A Tutorial on CλaSH: From Haskell to Hardware

by

Christiaan P.R. Baaij

Email:
c.p.r.baaij@utwente.nl

Group: Computer Architecture for Embedded Systems
Faculty of Electrical Engineering, Mathematics, and Computer Science
University of Twente, Enschede, The Netherlands

November 25, 2010

Abstract

CλaSH is a functional hardware description language that borrows both its syntax and semantics from the functional programming language Haskell. Polymorphism and higher-order functions provide a level of abstraction and generality that allow a circuit designer to describe circuits in a more natural way than possible with the language elements found in the traditional hardware description languages.

Circuit descriptions can be translated to synthesizable VHDL using the prototype CλaSH compiler. As the circuit descriptions, simulation code, and test input are also valid Haskell, complete simulations can be done by a Haskell compiler or interpreter, allowing high-speed simulation and analysis.

Table of Contents

List of Acronyms

ASIC Application-Specific Integrated Circuit

CAES Computer Architecture for Embedded Systems

CλaSH CAES language for synchronous hardware

FPGA Field-Programmable Gate Array

GHC Glasgow Haskell Compiler

HDL Hardware description language

VHDL VHSIC HDL

VHSIC Very High Speed Integrated Circuit

Chapter 1

Introduction

Hardware description languages (HDLs) have not allowed the productivity of hardware engineers to keep pace with the development of chip technology. While traditional HDLs, like VHSIC HDL (VHDL) and Verilog, are very good at describing detailed hardware properties such as timing behavior, they are generally cumbersome in expressing the higher-level abstractions needed for today's large and complex circuit designs. In an attempt to raise the abstraction level of the descriptions, a great number of approaches based on functional languages have been proposed.

CAES language for synchronous hardware (CλaSH) is an experimental tool for designing and simulating hardware. With CλaSH you can describe circuits in a simple and concise functional hardware description language. We can simulate circuits like the standard HDLs, but also generate structural VHDL descriptions from the CλaSH circuit descriptions. The aim of this tutorial is to introduce this new style of functional circuit design by means of examples.

CλaSH is really a set of two libraries written in Haskell, one library with functions and datatypes that can be simulated, and a second library that can compile circuit descriptions written in Haskell to VHDL. While the circuit descriptions are written in Haskell, descriptions should not be considered as Haskell programs, but really as declarative specifications of hardware. The generated VHDL circuit descriptions are synthesizable, meaning that industry standard tooling can process these description to either configure an Field-Programmable Gate Array (FPGA) or make a silicon layout for an Application-Specific Integrated Circuit (ASIC).

This tutorial introduces the style of circuit descriptions used in CλaSH, by means of very simple examples. The tutorial is (loosely) based on the excellent tutorial for the Chalmers Lava system. Later examples emphasize how higher-order functions, polymorphism, and partial application lead to concise and clear circuit descriptions. The tutorial shows the available interpretations of circuits: simulation, and the generation of VHDL code. After the tutorial you should be able to describe and analyze simple circuits using CλaSH. The quick reference at the end of this tutorial will hopefully get you started with future endeavors.

Chapter 2

Getting Started

This chapter briefly discusses the installation of CλaSH, shows how to describe a simple circuit and simulate them in the interpreter, and translate the circuit description to VHDL.

2.1 Installation

We only discuss the installation of the CλaSH compiler and interpreter in this section; usage will be discussed throughout the rest of the tutorial. You should download and install a version of the Haskell Platform (`http://hackage.haskell.org/platform/`) that contains Glasgow Haskell Compiler (GHC) version 6.12 (e.g. Haskell Platform release 2010.2.0.0). After installation of the Haskell Platform you should also make sure that the latest package list from `http://hackage.haskell.org` is known to the `cabal` package manager by running:

```
$ cabal update
```

Once installed, a you can download the `clash-bin-0.1.1.tar.gz` from the CλaSH website (`http://clash.ewi.utwente.nl`). Once this tar-ball is unpacked, you should run the following command inside the created directory:

```
$ cabal install
```

After `cabal` has finished installing the CλaSH binary, you should be able to run the CλaSH interpreter from the terminal by running:

```
$ clash --interactive
```

2.2 Your First Circuit

To make your first circuit description, start up your favorite text editor and create a text file called `First.hs`. CλaSH file names have the extension `.hs`.

We are going to define a so-called *half adder* (Figure 2.1). A half adder is a

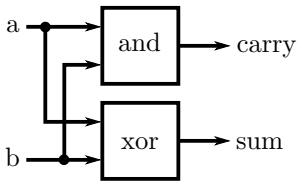

Figure 2.1: Half adder circuit

component that is for example used in the implementation of a binary adder. It takes two bits as input, and adds them. The result is a *sum* and a *carry* bit. A half adder is usually realized using one `and` and one `xor` gate. Here is how we define a half adder, *halfAdd*, in CλaSH.

```
module First where

import CLasH.HardwareTypes

halfAdd a b = (sum, carry)
  where
    sum   = hwxor a b
    carry = hwand a b
```

We import a module called `CLaSH.HardwareTypes`, which defines a number of operations that we can use to build circuits. Notably it contains the gates `hwxor` and `and2`. Appendix A contains a list of such predefined operations.

Note that the order of the definitions in the `where`-clause does not matter! Since these circuit components act in parallel, we could just as well have put them the other way around.

2.3 The CλaSH Interpreter

During the development of a collection of circuits, we mainly use the CλaSH *interpreter*. It is actually the GHC interpreter extended with the CλaSH library[1].

The installation section already discussed how we can start the CλaSH interpreter. Once the CλaSH interpreter is started you should see the following:

```
GHCi + CLaSH (:vhdl command), version 6.12.1: http://www.haske...
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

`Prelude>` is the prompt of the interpreter, the part before the `>` symbol shows the currently loaded module. In this case it is the default module, called *Prelude*. We can use the interpreter to load different modules with circuit definitions and to type in commands that we want to execute. If the circuit descriptions are not in the directory in which you started the interpreter you can change to the working directory using the `:cd <path>` command (Tab-completion should work).

```
Prelude> :cd ~/Documents/CλaSH/
```

If we type in the half adder definition in the file `First.hs`, we can than load the design into the interpreter using the `:l` command:

```
Prelude> :l First.hs
[1 of 1] Compiling First            ( First.hs, interpreted )
```

[1] Users familiar with Haskell should read the accompanied Haddock information if they want to use CλaSH as a library.

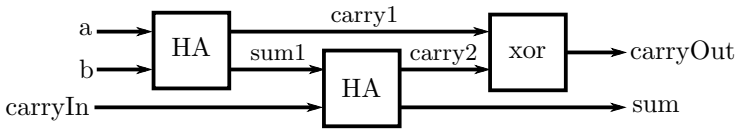

Figure 2.2: Full adder circuit

```
... Loading many support modules ...

