in CλaSH.

Related work is mainly implemented in different programming languages, in which often an imperative design is chosen. Popular implementations are *MyHDL* and *Cocotb*, both written in Python. Within Python it is possible to use generators, and using *yield* the simulation control can be switched. *MyHDL* supports co-simulation with Verilog, using the VPI standard. *Cocotb* provides co-simulation with VHDL, Verilog, and SystemVerilog, using the VPI, VPHI, FLI, and DPI standards.

In chapter 3, the *Verilog Procedural Interface* (VPI), part of the IEEE 1364 Verilog standard, is investigated. Simulation callbacks can be used to synchronize CλaSH to the simulator. These events can be related to the simulator, like start and end of simulation, but also cycle based, like the next simulation step. Values can be exchanged in different type formats, like integer, string and vectors. One of the main differences between a functional HDL, like CλaSH, and a traditional HDL is the support for delay operations and scheduling. With CλaSH , signals can be seen as streams of values, which are simulated using lazy evaluation. In a traditional HDL, the simulation is event-driving using delta-delays to ensure race-free operations.

In chapter 4, co-simulation between CλaSH and Verilog is defined using the VPI standard. The Verilog simulator (Icarus Verilog) is started using the FFI, after which the Verilog code is analyzed. The co-simulation automatically recognizes the Verilog modules and maps CλaSH values to the Verilog ports.

Lazy evaluation is used to perform simulation steps and the problems of lazy IO are avoided by using callbacks into Haskells Garbage Collector. Every simulation step, callbacks are registered in the Verilog simulator in which data exchange occur. Support for both combinational and sequential circuits is provided.

In chapter 5, the implementation as defined in chapter 4, is tested with *ModelSim* and *GHDL*. The same implementation, as defined for Icarus Verilog, can be used with ModelSim. However, the implementation does not work with GHDL, because certain VPI properties are not implemented. On GitHub an issue is created about the missing VPI properites, which is confirmed by the owner of GHDL. When these properties are implemented, co-simulation between CλaSH and VHDL should be possible (theoretical).

Three examples, a multiplier, a FIR filter, and a GFSK demodulator, are tested with co-simulation. These three examples have the same behaviour and output as the CλaSH implementation, when the reset and clock signals were set correctly. The multiplier and the FIR filter needs comparable simulation time compared with the CλaSH implementation, but the GFSK demodulator, a much larger design, was significantly slower.

Finally, in chapter 6, recommendations and ideas for future work are given. The defined co-simulation only uses the VPI standard to support co-simulation with Verilog. To be able to support co-simulation with other HDLs, like VHDL, other standards, like VHPI, have to be supported.

The support for *Inline Verilog* is currently very limited. Only valid Verilog modules are allowed and there is no support for *AntiQuoters*, which can splice Haskell values into the DSL. Furthermore, by using one Verilog module, the name of the top-module can be extracted and forwarded to the Verilog simulator, instead of giving it as parameter.

Currently, there is no support for multiple clock domains. Every signal is used as a stream of values and every value is exchanged in the same simulation step, without any connection to a specific clock. By supporting multiple clock-domains, the speed of the streams will become different, which will have an influence on the synchronization and data exchange between CλaSH and the Verilog simulator.

As formulated in section 1.1, the research addressed in this thesis is:

≫ *How can co-simulation with traditional HDLs be supported within CλaSH?*

This research question is answered by investigating the *Verilog Procedural Interface* (VPI), part of the IEEE 1364 Verilog standard, and the *Foreign Function Interface* (FFI). With the FFI, CλaSH is be able to execute C-functions and manipulate foreign memory. Through the FFI, the VPI can be used to communicate and control Verilog simulators. Data exchange is possible using scheduled callbacks in the Verilog simulator. At the CλaSH side, lazy evaluation is used as scheduling technique and the *Garbage Collector* is used to finish a certain co-simulation.

Personally, I expect that co-simulation will become very useful to simulate CλaSH with existing (Verilog) designs. Looking backwards, I see one other reason: verifying the consequences of the clock-cycles and the reset-phase in a synchronous sequential design. Within CλaSH, the design will be implemented from a functional point of view, without any notion of reset phases and clock edges. The co-simulation can be used to have more insight in reset phases and clock-cycles.

The source-code and the install instruction of the implemented co-simulation are uploaded to GitHub [67]. As prerequisites, CλaSH and Icarus Verilog have to be installed, after which the uploaded Makefile can be used to automatically build and load the implemented co-simulation.

