# A Slightly Revised Tutorial on Lava: A Hardware Description and Verification System

Koen Claessen
koen@cs.chalmers.se

Mary Sheeran
ms@cs.chalmers.se

May 7, 2007

# Contents

# Chapter 1

# Introduction

Lava is an experimental tool for hardware design and verification. Using Lava, one can describe circuits using a simple functional hardware description language. The descriptions are short and sweet, and do not suffer from the verbosity of more standard hardware description languages (HDLs) like VHDL and Verilog. On the other hand, we cannot express the same things as in these large, expressive (and complicated) languages. For example, we cannot express low level details about timing. What we can express very nicely, though, is the ways in which circuits are built from sub-circuits. Lava facilitates the description of *connection patterns* so that they are easily reusable. For some kinds of circuits, for example in signal processing, this is exactly what we want to do. Lava also provides many different ways of analysing our circuit descriptions. We can simulate circuits, just as with more standard HDLs, but we can also use symbolic methods to generate input to analysis tools such as automatic theorem provers and model checkers. Indeed, the same methods are used to generate structural VHDL from Lava circuit descriptions. Our aim in this tutorial is to gently introduce this new style of circuit design and analysis, by means of examples.

Lava is used at Chalmers as a platform for experiments in the formal verification of hardware [3, 2]. (Note, however, that both of these references are about an older version of Lava, in which circuit descriptions are a bit more complicated.) Satnam Singh, on the other hand, uses Lava in real industrial design projects at Xilinx Inc., one of the main suppliers of Field Programmable Gate Arrays (FPGAs). In particular, Lava has been used with great success in the development of FPGA cores such as filters and Bezier curve drawing circuits, and of customer applications such as digital signal processing for high speed networks and for high performance graphics applications.

Lava really consists of a simple hardware description language embedded in the powerful functional programming language Haskell. So it can be seen as a domain specific language embedded in a general purpose programming language. We describe circuits by writing Haskell programs – and the Lava system itself consists of a set of Haskell modules that give the user various facilities. The embedded language is quite similar to the Lustre synchronous dataflow language [7]. The idea of using a functional programming language to describe hardware was first proposed in the early eighties [14, 15, 8], and there has been quite a lot of work in the area since then [16, 17, 11, 13, 12, 6]. Our intention in building the Lava system (together with Singh) is to provide a tool that demonstrates the feasibility of doing circuit design and analysis using a functional language.

The main idea in Lava is that a single circuit description can be analysed in a variety of different ways, by giving different *interpretations* to its components (and sometimes even to its connection patterns). The simplest of these interpretations gives us ordinary simulation. But we can do much more. We can allow *symbolic* rather than concrete data to flow in the circuit, and in this way collect information about the circuit in various different ways. For example, we can run the circuit on symbolic data and produce expressions on the outputs that indicate how each output is related to the inputs. This can be useful when developing a first implementation. However, the expressions can get too large for humans to interpret. Then, we hook up external analysis tools, such as automatic theorem provers, to help us to analyse our circuits. When we hook up to external theorem provers, we are actually using Haskell as a proof scripting language. This turns out to be very convenient. Similarly, when we hook up to other external tools, such as VHDL-based CAD tools, we use Haskell as a scripting language. One way to view the Lava system is as a tool for linking together and controlling other tools in a unified way! Thus Haskell is used not only to construct circuit descriptions but also to control the tools that process those descriptions. The user sees only one language, rather than having to work with many, as is more usual in the CAD world.

This tutorial introduces the style of circuit description used in Lava, by means of very simple examples. It emphasises the way in which Lava *combinators* can be used to capture common interconnection patterns. It shows the three most important *interpretations* or circuit analysis methods – simulation, generation of VHDL code, and generation of logical formulas for input to theorem provers. After working through the tutorial, you should understand how to describe and analyse simple combinational and sequential circuits using Lava. We hope that the quick reference sections at the back of the tutorial will also help you to get started.

# Chapter 2

# Getting Started

In this chapter, we show how to describe some simple circuits in the Lava system, and run the interpreter on them.

## 2.1 Your First Circuit

To make a first circuit description, start up the text editor of your choice, and create a text file called `First.hs`, for example. Lava file names have the extension `.hs`.

We are going to define a so-called *half adder* (see figure 2.1). A half adder is a component that is for example used in the implementation of a binary adder. It takes as an input two bits, and adds them up. 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 Lava.

```
import Lava

halfAdd (a, b) = (sum, carry)
  where
    sum   = xor2 (a, b)
    carry = and2 (a, b)
```

We import a module called `Lava`, which defines a number of operations that we can use to build circuits. Notably, it contains the definitions of the gates `xor2` and `and2`. Appendix A contains a list of such predefined operations.

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

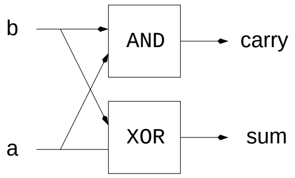

Figure 2.1: A half adder.

## 2.2 The Lava Interpreter

During the development of a collection of circuits, we mainly use the Lava *interpreter*. This is actually the Haskell interpreter *Hugs* [9]. The command is `lava`.

```
%  lava
-- Lava2000 -------------------------------------------------
...
Prelude>
```

We can use the interpreter to load different modules with circuit definitions, and to type in commands that we want to execute.