Ok, modules loaded: First.
*First>
```

The prompt now starts with `First`, indicating that the design is indeed loaded by the interpreter. One of the things we can do with a circuit is to simulate it. Simulation of purely combinational circuits, such as `halfAdd`, is simple done by calling the circuit definition and supply the input arguments (in this case two bits).

```
*First> halfAdd Low Low
(Low,Low)
*First> halfAdd High High
(Low,High)
```

If we make any changes to the file with our circuit definitions, we can type the reload command `:r` in the interpreter:

```
*First> :r
...
*First>
```

The changers are now updated. If you ever want to exit from the interpreter, you can use the `:q` command.

```
*First> :q
Leaving GHCi.
$
```

2.4 Your Second Circuit

Indeed, you guessed it: your second circuit is going to be a *full adder* (Figure 2.2), a component `fullAdd` that consists of two half adders. To define it, add the following definition to the file `First.hs`:

```
fullAdd carryIn (a, b) = (sum, carryOut)
  where
    (sum1, carry1) = halfAdd a b
    (sum, carry2)  = halfAdd carryIn sum1
    carryOut       = hwxor carry2 carry1
```

Note that unlike the half adder, the second input of this circuit consists of a pair of bits.

We transcribe the diagram of the circuit (Figure 2.2) by giving names to all the internal signals (here `sum1`, `carry1`, and `carry2`) and then simply writing down all the sub-parts of the circuit. To ease this process, we have decided to read the inputs to

a sub-component from bottom to top. The order of the resulting equations doesn't matter. The equations can make use either of previously defined components (such as `halfAdd`) or of the Boolean gates.

We can simulate this circuit by calling the function, like we did in the previous section. Though as inputs get bigger, entering different test inputs in the interpreter is a lot of work. To avoid this, we can describe a number of test cases in the file `First.hs`:

```
test1 = halfAdd Low  Low
test2 = fullAdd Low  (High,Low)
test3 = fullAdd High (Low,High)
```

And we can perform tests in the interpreter.

```
*First> test3
(Low,High)
*First> test2
(High,Low)
```

Note that if we try to simulate a circuit with inputs of the wrong type, we get a type error:

```
*First> fullAdd Low High Low

<interactive>:1:12:
    Couldn't match expected type `(Bit, Bit)'
           against inferred type `Bit'
    In the second argument of `fullAdd', namely `High'
    In the expression: fullAdd Low High Low
    In the definition of `it': it = fullAdd Low High Low
```

To simulate your circuit for more than one input at a time, you can use the `map` operation found in the `Prelude` module (which is loaded by default). It takes a function and a list of inputs as a parameter. As `map` takes a list of singular inputs, we have to transform the `fullAdd` function slightly because it currently takes two inputs:

```
fullAdd' (carryIn,(a,b)) = fullAdd carryIn (a,b)
```

Lists are denoted between square brackets:

```
*First> map fullAdd' [(Low,(High,Low)),(High,(High,Low))]
[(High,Low),(Low,High)]
```

It is also possible to ask for the type of a given function using the `:t` command:

```
*First> :t fullAdd'
fullAdd' :: (Bit, (Bit, Bit)) -> (Bit, Bit)
```

2.5 Generating VHDL

Before we can translate a circuit to VHDL we need to add some extra information to the module that holds our circuit description. We need to add the `TopEntity` annotation pragma to the `fullAdder` description to indicate that the full adder circuit is at the top of the circuit hierarchy:

```
{-# ANN fullAdd TopEntity #-}
```

From the interpreter we can now translate the full-adder circuit to VHDL using `:vhdl` command:

```
*First> :vhdl
  ...
  Normalization process
  ...
  Output VHDL
  ...
*First>
```

You will see a lot of debug info flowing across the screen, including the generated VHDL. If the compiler does not raise an exception/error, then that means that the CλaSH design is successfully translated to VHDL. The generated code can be found in the `vhdl` directory placed in the current working directory.

The result of the above `:vhdl` command is shown in subsection 2.5.1. Looking at this VHDL code, you can see that it is odd, in that it passes a clock and reset to circuits that are purely combinational. The reason is that at the moment, information concerning if a circuit is purely combinational or not is not stored after the VHDL is generated; so we always need to pass a clock and reset signal to a circuit. Future versions of CλaSH might omit these clock and reset ports for purely combinational circuits.

2.5.1 GENERATED VHDL

```
-- Automatically generated VHDL
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;


entity fullAddComponent_0 is
     port (carryInzBIj2 : in std_logic;
           dszBIl2 : in Z2TZLz2cUZRstd_logicstd_logic;
           casevalzBJHzBJH1 : out Z2TZLz2cUZRstd_logicstd_logic;
           clock : in std_logic;
           resetn : in std_logic);
end entity fullAddComponent_0;


architecture structural of fullAddComponent_0 is
     signal carryOutzBJf2 : std_logic;
     signal dszBIR1 : Z2TZLz2cUZRstd_logicstd_logic;
     signal sumzBIV1 : std_logic;
     signal carry2zBJ51 : std_logic;
     signal dszBIt2 : Z2TZLz2cUZRstd_logicstd_logic;
     signal sum1zBIx2 : std_logic;
     signal carry1zBIH1 : std_logic;
     signal bzBIr2 : std_logic;
     signal azBIp2 : std_logic;
begin
     casevalzBJHzBJH1.A <= sumzBIV1;

     casevalzBJHzBJH1.B <= carryOutzBJf2;

     carryOutzBJf2 <= carry2zBJ51 xor carry1zBIH1;
```

```
     comp_ins_dszBIR1 : entity halfAddComponent_1
                              port map (azBJj2 => carryInzBIj2,
                                        bzBJl2 => sum1zBIx2,
                                        reszBJvzBJv2 => dszBIR1,
                                        clock => clock,
                                        resetn => resetn);

     sumzBIV1 <= dszBIR1.A;

     carry2zBJ51 <= dszBIR1.B;

     comp_ins_dszBIt2 : entity halfAddComponent_1
                              port map (azBJj2 => azBIp2,
                                        bzBJl2 => bzBIr2,
                                        reszBJvzBJv2 => dszBIt2,
                                        clock => clock,
                                        resetn => resetn);

     sum1zBIx2 <= dszBIt2.A;

     carry1zBIH1 <= dszBIt2.B;

     bzBIr2 <= dszBIl2.B;

     azBIp2 <= dszBIl2.A;
end architecture structural;