A Higher Order Functions

The image, as shown in Figure A.1, shows the structural representations of the higher order functions *map*, *zipWith*, *fold*, *scanl*, and *mapAccumL*. The image is copied from the course *Embedded Computer Architectures 2* (192130250).

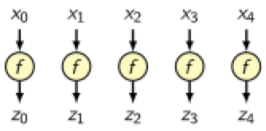
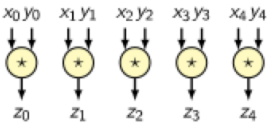
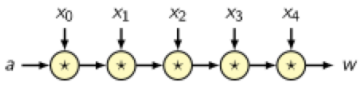
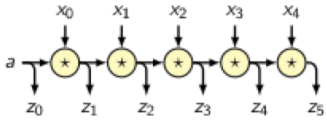
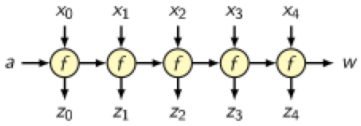
map		$f\ x \Rightarrow z$	$zs = map\ f\ xs$
zipWith		$x\ \star\ y \Rightarrow z$	$zs = zipWith\ (\star)\ xs\ ys$
foldl		$a\ \star\ x \Rightarrow a'$	$w = foldl\ (\star)\ a\ xs$
scanl		$a\ \star\ x \Rightarrow a'$ $z = a$	$zs = scanl\ (\star)\ a\ xs$
mapAccumL		$f\ a\ x \Rightarrow (a', z)$	$(w, zs) =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

Figure A.1: Structural representation of higher-order functions in Haskell

B VPI system function pow

The following examples are copied from [1].

```
void pow_register()
{
  s_vpi_systf_data tf_data;
  tf_data.type        = vpiSysFunc;
  tf_data.sysfunctype = vpiSysFuncReal;
  tf_data.tfname      = "$pow";
  tf_data.calltf      = pow_calltf;
  tf_data.compiletf   = pow_compiletf;
  tf_data.sizetf      = NULL; /*sizetf is not used for real functions*/
  tf_data.user_data   = NULL;
  vpi_register_systf(&tf_data);
}
```

Listing B.1: The registration of the pow-function

```
PLI_INT32 pow_calltf(PLI_BYTE8 *user_data)
{
  s_vpi_value value_s;
  vpiHandle   systf_handle, arg_itr, arg_handle;
  double      base, exp, result;

  value_s.format = vpiRealVal;

  /* obtain handle to system task arguments;
     compiletf has already verified only 2 args with correct types */
  systf_handle = vpi_handle(vpiSysTfCall, NULL);
  arg_itr = vpi_iterate(vpiArgument, systf_handle);

  /* read base value of system function arg 1 & arg 2*/
  arg_handle = vpi_scan(arg_itr);
  vpi_get_value(arg_handle, &value_s);
  base = value_s.value.real;
  arg_handle = vpi_scan(arg_itr);
  vpi_get_value(arg_handle, &value_s);
  exp = value_s.value.real;
  vpi_free_object(arg_itr); /* free iterator--did not scan till null */

  /* calculate result of base to power of exponent */
  result = pow(base,exp);
```

Listing B.2: The calltf routine needed for the pow-function (part A)

```
/* write result to simulation as return value $pow_func */
value_s.value.real = result;
vpi_put_value(systf_handle, &value_s, NULL, vpiNoDelay);
return(0);
```