-- Automatically generated VHDL
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;


entity halfAddComponent_1 is
     port (azBJj2 : in std_logic;
           bzBJl2 : in std_logic;
           reszBJvzBJv2 : out Z2TZLz2cUZRstd_logicstd_logic;
           clock : in std_logic;
           resetn : in std_logic);
end entity halfAddComponent_1;


architecture structural of halfAddComponent_1 is
     signal sumzBJr2 : std_logic;
     signal carryzBJn2 : std_logic;
begin
     reszBJvzBJv2.A <= sumzBJr2;

     reszBJvzBJv2.B <= carryzBJn2;

     sumzBJr2 <= azBJj2 xor bzBJl2;

     carryzBJn2 <= azBJj2 and bzBJl2;
end architecture structural;
```

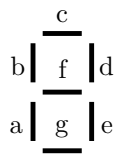

Figure 2.3: Digital Display

2.6 Exercises

2.1 Define the circuits `swap` and `copy`. `Swap` gets a pair of inputs, and outputs them in the swapped order. `Copy` gets one input and outputs it twice, as a pair. Here is how it should behave:

```
*First> map swap [(Low,High), (Low,Low), (High,Low)]
[(High,Low),(Low,Low),(Low,High)]
*First> map copy [Low, High]
[(Low,Low),(High,High)]
```

2.2 Define a `two-bit sorter`. It takes as input a pair of bits, and outputs the same bits, but the lowest one on the left hand side, and the biggest one on the right hand side.

2.3 Define a circuit with no inputs, and one output, which is alway high. Hint: input consisting of no wires is written as: `()`.

2.4 Define and simulate a *multiplexer* in CλaSH. A multiplexer circuit has two inputs: a signal, `c`, and a pair `(x,y)`. The output is equal to `x` if the signal `c` is `Low`, and to `y` if the signal `c` is `High`.

2.5 Use three full adders to make a three bit binary adder. Simulate your design and generate VHDL code.

2.6 Suppose you are designing a digital watch. It might come in handy to have a circuit that takes a four-bit binary number and displays it as a digital digit using a seven segment display. Your circuit might have the following interface (see Figure 2.3):

```
digitalDisplay (one, two, four, eight) =
  (a, b, c, d, e, f, g)
  where ...
```

Hint: start by making a table with 10 entries (0 .. 9) where you can see what parts of the display should light up for what number.

Chapter 3

Bigger Circuits

In this chapter we describe how to make more complicated circuits using choice constructs and abstract data types. We will also see how to use numbers in CλaSH.

3.1 Arithmetic

In CλaSH, we can not only deal with low-level wire types like bits, and gates like `hwand` and `hwxor`, but also with more abstract wire types and gates. One of these types is integers, of which we have several variants. On these integers, we have operations corresponding to abstract gates over integers. A list of the integer variants and the operations defined over them can be found in Appendix A

A simple circuit using these arithmetic gates is called `multiplyAccumulate` (See Figure 3.1). It takes three number as input: `x`, `y`, and `acc`. The number `x` and `y` are multiplied, and the result of that is added to the value of accumulator, `acc`.

```
mac acc x y = x * y + acc
```

We can straightforwardly simulate this circuit:

```
*First> mac 11 12 13
167
```

However, when we annotate the `mac` circuit to be the top entity and try to translate it to VHDL using the `:vhdl` command we get an error.

```
*First> :vhdl
*** Exception: Normalize.getNormalized: Unknown or non-representable
  function requested: First.mac
```

The reason that we can not translate the `mac` function to structural VHDL is due to its (non-representable) type:

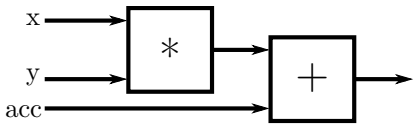

Figure 3.1: Multiply and Accumulate

```
*First> :t mac
mac :: (Num a) => a -> a -> a -> a
```

The `mac` circuit is polymorphic, although restricted to "number" types[1] as indicated by the `(Num a) =>` part of the type definition. Structural VHDL is not allowed to have signals with polymorphic types, meaning that the top-level entity can only have signals that are concrete/monomorphic. Note however that this does not mean that we can not use polymorphic definitions in our circuits: the CλaSH compiler will automatically propagate concrete types from the top entity down to any polymorphic circuits.

So a solution to our problem is to make a new circuit definition which is annotated directly with concrete types, and uses our polymorphic `mac` circuit:

```
type Int8 = Signed D8
mac8 :: Int8 -> Int8 -> Int8 -> Int8
mac8 acc x y = mac acc x y
```

We use the `type` keyword to create a type alias, `Int8`, for the 8-bit signed integer type, `Signed D8`. The input of the function `mac8` now consists of three values of the type `Int8` (the first three occurrences of `Int8` in the type of `mac8`), and the result of `mac8` (the last occurrence of `Int8` in the type expression) also is of type `Int8`. We could have annotated the original `mac` circuit with a concrete type, but defining a new circuit allows reuse of `mac` for different number types and bit-widths.

Unlike the original `mac` function which worked with numbers of infinite precision, the `mac8` function works with signed integers which have an 8-bit precision, meaning that the representation can overflow. Signed and Unsigned integers in CλaSH deal with an overflow by performing a wrap-around, meaning that when an integer exceeds one extreme of its representation that it continues from the opposite extreme. For example, in case of adding 7 to an 8-bit signed integer with a value of 127 (its positive extreme), results in the value -122:

```
*First> let k = (127::Int8)
*First> k
127
*First> k + 7
-122
```

So when we simulate the `mac8` circuit with the same inputs as we did with the `mac` circuit, we get a different result:

```
*First> mac 11 12 13
167
*First> mac8 11 12 13
-89
```

Finally, when we now annotate the `mac8` circuit to be the top entity, the CλaSH compiler will now correctly generate VHDL without raising an error:

```
-- Automatically generated VHDL
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
```

[1] Number types are types for which the operations `+`, `-`, `*`, and `fromInteger` are defined.

```
use std.textio.all;


entity mac16Component_0 is
     port (acczAe52 : in signed_16;
           xzAe72 : in signed_16;
           yzAe92 : in signed_16;
           reszAehzAeh3 : out signed_16;
           clock : in std_logic;
           resetn : in std_logic);
end entity mac16Component_0;


architecture structural of mac16Component_0 is
begin
     comp_ins_reszAehzAeh3 : entity macComponent_1
                                  port map (paramzAeYzAeY2 => acczAe52,
                                            paramzAf0zAf02 => xzAe72,
                                            paramzAf2zAf22 => yzAe92,
                                            reszAf4zAf42 => reszAehzAeh3,
                                            clock => clock,
                                            resetn => resetn);
end architecture structural;

-- Automatically generated VHDL
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;


entity macComponent_1 is
     port (paramzAeYzAeY2 : in signed_16;
           paramzAf0zAf02 : in signed_16;
           paramzAf2zAf22 : in signed_16;
           reszAf4zAf42 : out signed_16;
           clock : in std_logic;
           resetn : in std_logic);
end entity macComponent_1;


architecture structural of macComponent_1 is
     signal argzAfkzAfk3 : signed_16;
begin
     reszAf4zAf42 <= argzAfkzAfk3 + paramzAeYzAeY2;

     argzAfkzAfk3 <= resize(paramzAf0zAf02 * paramzAf2zAf22, 16);
end architecture structural;
```

We will return to the Multiply-Accumulate circuit in the next chapter on sequential circuits, in which we can actually store the accumulation in a memory element.

3.2 Elements of Choice

In CλaSH, choice can be achieved by a large set of syntactic elements, consisting of: `case` expressions, `if-then-else` expressions, pattern matching, and guards. The most general of these are the `case` expressions. All of these syntactic choice elements can also be translated to structural VHDL.

We will use all four syntactic choice elements to describe a multiplexer circuit. A multiplexer circuit has two inputs a signal, c, and a pair (x,y). The output is equal to x if the signal c is `Low`, and to y if the signal c is `High`:

```
*First> multiplexer Low (1,2)
1
*First> multiplexer High (1,2)
2
```

We start with defining a multiplexer using the most basic choice construct, the `if-then-else` expression:

```
multiplexerIfThenElse c (x,y) =
  if (c == Low) then
    x
  else
    y
```

As the result of an `if-then-else` expression is assigned to a signal, the `else`-clause must always be defined: i.e. there is no support for 'unknown' values.

```
multiplexerCase c (x,y) =
  case c of
    Low  -> x
    High -> y
```

Instead of matching on the `High` value, we could also use a default case, indicated by the `_` symbol:

```
multiplexerCase' c (x,y) =
  case c of
    Low -> x
    _   -> y
```

Expressions can also contain guards, where the expression is only executed if the guard evaluates to true, and continues with the next clause if the guard evaluates to false. The guard is the expression that follows the vertical bar (`|`) and precedes the assignment operator (`=`).

```
multiplexerGuards c (x,y) | c == Low  = x
                          | c == High = y
```

Instead of comparing the signal c to `High`, we could also use the `otherwise` expression. The `otherwise` expression always evaluates to `True`.

```
multiplexerGuards' c (x,y) | c == Low  = x
                           | otherwise = y
```

A *user-friendly* and also powerful form of choice that is not found in the traditional hardware description languages is pattern matching. A function can be defined in multiple clauses, where each clause corresponds to a pattern. When an argument matches a pattern, the corresponding clause will be used.

```
multiplexerPatterns Low  (x,y) = x
multiplexerPatterns High (x,y) = y
```

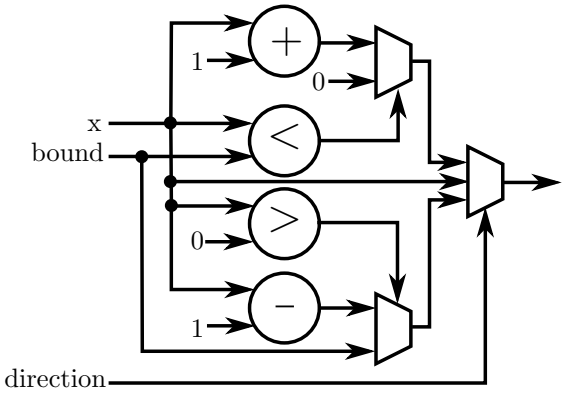

Figure 3.2: Counter circuit

Instead of making a clause for the `High` value, we could also make a default clause, again indicated by the `_` symbol:

```
multiplexerPatterns' Low (x,y) = x
multiplexerPatterns' _   (x,y) = y
```

We will now write a somewhat elaborate test case, in which we use simulation to check the equality of the outputs of the above circuits. We use the `all` function from the Prelude module, which takes a test function and a list of values, to test if all outputs are equal to 3. If this is the case, evaluation of `test4` function returns the value `True`.

```
test4 = all (==3)
  [ multiplexerIfThenElse High (2,3)
  , multiplexerCase       High (2,3)
  , multiplexerCase'      High (2,3)
  , multiplexerGuards     High (2,3)
  , multiplexerGuards'    High (2,3)
  , multiplexerPatterns   High (2,3)
  , multiplexerPatterns'  High (2,3)
  ]
```

Running `test4` in the interpreter returns the expected result:

```
*First> test4
True
```

3.3 A counter: combining arithmetic with choice

Using the arithmetic operations and choice elements we will now make a simple counter circuit (Figure 3.2). The counter counts up or down depending on the `direction` variable, and has a `bound` variable that determines both the upper bound and wrap-around point of the counter. The `direction` variable is of the following, user-defined, enumeration datatype. We introduce a new datatype using the `data` keyword.

```
data Direction = Up | Down | Hold
```

Firstly, we will define the circuit using the more familiar `case` and `if-then-else` expressions.

```
counter bound direction x = case direction of
  Up   -> if x < bound then
            x + 1
          else
            0
  Hold -> x
  Down -> if x > 0 then
            x - 1
          else
            bound
```

A second version of the counter, now using both pattern matching and guards, can be seen on the next page.

```
counter' bound Up   x | x < bound = x + 1
                      | otherwise = 0
counter' _     Hold x             = x
counter' bound Down x | x > 0     = x - 1
                      | otherwise = bound
```

Note that in the second clause, when we match on `Hold`, that we use an underscore for the first variable as we are not interested in the actual value of the bound, and hence do not need to give it a name.

As was the case with the multiplexer circuit, both circuit descriptions will result in the same physical circuit. The best choice element for the job is both highly personal, and dependent on the intended control flow. In some situations pattern matching leads to a clearer and concise description, while a `case`-expression might give the clearest descriptions in another situation.

3.4 Exercises

3.1 Define a swapper, a circuit that take in two inputs: an activate signal and a pair of signals, and the output is the pair of signals. If the activate signal is high, the order of the input pair is swapped, otherwise it stays the same.

3.2 Define a sorter circuit that sorts three numbers. On the output, the lowest number should be on the left, and the highest number should be on the right. Hint: start by making a table with all the orderings, so that you can see what relations should hold.

Chapter 4

Sequential Circuits

In this chapter we describe how to deal with sequential circuit in CλaSH. Sequential circuits in CλaSH are synchronous circuits, which means that there is one global clock affecting all delay components in the circuit.

4.1 The Mealy machine premise

The basic premise in CλaSH is that combinational circuit designs are lifted into a Mealy machine to make it a synchronous circuit. A graphical representation of a mealy machine is shown in Figure 4.1. We can annotate a combinational circuit with the `State` keyword to indicate which signals are connected to memory elements. Here is an example of a simple circuit called `edge`, that checks if its input has changed with respect to its previous input. It uses the `State` keyword to indicate which signal are connected to a memory element to remember the previous input.