Listing B.3: The calltf routine needed for the pow-function (part B)

```
PLI_INT32 pow_compiletf(PLI_BYTE8 *user_data)
{
  vpiHandle systf_handle, arg_itr, arg_handle;
  int       tfarg_type;

  systf_handle = vpi_handle(vpiSysTfCall, NULL);
  arg_itr = vpi_iterate(vpiArgument, systf_handle);
  if (arg_itr == NULL) {
    vpi_printf("ERROR: $pow requires 2 arguments\n");
    vpi_control(vpiFinish, 1);  /* abort simulation */
    return(0);
  }
  arg_handle = vpi_scan(arg_itr);
  tfarg_type = vpi_get(vpiType, arg_handle);
  if ( (tfarg_type != vpiReg) &&
       (tfarg_type != vpiIntegerVar) &&
       (tfarg_type != vpiRealVar) &&
       (tfarg_type != vpiConstant)    ) {
    vpi_printf("ERR: $pow arg1 must be number, variable or net\n");
    vpi_control(vpiFinish, 1);  /* abort simulation */
    return(0);
  }
  arg_handle = vpi_scan(arg_itr);
  if (arg_handle == NULL) {
    vpi_printf("ERROR: $pow requires 2nd argument\n");
    vpi_control(vpiFinish, 1);  /* abort simulation */
    return(0);
  }
  tfarg_type = vpi_get(vpiType, arg_handle);
  if ( (tfarg_type != vpiReg) &&
       (tfarg_type != vpiIntegerVar) &&
       (tfarg_type != vpiRealVar) &&
       (tfarg_type != vpiConstant)    ) {
    vpi_printf("ERR: $pow arg2 must be number, variable or net\n");
    vpi_control(vpiFinish, 1);  /* abort simulation */
    return(0);
  }
  if (vpi_scan(arg_itr) != NULL) {
    vpi_printf("ERROR: $pow requires only 2 arguments\n");
    vpi_free_object(arg_itr);
    vpi_control(vpiFinish, 1);  /* abort simulation */
    return(0);
  }
  return(0);  /* no syntax errors detected */
}
```

Listing B.4: The compiletf routine needed for the pow-function

C Marshalling functions

```
foreign import ccall "writeToFile" c_writeToFile :: CString → IO CString

coSimMarshall :: CoSimSettings' → [SignalStream] →
                        IO (Bool, Ptr CInt, CString, Ptr CString)
coSimMarshall settings xs = do

    -- files
    c_f           ← (newCString m) >>= c_writeToFile
    c_fs          ← mapM newCString d
    let c_files   =  c_f : c_fs
    c_filePtrs    ← newArray c_files

    -- topEntity & settings
    c_topEntity   ← newCString top
    c_settingsPtr ← newArray $ Prelude.map fromIntegral $ c_sf c_files

    -- return
    return (c, c_settingsPtr, c_topEntity, c_filePtrs)

    where
        (set,sim,top)       = settings
        (a, b, c, m, d, e)  = set --(hdl, period, rst, data, files, stdOut)
        c_sf fs   = [sim2Num sim, a, b, rst, stdOut, lenf fs, lenf xs]
        lenf      = Prelude.length
        rst       = bool2Num c
        stdOut    = bool2Num e
```

Listing C.1: The function *coSimMarshall*

```
pokeArray' :: Storable a ⇒ Ptr a → [a] → IO ()
pokeArray' ptr [] = return ()
pokeArray' ptr xs
 | ptr == nullPtr = error "null-pointer"
 | otherwise      = pokeArray ptr xs
```

Listing C.2: The function *pokeArray'*

```
peekArray' :: (Storable a, Integral b) ⇒ b → Ptr a → IO [a]
peekArray' (-1) _ = error "null-pointer"
peekArray' size ptr
 | ptr == nullPtr = error "null-pointer"
 | otherwise      = peekArray (fromIntegral size) ptr
```

Listing C.3: The function *peekArray'*

```
foreign import ccall "getInputLength"  c_inputLength  :: Ptr a → IO CInt
foreign import ccall "getOutputLength" c_outputLength :: Ptr a → IO CInt
foreign import ccall "getInputSizes"   c_inputSizes   :: Ptr a →
                                                  IO (Ptr CInt)
foreign import ccall "getOutputSizes"  c_outputSizes  :: Ptr a →
                                                  IO (Ptr CInt)
foreign import ccall "getInputPtr"     c_inputPtr     :: Ptr a →
                                                  IO (Ptr (Ptr CInt))
foreign import ccall "getOutputPtr"    c_outputPtr    :: Ptr a →
                                                  IO (Ptr (Ptr CInt))
```

Listing C.4: Foreign imports

```
coSimInput :: ForeignPtr a → [[Int32]] → IO ()
coSimInput state xs = do

    -- check sizes
    c_iLength     ← withForeignPtr state c_inputLength
    c_iSizes      ← withForeignPtr state c_inputSizes
                                              >>= peekArray' c_iLength
    when (f c_iSizes) $ error $ errStr c_iSizes

    -- write input
    c_inputPtrs   ← withForeignPtr state c_inputPtr
                                              >>= peekArray' c_iLength
    zipWithM_ pokeArray' c_inputPtrs xs'

    where
        xs'       = Prelude.map (Prelude.map fromIntegral) xs
        f         = not . and . Prelude.zipWith compF iSizes
        compF x y = ( x == 0 ) || ( x == y )
        iLength   = fromIntegral $ Prelude.length xs
        iSizes    = Prelude.map (fromIntegral . Prelude.length) xs
        errStr ls = Prelude.concat ["Simulator expects ", show ls,
                          " input words, but ", show iSizes, " given"]
```