```
edge (State prev) inp = (State inp, change)
  where
    change = hwxor inp prev
```

The function argument `prev` is annotated with `State` keyword, indicating that it is the output of the memory element. The first element of the output tuple is annotated with the `State` keyword indicating that it is the input of the memory element.

Having described the combinational logic and having connected the correct signals to the memory element, all that remains is to describe the initial content of the memory element to make the Mealy machine analogy complete. We give the circuit an initial state using the lifting function `^^^`:

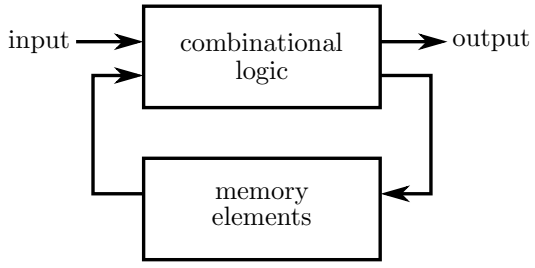

Figure 4.1: Mealy Machine

```
edgeL = edge ^^^ Low
```

When we ask for the type of the `edgeL` circuit using the `:t` command:

```
*First> :t edgeL
edgeL :: Stat Bit Bit
```

We see that it is a statefull machine which has a `Bit` as input type and `Bit` as output type. We can simulate a sequential circuit by using the operation `simulate`. It needs a circuit of type `Stat a b` (like `edgeL`) and a list of inputs. The list of inputs is interpreted as the different inputs at each clock tick.

```
*First> simulate edgeL [High,Low,Low,High]
[High,High,Low,High]
```

Here is another sequential circuit, which is called `toggle`. It has an internal state, which it outputs, and it takes one input. If the input is high, it changes the state. If not, it stays the same.

```
toggle (State prev) change = (State out, out)
  where
    out = hwxor change prev

toggleL = toggle ^^^ Low
```

As we can see, the definition of `out` is dependent on the previous value of `out`. Thus, there is a loop in the circuit. Loops are not allowed in combinational circuits, since the meaning of such circuits is unclear. But in sequential circuit, they are essential to implement any interesting behavior. Simulating `toggleL` gives:

```
*First> simulate toggleL [High,Low,Low,High]
[High,High,High,Low]
```

4.2 Composing components

Returning to the Multiply-Accumulate circuit of the previous chapter, we now store the accumulator in a memory element (see Figure 4.2):

```
macS (State s) (a,b) = (State s', s')
  where
    s' = mac8 s a b

macS0 = macS ^^^ 0
```

If we want to compose component, we need to add the 'Arrows' language pragma:

```
{-# LANGUAGE Arrows #-}
```

At the top of our design file.

```
macsum = proc (a,b,c,d) -> do
  r1 <- macS ^^^ 0 -< (a,b)
  r2 <- macS ^^^ 0 -< (c,d)
  returnA -< (r1 + r2)
```

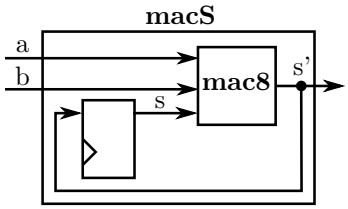

Figure 4.2: Stateful Multiply-Accumulate

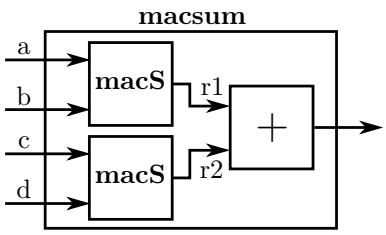

Figure 4.3: Composing two MACC circuits

4.3 Exercises

4.1 Define a circuit `evenSoFar`, which takes one input, and has one output. The output is high if and only if the number of high inputs has been even so far.

Simulate your circuit in CλaSH and generate VHDL.

4.2 Implement a `flipFlop` circuit, which takes two inputs (set,reset), and has one output. The circuit keeps an internal state, which is set to high when set is high, and set to low when reset is high. The internal state is also the output. You may decide yourself what to do when both inputs are high.

4.3 Define a circuit `always`, which has one input and one output. The output is high as long as the input stays high. If the input drops to low, the output stays low forever.

Chapter 5

Vectors and Higher-Order Functions

In this chapter we describe how to make more complicated circuits by aggregating signals in vectors and using higher-order functions to specify functions that work on entire vectors of signals. We have already seen some higher-order functions such a the `map` operation defined in the `Prelude` module, and the `simulate` function of the previous chapter; though it should be noted that these worked for lists. Lists can not be translated to VHDL by the CλaSH compiler, as their length cannot be easily identified at compile-time. Vectors on the other hand have their length encoded in their type, so their length can as such be easily identified at compile-time.

5.1 Binary Adders

A *bit adder* takes a pair of inputs. The first part is the carry bit, the second part a binary number, represented as a vector bits, *least significant bit first*. The bit adder will add the bit to the binary number, resulting in a binary number and a carry out.

We define a bit adder `bitAdder` in CλaSH using higher-order functions over vectors. In this case, we will use half-adders to add the carry to the binary number: the first half-adder will add the carry bit to the first bit of the binary number, the second half-adder will add the carry from the previous half adder to the second bit of the binary number, and so on. The result carry of the last half adder will be the carry out of the entire bit adder.

```
bitAdder carryIn xs = (sum,carryOut)
  where
    res           = vzipWith halfAdd (carryIn +> carries) xs
    (sum,carries) = vunzip res
    carryOut      = vlast carries
```

The `vzipWith` higher-order function takes a half-adder and applies it pairwise to the two bit-vectors; note that the length of the result vector is equal to the smallest of the two input vectors. The first bit-vector is the carry-in put in front of the carries produced by the half-adders. The `+>` operation concatenates an element to a vector.

The `vunzip` function extracts the sum and carry bits out of the vector of half-adder results. The `vunzip` operation takes a list of tuples and returns a tuple of lists.

Finally, the `vlast` function is used get the last element of the carries-vector. Note that `res` depends on the value of `carries`, while `carries` depends on the value of `res`. This is perfectly legal in CλaSH as we did *not* create combinational loop: the

half-adders do not depend on the values of their own results, but only on the results of the previous half-adders.

A slightly more complicated circuit is the `adder` circuit that takes a carry and a pair of binary numbers, and adds them up. This is called a binary adder. The description using higher-order functions is almost the same as the bit-adder circuit, except that we use full-adders instead of half-adders, and that the type of second input is a vector of bit-tuples instead of single bits.

```
adder carryIn xs = (sum, carry)
  where
    res           = vzipWith fullAdd (carryIn +> carries) xs
    (sum,carries) = vunzip res
    carryOut      = vlast carries
```

5.1.1 simulation and vhdl generation

Due to some bugs in CλaSH we current need to add type signatures to `bitAdder` and `adder` before we can simulate them; so let's make them 4-bit adders:

```
bitAdder :: Bit -> Vector D4 Bit -> (Vector D4 Bit, Bit)
adder    :: Bit -> (Vector D4 Bit, Vector D4 Bit) -> (Vector D4 Bit, Bit)
```

5.2 Connection Patterns

```
row f z xs = (ys,z')
  where
    res     = vzipWith f (z +> zs) xs
    (ys,zs) = vunzip res
    z'      = vlast zs

bitAdderRow carryIn xs = row halfAdd carryIn xs
bitAdderRow'           = row halfAdd

binaryAdder            = row fullAdd
```

Appendix A

Quick Reference Guide

CλaSH has a myriad of types that a developer can use. This appendix will explain which types are currently supported, and highlights their (sometimes cumbersome) details.

A.1 Primitive Types

In CλaSH we have three basic primitive types, all of which are enumerations.

A.1.1 BIT

```
data Bit = High | Low
```

As the above type signature indicates, the `Bit` type currently has only two states (unlike the familiar `std_logic` type found in VHDL). Support for a high impedance state (`Z`) might be added in a future version of the CλaSH compiler.

Functions

```
hwand :: Bit -> Bit -> Bit
High `hwand` High   = High
_ `hwand` _         = Low

hwor  :: Bit -> Bit -> Bit
High `hwor` _       = High
_ `hwor` High       = High
Low `hwor` Low      = Low
```

```
hwxor :: Bit -> Bit -> Bit
High `hwxor` Low    = High
Low `hwxor` High    = High
_ `hwxor` _         = Low

hwnot :: Bit -> Bit
hwnot High          = Low
hwnot Low           = High
```

A.1.2 BOOL

```
data Bool = True | False
```

Functions

Besides the boolean data type, CλaSH has also support for:

- Several comparison operators (infix):

  - Strict lesser-than: `<`
  - Non-strict lesser-than: `<=`
  - Strict greater-than: `>`
  - Non-strict greater-than: `>=`
  - Equality: `==`
  - Inequality: `/=`
- Binary operators (infix):
  - And: &&
  - Or: ||
- The unary negation operator: *not*

A.1.3 custom enumerations

Besides the built-in enumeration types defined earlier, CλaSH has also support for custom enumeration types. These custom enumeration types are constructed using the same syntax as the earlier enumeration types. For example, we can define the type `Color` as an enumeration of the colors `Red`, `Green` and `Blue`:

```
data Color = Red | Green | Blue
```

A.2 Special Types

CλaSH has several special types, the `State` wrapper type, which tells the CλaSH compiler if something is part of the state of a function. The other special types are the type-level numerals and booleans.

The reader should note that these special types have *no* run-time representation! And as such a user should never attempt to evaluate a term that has one of these special types!

A.2.1 type-level primitives

CλaSH has support for several type-level primitives, meaning primitives that can be used for *type*-level computations. Though they have term-level representations, so that they can be used as function arguments, their actual value is `undefined`. The symbol `undefined` denotes a computation that never terminates! So a user should never evaluate one of these terms!

CλaSH has support for *type*-level booleans and several boolean comparison operators. Also, CλaSH has support for *type*-level numerals and several numeric classes and arithmetic operators.

An important operator that comes into play when using *type*-level computations, is the *type equality coercion* operator, `~`. The `~` operator asks the type-checker to enforce that the type on the left-hand side of the `~` operator is 'equal' to the type on the right-hand side. We put 'equal' between quotes to indicate that the types are not intentionally *equal*, rather that, after type erasure the program will not 'go wrong'. This operator can only be used in the context of a type signature. Below we see the `~` operator being used to ask the type-checker to witness that the *type*-variable *s'* is indeed the `Succ`essor of the *type*-variable *s*.

```
f :: (Succ s ~ s') => ...
```

Type-level Booleans

We have, as expected, two type-level booleans, `True` and `False`, the type-system underlying CλaSH does not allow us to group these two types (into a *kind*). A user should not confuse the types `True` and `False` with the *terms*, `True` and `False`. These terms have the *type* `Bool`! Whilst `True` and `False` are themselves types! Though these 2 types have term-level representations, they are not discussed in this manual as they are not allowed to be used as function arguments.

There are several type-level comparison operators, similar to how we know them for the familiar term-level boolean comparison operators. However, these being *infix* type-level operators, they are placed between colons (:). Meaning that, for example, the type-level strict lesser-than operator is written like: `:<:`. The supported operators are:

- Several comparison operators (infix):
  - Strict lesser-than: `:<:`
  - Non-strict lesser-than: `:<=:`
  - Strict greater-than: `:>:`
  - Non-strict greater-than: `:>=:`
  - Equality: `:==:`
- Binary operators (infix):
  - And: `:&&:`
  - Or: `:||:`
- The unary negation operator: `Not`

Throughout the rest of this document, the enclosing colons for infix operations are removed from the text for purposes of presentation. They should however be present when used in actual code!

Type-level Integers

Besides the type-level booleans, CλaSH has support for type-level numerals as well. These type-level numerals are used in both the fixed-size vector types, as the two integer types supported in CλaSH. The type-level numerals themselves are cumbersome in their use, that is why type aliases have been defined for integers in the range of -10000 to +10000. So, being type-level integers, each numeric instance is a type in its own right! All type-level integer aliases are prefixed with a capital `D`. So, the type-level integer 2, for example, is written as: `D2`.

Some functions require a type-level integer to be passed as an argument, in this case, the *term*-level representations of these *type*-level integers have to be used. All *term*-level representations are prefixed by a lower-case *d*. So, the *term*-level representation of the *type*-level integer `D2`, for example, is written as: *d2*.

CλaSH has defined three classes to which an type-level integer (and thus also its alias) can belong:

- `IntegerT` – The integer class that covers the complete range of integers.
- `NaturalT` – The natural class that covers the natural range of integers.
- `PositiveT` – The positive class that covers the positive range of integers.

A user can ask the type-checker to witness that a type-level integer belongs to one of these classes by specifying it in the context of a (function/type) definition. For example, we ask the type-checker in the code below, that the variable *s* belongs to the type-level integers in the natural range.