Listing C.5: The function *coSimInput*

```
coSimOutput :: ForeignPtr a → IO [[Int32]]
coSimOutput state = do

    -- read output
    c_oLength     ← withForeignPtr state c_outputLength
    c_oSizes      ← withForeignPtr state c_outputSizes
                                              >>= peekArray' c_oLength
    c_outputPtrs  ← withForeignPtr state c_outputPtr
                                              >>= peekArray' c_oLength
    ys            ← zipWithM peekArray' c_oSizes c_outputPtrs

    -- convert and return
    return $ Prelude.map (Prelude.map fromIntegral) ys
```

Listing C.6: The function *coSimOutput*

D Installation Simulators

In this thesis three simulators are used: *Icarus Verilog*, *GHDL*, and *ModelSim*. *Ubuntu 16.04*, executed as a virtual system on a Mac OS X (*El Capitan*) system, is used as operation system.

D.1 Icarus Verilog

Icarus Verilog version 11 (or higher) is needed to have support for co-simulation. Binaries are only available for version 9 & 10 (currently), and thus the latest version is pulled from GitHub. Furthermore, the following dependencies must be installed: *bison*, *flex*, *gperf*, *autoconf*, *g++*, and *build-essential*. *Icarus Verilog* can then be installed with the *configure* and *make* commands.

```
#!/bin/bash
sudo apt-get install bison
sudo apt-get install flex
sudo apt-get install gperf
sudo apt-get install autoconf
sudo apt-get install g++
sudo apt-get install build-essential

# Icarus Verilog
git clone https://github.com/steveicarus/iverilog
cd iverilog
autoreconf -i
./configure
make
sudo make install
```

Listing D.1: Installing Icarus Verilog

D.2 GHDL

In GHDL version 0.34, the *VPI* implementation is extended to support co-simulation with *Cocotb*. But for CλaSH this is not yet sufficient. The install instruction for version 0.34 (dev) are given and hopefully co-simulation is possible soon.

GHDL is available with different back-ends (code-generators): *GCC*, *mcode*, and *llvm*. The *mcode* back-end cannot be used to support co-simulation, because no object file will be generated. In this case, *llvm* will be used, because it is easier to build then with *GCC*. The following dependencies must be installed: *GNAT* (GNU Ada compiler), *zlib*, and *llvm 3.5*. *GHDL* can be installed in the same way as *Icarus Verilog* (git clone and the make commands).

```
#!/bin/bash
sudo apt-get install zlib1g-dev
sudo apt-get install llvm-3.5
sudo apt-get install clang
sudo apt-get install libedit-dev

# GNAT GPL 2016
wget http://mirrors.cdn.adacore.com/art/5739cefdc7a447658e0b016b
tar xf 5739cefdc7a447658e0b016b
cd gnat-gpl-2016-x86_64-linux-bin/
sudo ./doinstall
PATH="/usr/gnat/bin:$PATH"; export PATH

# GHDL
git clone https://github.com/tgingold/ghdl
cd ghdl
./configure --with-llvm-config=/usr/bin/llvm-config-3.5
make
sudo "PATH=$PATH" make install
```

Listing D.2: Installing GHDL

D.3 ModelSim

ModelSim is available in different versions, like ModelSim PE, ModelSim SE, and ModelSim XE. In this case the *ModelSim-Altera Starter Edition* is used [62]. This version only includes the base features of ModelSim PE, including behavioral simulation, HDL testbenches, and Tcl scripting. Optional features, like *VHDL plus Verilog Dual language* and *SystemC 2.2* are not supported. Furthermore, the simulation performance is slower and only 10.000 executable lines can be used.

The *ModelSim-Altera Starter Edition* is only available as 32-bit version. On a 64 bit Linux version, the 32 bit dependencies must be installed, in order to install this ModelSim version [63]. The setup procedure of ModelSim can be used to install this simulator.