```
f :: NaturalT s => s -> ...
```

Besides these numeric classes, there are also several arithmetic operators. Like the type-level boolean comparison operators, *infix* operators must be placed between colons (:). So, the type-level addition, for example, is written as such: `:+:`. The supported operators are:

- Arithmetic computations (infix):
  - Addition: `:+:`
  - Subtraction: `:-:`
  - Multiplication: `:*:`
- Arithmetic computations (prefix):
  - Multiply by 2: `Mul2`
  - Faculty: `Fac`
  - Division by 2: `Div2`
  - Power of 2: `Pow2`
- Arithmetic relations (prefix):
  - Successor: `Succ`
  - Predecessor: `Pred`
- Arithmetic property query (prefix):
  - Is it an even number?: `IsEven`
  - Is it an odd number?: `IsOdd`

Again, throughout the rest of this document, the enclosing colons for infix operations are removed from the text for purposes of presentation. They should however be present when used in actual code!

A.3 Aggregate Types

Aggregate types are types that use a single constructor to group/aggregate multiple values. CλaSH has support for many of these aggregate types.

A.3.1 tuples

Tuples can group values using the `(,)` syntax, which represents a two-tuple, three-tuples are constructed using the `(,,)` syntax. Every increase in tuple size adds another comma. As an example, we will use tuples to define a screen that is 5 pixels wide and 4 pixels high, and each pixel is made up out of 3 colors. We can define the types as follows:

```
type Pixel    = (Color,Color,Color)
type Row      = (Pixel,Pixel,Pixel,Pixel,Pixel)
type Screen   = (Row,Row,Row,Row)
```

We can than make an instance/term of this screen as follows:

```
pixel :: Pixel
pixel = (Red,Green,Blue)

row :: Row
row = (pixel,pixel,pixel,pixel,pixel)

screen :: Screen
screen = (row,row,row,row)
```

A.3.2 RECORDS

We call the access to values inside a tuple an anonymous access: it has no name. For example, if we want to access the second color in the pixel tuple, we have to give it a name ourselves:

```
secondColor :: Pixel -> Color
secondColor (_,x,_) = x
```

For these small tuples the notational overhead manageable, however, once tuples grow larger they become unwieldy. For this, and other reasons, CλaSH has support for Record aggregate types, which have named accessors.

We will redefine the `Pixel`, previously defined as a tuple, using the record syntax, `Constructor {,}`.

```
data Pixel = Pix {red :: Color, green :: Color, blue :: Color}
```

Note that the record syntax also introduces a new constructor, *Pix*, used to create the new `Pixel` data type. We first redefine the *secondColor* function using the 'regular' record syntax:

```
secondColor :: Pixel -> Color
secondColor Pix {red = red, green = green, blue = blue} = green
```

The notational burden has increased instead of lightened by the use of records. To omit the explicit introduction of the accessor functions we have support for the record wildcard syntax[1], `Constructor {..}`. This syntax automatically introduces the names of the undefined accessor functions inside the scope of the function in which it is used:

```
secondColor :: Pixel -> Color
secondColor Pix {..} = green
```

Finally, our function has become both concise and easy to understand.

A.4 Integer Types

So as to be a partially behavioural HDL, CλaSH has support for integers and integer operations. All integers have a range specification. For the sized integers, the bit-size of the integer is encoded in the type, thus determining the representable range. The safe array index encode an inclusive upper bound in their type, allowing the integer to range from 0 to the upper bound.

[1] You must specify the RecordWildCards LANGUAGE Pragma to activate this syntax.

A.4.1 SIZED INTEGER

The integers with a bit-size indication come in two forms: signed and unsigned integers. The signed integers are encoded in 2's complement, so their representable range runs from -2^{n-1} to $+2^{n-1}-1$, where n is the bit size of the integer. The representable range for unsigned integers of course runs from 0 to $+2^{n}-1$.
The type declaration of the signed integer is:

```
newtype (NaturalT size) => Signed size = ...
```

The type declaration of the unsigned integer is:

```
newtype (NaturalT size) => Unsigned size = ...
```

Where the *size* variable has to be a type-level integer indicating the bit-size of the integer. For example, we define the type alias for a 8-bit signed integer as such:

```
type Int8 = Signed D8
```

In general, we can construct integer literals as such:

```
x = (3 :: Int8)
```

Sometimes the CλaSH compiler might simplify an expression too much, and the above code might result in an error where the compiler complains that "undefined" is not a built-in function. If you encounter this error, it might help to rewrite the above code to the code shown below:

```
x = (fromInteger 3) :: Int8
```

Functions

The `Signed` type has support for:

- Addition (infix): `+`
- Subtraction (infix): `-`
- Multiplication (infix): `*`
- Negation (unary prefix): `-`
- Bit-shift left (prefix): `shiftL`
- Bit-shift right (prefix): `shiftR`

The `Unsigned` type on the other hand only has support for:

- Addition (infix): `+`
- Subtraction (infix): `-`
- Multiplication (infix): `*`
- Bit-shift left (prefix): `shiftL`
- Bit-shift right (prefix): `shiftR`

All the above operators perform a wrap-around when an overflow occurs. Also, if you try to apply the prefix negation operator on a `Unsigned` (note that it is *not* listed as a supported operator!) the compiler will promptly throw an error.

It should be noted that the above arithmetic operations have the same argument and result type, meaning that their representation size remains the same:

```
mult :: Unsigned D4 -> Unsigned D4 -> Unsigned D4
mult a b = a * b
```

So if we call the above function with a and b both set to the maximum representable value of 15, the result of the multiplication will of course *not* be 225:

```
CLasHi> mult 15 15
1
```

The result is not 225 because you need 8 bits to represent that number. If we want to make sure that the result is 8 bits, than we will need to enlarge the arguments to 8 bits as well. For this enlargement we can use the *resizeUnsigned* function function:

```
resizeUnsigned :: (NaturalT nT, NaturalT nT') => Unsigned nT -> Unsigned nT'
```

This function changes the bitsize of the integer. The *resizeUnsigned* function zero-extends, and the equivalent function for the `Signed` type, *resizeSigned*, sign-extends.

So, if we want the *mult* function to give 225 as a result after multiplying 15 by 15, we will need to rewrite it using the *resizeWord* function:

```
mult' :: SizedWord D4 -> SizedWord D4 -> SizedWord D8
mult' a b = (resizeWord a) * (resizeWord b)

CLasHi> mult' 15 15
225
```

A.4.2 SAFE ARRAY INDEX

For the vector operations discussed earlier we need *safe* indexes, indexes with an *exclusive* upper bound that give an error when a value exceeds either the lower bound (of zero) or the specified upper bound. For this purpose CλaSH has support for the `Index` type type:

```
newtype (NaturalT upper) => Index upper = ...
```

Where the *upper* variable has to be a type-level integer indicating the inclusive upper bound of the integer. For example, we define the type alias for a integer that allows values between 0 and 8 as such:

```
type WordMax8 = Index D9
```

In general, we can construct integer literals as such:

```
x = (3 :: WordMax8)
```

Sometimes the CλaSH compiler might simplify an expression too much, and the above code might result in an error where the compiler complains that "undefined" is not a built-in function. If you encounter this error, it might help to rewrite the above code to the code shown below:

```
x = (fromInteger 3) :: WordMax8
```

If a user attempts to create a literal that exceeds either side of the allowed range, the circuit will throw an error at *simulation*-time!

Functions

The `Index` type has support for:

- Addition (infix): `+`
- Subtraction (infix): `-`
- Multiplication (infix): `*`

When any of the above operator results in a value that exceeds either side of the allowed range, the circuit will throw an error at *simulation*-time! Also, if you try to apply the prefix negation operator on a `Index` (note that it is *not* listed as a supported operator!) the compiler will promptly throw an error.

Interesting are the conversion functions between `Index` and `Unsigned`. Using these conversion functions, a developer can still exploit the *overflow* behaviour of `Unsigned`, but guarantee the range safety offered by `Index`. Of course, there are some limitations: The largest being that when converting from `Unsigned` to `Index`, then the upper bound of a resulting `Index` has to be of a power of two, as that is the only static guarantee the conversion can give. The function that converts a `Index` to a `Unsigned` is seen in Code Snippet **??**, and the function that converts a `Unsigned` to a `Index` is seen in Code Snippet **??**.

```
fromIndex ::
  ( NaturalT nT
  , NaturalT nT'
  , ((Pow2 nT') :>: nT) ~ True
  , Integral (Index nT)
  ) => Index nT -> Unsigned nT'
fromIndex index = Unsigned (toInteger index)
```

Most of the type context should be familiar; the line, `((Pow2 nT') :>: nT) ~ True`, asks the type-checker that the upper bound (`nT`) of the `RangedWord` type fits in the maximum representable value (`Pow2 nT'`) of the `SizedWord` type. The context, `Integral (Index nT)`, has to be specified so that type-checker knows that the `Index` type has a *toInteger* function.

```
fromUnsigned ::
  ( PositiveT nT
  , Integral (Unsigned nT)
  ) => Unsigned nT -> Index (Pow2 nT)
fromUnsigned unsigned = Index (toInteger unsigned)
```

The functionality, and the type-signature, of the above conversion should be straightforward for the reader. Also here, the context, `Integral (Unsigned nT)`, has to be specified to let the type-checker know that the `Unsigned` type has a *toInteger* function.

Appendix B

Answers

2.1 Here is how we define `swap` and `copy`

```
swap (a,b) = (b,a)
copy a     = (a,a)
```

2.2 We could define the sorter `twoBitSort` in the following way:

```
twoBitSort (a,b) = (min,max)
  where
    min = hwand a b
    max = hwor  a b
```

2.3 Here is the constant `alwaysHigh` circuit:

```
alwaysHigh () = High
```

2.4 One could define a `multiplexer` as follows:

```
multiplexer c (x,y) = out
  where
    out   = hwor left right
    left  = hwand (hwnot c) x
    right = hwand c y
```

Using a more behavioral approach, based on pattern matching, we could also define it as follows:

```
multiplexer Low  (x,y) = x
multiplexer High (x,y) = y
```

2.5 A `treeBitAdder` can be defined as follows:

```
treeBitAdder carryIn (a1,b1,c1) (a2,b2,c2) =
  ((a3,b3,c3),carryOut)
  where
    (a3, carryA)   = fullAdd carryIn (a1,a2)
    (b3, carryB)   = fullAdd carryA  (b1,b2)
    (c3, carryOut) = fullAdd carryB  (c1,c2)
```

2.6 Using pattern matching we can define `digitalDisplay` as follows:

```
digitalDisplay (Low , Low , Low , Low ) = (High, High, High, High, High, Low , High)
digitalDisplay (Low , Low , Low , High) = (Low , Low , Low , High, High, Low , Low )
digitalDisplay (Low , Low , High, Low ) = (High, Low , High, High, Low , High, High)
digitalDisplay (Low , Low , High, High) = (Low , Low , High, High, High, High, High)
digitalDisplay (Low , High, Low , Low ) = (Low , High, Low , High, High, High, Low )
digitalDisplay (Low , High, Low , High) = (Low , High, High, Low , High, High, High)
digitalDisplay (Low , High, High, Low ) = (High, High, High, Low , High, High, High)
digitalDisplay (Low , High, High, High) = (Low , Low , High, High, High, High, Low )
digitalDisplay (High, Low , Low , Low ) = (High, High, High, High, High, High, High)
digitalDisplay (High, Low , Low , High) = (Low , High, High, High, High, High, High)
digitalDisplay _                        = (High, High, High, Low , Low , High, High)
```

3.1 Using pattern matching we can define `swapper` as follows:

```
swapper Low   (x,y) = (x,y)
swapper High  (x,y) = (y,x)
```

3.2 We could define `threeNumSort` as follows:

```
threeNumSort a b c  | b <= c && a <= b = (a,b,c)
                    | b <= c && a <= c = (b,a,c)
                    | a <= b && c <= a = (c,a,b)
                    | a <= b && c <= b = (a,c,b)
                    | b <= a && c <= b = (c,b,a)
                    | b <= a && c <= a = (b,c,a)
```

4.1 We can define evenSoFar as follows:

```
evenSoFarS (State evensofar) inp = (State even, evensofar)
  where
    even = hwxor evensofar inp

evenSoFar = evenSoFarS ^^^ High
```

This is *almost* the same as the `edge` circuit.

4.2 We can define `flipFlop` as follows:

```
flipFlopS (State s) (set, reset) = (State s', s)
  where
    s' = hwand up (hwnot reset)
    up = hwor s set

flipFlop = flipFlopS ^^^ Low
```

4.3 The circuit `always` can be defined as follows:

```
alwaysS (State sofar) inp = (State ok, ok)
  ok = hwand sofar inp

always = alwaysS ^^^ High
```