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install build-essential
sudo apt-get install gcc-multilib g++-multilib \
lib32z1 lib32stdc++6 lib32gcc1 \
expat:i386 fontconfig:i386 libfreetype6:i386 libexpat1:i386 libc6:i386
    libgtk-3-0:i386 \
libcanberra0:i386 libpng12-0:i386 libice6:i386 libsm6:i386 libncurses5:i386
     zlib1g:i386 \
libx11-6:i386 libxau6:i386 libxdmcp6:i386 libxext6:i386 libxft2:i386
    libxrender1:i386 \
libxt6:i386 libxtst6:i386

wget http://download.altera.com/akdlm/software/acdsinst/16.0/211/
    ib_installers/ModelSimSetup-16.0.0.211-linux.run
chmod +x ModelSimSetup-16.0.0.211-linux.run
sudo ./ModelSimSetup-16.0.0.211-linux.run
```

Listing D.3: Installing ModelSim

If Linux version *3.x.x.x* is used, *vsim* will search in a non-existing directory for the needed libraries. Furthermore, the *MTI_VCO_MODE* environment variable is used to force selection of 32-bit or 64-bit platform directories for executables. Both settings can be fixed in the *vsim* file [64].

```
vim $HOME/altera/16.0/modelsim_ase/bin/vsim

mode=${MTI_VCO_MODE:-""}                    # line 13
mode=${MTI_VCO_MODE:-"32"}                  # has to be changed to

3.[0-9]*)                   vco="linux_rh60" ;; # line 209
3.[0-9]*)                   vco="linux" ;;      # has to be change to
```

Listing D.4: Vsim settings

Bibliography

[1] Stuart Sutherland. *The Verilog PLI Handbook. [A User's Guide and Comprehensive Reference on the Verilog Programming Language Interface]*
Springer Science+Business Media, LLC, 2002

[2] Swapnajit Mittra *Principles of Verilog PLI*
Springer Science+Business Media, LLC, 1999

[3] CλaSH: From Haskell to Hardware
/http://www.clash-lang.org
Accessed: January 2016

[4] IEEE Standard for Verilog Hardware Description Language (1364-2001)
/http://ieeexplore.ieee.org/document/954909
Accessed: May 2016

[5] IEEE Standard for Verilog Hardware Description Language (1364-2005)
/http://ieeexplore.ieee.org/document/1620780
Accessed: May 2016

[6] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language (1800-2005)
/http://ieeexplore.ieee.org/document/1560791
Accessed: May 2016

[7] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language (1800-2009)
/http://ieeexplore.ieee.org/document/5354441
Accessed: May 2016

[8] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language (1800-2012)
/http://ieeexplore.ieee.org/document/6469140
Accessed: May 2016

[9] IEEE Standard VHDL Language Reference Manual (1076-2002)
/http://ieeexplore.ieee.org/document/1003477
Accessed: May 2016

[10] IEEE Standard VHDL Language Reference Manual (1076-2008)
/http://ieeexplore.ieee.org/document/4772740
Accessed: May 2016

[11] VHPI Applications
/https://www.aldec.com/en/support/resources/documentation/articles/1457
Accessed: May 2016

[12] The process package; contains libraries for dealing with system processes.
/https://hackage.haskell.org/package/process
Accessed: July 2016

[13] The Foreign Function Interface
/https://wiki.haskell.org/GHC/FAQ#The_Foreign_Function_Interface
Accessed: April 2016

[14] Foreign Function Interface
/https://wiki.haskell.org/Foreign_Function_Interface
Accessed: April 2016

[15] Make GHCi load and interpret a module with a foreign export declaration
/http://stackoverflow.com/questions/28899620/make-ghci-load-and-interpret-a-module-with-a-foreign-export-declaration-for-f
Accessed: April 2016

[16] The tf_nodeinfo has been deprecated by IEEE
/http://stackoverflow.com/questions/30381195/tf-nodeinfo-has-been-deprecated-by-ieee
Accessed: May 2016

[17] ModelSim Foreign Language Interface
/http://homepages.cae.wisc.edu/~ece554/new_website/ToolDoc/Modelsim_docs/docs/pdf/fli.pdf
Accessed: May 2016

[18] Using ModelSim Foreign Language Interface
for c VHDL Co-Simulation and for Simulator Control on Linux x86 Platform
/http://opencores.org/usercontent,doc,1380917197
Accessed: May 2016

[19] SystemVerilog DPI Tutorial - SystemVerilog Layer
/http://www.project-veripage.com/dpi_tutorial_1.php
Accessed: May 2016

[20] The Verilog PLI Is Dead (maybe) Long Live The SystemVerilog DPI!
/http://sutherland-hdl.com/papers/2004-SNUG-paper_Verilog_PLI_versus_SystemVerilog_DPI.pdf
Accessed: May 2016

[21] Critical Components for Smart Grid Ready Instrumentation
/http://www.ni.com/white-paper/14529/en/
Accessed: February 2016

[22] What is GPU computing?
/http://www.nvidia.com/object/what-is-gpu-computing.html
Accessed: February 2016

[23] A Survey of Methods For Analyzing and Improving GPU Energy Efficiency
/https://arxiv.org/pdf/1404.4629.pdf
Accessed: February 2016

[24] Optimizing Application Energy Efficiency Using CPUs, GPUs and FPGAs
/http://www.scientificcomputing.com/article/2015/03/optimizing-application-energy-efficiency-using-cpus-gpus-and-fpgas
Accessed: February 2016

[25] Using CLaSH with an existing codebase
/https://groups.google.com/forum/#!topic/clash-language/rrDEKE_r2bk
Accessed: January 2016

[26] You Could Have Invented Monads! (And Maybe You Already Have.)
/http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html
Accessed: February 2016

[27] Control.Monad: Functor and monad classes
/https://hackage.haskell.org/package/base-4.8.0.0/docs/Control-Monad.html
Accessed: February 2016

[28] IO inside
/https://wiki.haskell.org/IO_inside
Accessed: February 2016

[29] Why It's Nice to be Quoted: Quasiquoting for Haskell
/http://www.cs.tufts.edu/comp/150FP/archive/geoff-mainland/quasiquoting.pdf
Accessed: July 2016

[30] The Haskell 2010 Language Report
/https://www.haskell.org/onlinereport/haskell2010/
Accessed: February 2016

[31] Non-strict semantics
/https://wiki.haskell.org/Non-strict_semantics
Accessed: March 2016

[32] Real World Haskell. Interfacing with C: the FFI
/http://book.realworldhaskell.org/read/interfacing-with-c-the-ffi.html
Accessed: February 2016

[33] VHPI, A programming Language Interface for VHDL
/http://www.eda.org/VIUF_proc/Fall96/DUNLOP96A.PDF
Accessed: May 2016

[34] Manifesto
/http://verificationhack.com/2011/05/11/manifesto/
Accessed: January 2016

[35] Improved clock-and-reset-modelling in CLaSH
/https://github.com/clash-lang/clash-prelude/wiki/Clock-and-reset-modelling
Accessed: June 2016

[36] Strongly Typed Heterogeneous Collections
/http://okmij.org/ftp/Haskell/HList-ext.pdf
Accessed: April 2016

[37] Currying
/https://wiki.haskell.org/Currying
Accessed: January 2016

[38] Source-code for the ByteString package
/https://github.com/haskell/bytestring/blob/master/Data/ByteString/Internal.hs
Accessed: July 2016

[39] Unsafe IO operations
/https://hackage.haskell.org/package/base-4.9.0.0/docs/System-IO-Unsafe.html
Accessed: June 2016

[40] GHDL Features
/http://ghdl.free.fr/site/pmwiki.php?n=Main.Features
Accessed: July 2016

[41] GHDL commands
/http://home.gna.org/ghdl/ghdl/Building-commands.html#Building-commands
Accessed: July 2016

[42] GitHub issue: vpiPorts vpiVectorVal
/https://github.com/tgingold/ghdl/issues/101
Accessed: July 2016

[43] HDL Cosimulation - Cosimulate hardware component
by communicating with HDL module instance executing in HDL simulator
/http://nl.mathworks.com/help/hdlverifier/ref/hdlcosimulation.html
Accessed: April 2016

[44] What is PyHVL?
/http://pyhvl.sourceforge.net/index.php
Accessed: March 2016

[45] Ruby-VPI 21.1.0 user guide
/http://snk.tuxfamily.org/lib/ruby-vpi
Accessed: March 2016

[46] What is Jove?
/http://jove.sourceforge.net
Accessed: March 2016

[47] The MyHDL manual
/http://docs.myhdl.org/en/stable/manual
Accessed: March 2016

[48] Co-simulation with Verilog
/http://docs.myhdl.org/en/stable/manual/cosimulation.html
Accessed: March 2016

[49] Welcome to Cocotbs documentation!
/http://cocotb.readthedocs.io
Accessed: March 2016

[50] ModelSim OS-support
/https://www.altera.com/support/support-resources/download/os-support.html
Accessed: July 2016

[51] School of Haskell - Quasiquotation 101
/https://www.schoolofhaskell.com/user/marcin/quasiquotation-101
Accessed: July 2016

[52] edsko.net - Brief Intro to Quasi-Quotation
/http://edsko.net/2013/05/09/brief-intro-to-quasi-quotation/
Accessed: June 2016

[53] Language.Haskell.TH.Syntax - Abstract syntax definitions for Template Haskell.
/https://hackage.haskell.org/package/template-haskell-2.11.0.0/docs/Language-Haskell-TH-Syntax.html
Accessed: July 2016

[54] Language.Haskell.TH.Quote - QuasiQuotation
/https://hackage.haskell.org/package/template-haskell-2.11.0.0/docs/Language-Haskell-TH-Quote.html
Accessed: July 2016

[55] Quasiquotation
/https://wiki.haskell.org/Quasiquotation
Accessed: July 2016

[56] A practical Template Haskell Tutorial
/https://wiki.haskell.org/A_practical_Template_Haskell_Tutorial
Accessed: July 2016

[57] Template Haskell
/https://downloads.haskell.org/~ghc/7.10.3/docs/html/users_guide/template-haskell.html
Accessed: July 2016

[58] Inline-c: Write Haskell source files including C code inline
/https://hackage.haskell.org/package/inline-c
Accessed: April 2016

[59] HaskellR - Programming R in Haskell
/http://tweag.github.io/HaskellR/
Accessed: July 2016

[60] Splicing Haskell values in R
/http://tweag.github.io/HaskellR/docs/splicing-haskell-values-in-r.html
Accessed: July 2016

[61] The verilog package - A parser and supporting a small subset of Verilog. Intended for machine generated, synthesizable code
/https://hackage.haskell.org/package/verilog
Accessed: July 2016

[62] ModelSim-Altera Software Products
/https://www.altera.com/products/design-software/model---simulation/modelsim-altera-software.html
Accessed: April 2016

[63] Making ModelSim ALTERA STARTER EDITION work on Ubuntu 14.04
/http://mattaw.blogspot.nl/2014/05/making-modelsim-altera-starter-edition.html
Accessed: July 2016

[64] Modelsim is not working after ubuntu 11.10 upgrade?
/http://askubuntu.com/questions/68146/modelsim-is-not-working-after-ubuntu-11-10-upgrade
Accessed: July 2016

[65] A criterion tutorial - Learn how to write Haskell microbenchmarks.
/http://www.serpentine.com/criterion/tutorial.html
Accessed: July 2016

[66] Foreign ForeignPtr : Finalised data pointers
/https://hackage.haskell.org/package/base-4.9.0.0/docs/Foreign-ForeignPtr.html
Accessed: June 2016

[67] Co-simulation between CaSH and standardized HDLs
/https://github.com/jgjverheij/clash-cosim
Accessed: August 2016

[68] GitHub: Icarus Verilog
/https://github.com/steveicarus/iverilog
Accessed: March 2016

[69] VPI - delays ignored when registering callbacks
/https://github.com/steveicarus/iverilog/issues/117
Accessed: August 2016

[70] Some Background on GFSK Modulation
/http://wwwhome.ewi.utwente.nl/~gerezsh/sendfile/sendfile.php/gfsk-intro.pdf?sendfile=gfsk-intro.pdf
Accessed: July 2016

[71] The Polyphase Implementation of FIR Filters
/http://wwwhome.ewi.utwente.nl/~gerezsh/sendfile/sendfile.php/idsp-poly.pdf?sendfile=idsp-poly.pdf
Accessed: July 2016

[72] Project GFS: The GFSK Receiver
/http://wwwhome.ewi.utwente.nl/~gerezsh/vlsidsp/projects/gfs15.html
Accessed: July 2016

[73] The cordic algorithm and cordic architectures
/http://wwwhome.ewi.utwente.nl/~gerezsh/sendfile/sendfile.php/idsp-cordic.pdf?sendfile=idsp-cordic.pdf
Accessed: July 2016

[74] Multiplierless filter design
/http://wwwhome.ewi.utwente.nl/~gerezsh/sendfile/sendfile.php/idsp-nomult.pdf?sendfile=idsp-nomult.pdf
Accessed: July 2016