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

```
Prelude> :l First.hs
Reading file "First.hs":
...
First.hs
Main>
```

One of the things we can do with a circuit is to simulate it. Simulation is done in Lava with the operation `simulate`. It takes two arguments; one is the circuit to simulate (in this case `halfAdd`), and the other is the input to the circuit (in this case a pair of bits).

```
Main> simulate halfAdd (low,low)
(low, low)
Main> simulate 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:

```
Main> :r
...
Main>
```

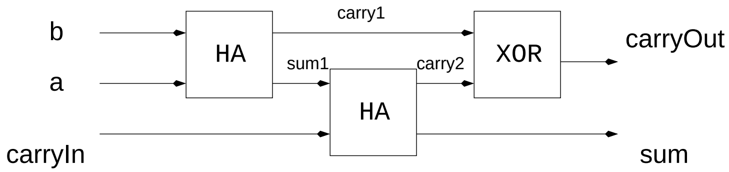

Figure 2.2: A full adder.

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

```
Main> :q
[Leaving Hugs]
%
```

## 2.3 Your Second Circuit

You guessed it! Your second circuit is going to be a *full adder* (see 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       = xor2 (carry2, carry1)
```

Note that, just like the half adder, this circuit has *one* input. This one input consists of a pair of a bit and a pair of bits. We could also have represented the input as a triple of bits, but we shall later see why we made this particular choice.

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 using the `simulate` operation that we used in the previous section. Though as inputs get bigger, typing in 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 = simulate halfAdd (low,low)
```

```
  test2 = simulate fullAdd (low,(high,low))
  test3 = simulate fullAdd (high,(low,high))
```

And we can perform tests in the interpreter.

```
  Main> test3
  (low, high)
  Main> test2
  (high, low)
```

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

```
Main> simulate fullAdd (low,high,low)
ERROR - Type error in application
*** Expression     : simulate fullAdd (low,high,low)
*** Term           : fullAdd
*** Type           : (Signal Bool,(Signal Bool,Signal Bool))
                                -> (Signal Bool,Signal Bool)
*** Does not match : (Signal Bool,Signal Bool,Signal Bool)
                                -> (Signal Bool,Signal Bool)
```

`Signal Bool` is the type of a single bit wire in Lava.

To simulate your circuit for more than one input at a time, you can use the operation `simulateSeq`. It takes a circuit and a list of sample inputs as a parameter. Lists are denoted between square brackets.

```
  Main> simulateSeq halfAdd [(low,low), (low,high), (high,low)]
  [(low,low), (high,low), (high,low)]
```

There is a special list, called `domain`, which contains all the values of a certain input shape.

```
  Main> simulateSeq halfAdd domain
  [(low,low), (high,low), (high,low), (low,high)]
```

Here, `domain` produced each possible two bit input. To check what those values were, we can simply ask for the value of `domain` at the appropriate type:

```
Main> domain::[(Signal Bool, Signal Bool)]
[(low,low),(low,high),(high,low),(high,high)]

Main> domain::[(Signal Bool, (Signal Bool, Signal Bool))]
[(low,(low,low)),(low,(low,high)),(low,(high,low)),(low,(high,high)),
 (high,(low,low)),(high,(low,high)),(high,(high,low)),(high,(high,high))]
```

It is also possible to ask for the type of a given function:

```
Main> :t halfAdd
halfAdd :: (Signal Bool,Signal Bool) -> (Signal Bool,Signal Bool)
```

Not all input shapes (for example inputs containing numbers!) have a finite domain list associated with them.

## 2.4 Generating VHDL

Given a Lava circuit description, we can generate VHDL from it, by using the operation `writeVhdl`. It takes two arguments, the name of the VHDL definition as a string, and the circuit.

```
  Main> writeVhdl "fullAdd" fullAdd
  Writing to file "fullAdd.vhd" ... Done.
```

The VHDL file that is generated will assume that there are definitions of the gates. The Lava distribution provides these definitions in the file `Lava2000/Vhdl/lava.vhd`. We must load this file into the VHDL working library and compile it.

Normally, the VHDL generator gives names to the inputs and outputs automatically. If we want to give names to the input ourselves, we can do this by using the operation `writeVhdlInput`. Here is how we use it:

```
  Main> writeVhdlInput "fullAdd" fullAdd
                          (var "carryIn", (var "a", var "b"))
  Writing to file "fullAdd.vhd" ... Done.
```

And lastly, if we also want to give names for the outputs, we can use the operations `writeVhdlInputOutput`. Here is how we use it:

```
  Main> writeVhdlInputOutput "fullAdd" fullAdd
                          (var "carryIn", (var "a", var "b"))
                                          (var "sum", var "carryOut")
  Writing to file "fullAdd.vhd" ... Done.
```

See figure 2.3 for the result of this last operation. Note that the description has been *flattened* all the way down to a gate-level netlist. No hierarchy remains. Lava really is just some modules that help with writing netlist generators. What happens under the hood is that we run the circuit description with symbolic inputs, producing an internal representation of the netlist. Then, we walk over this to print VHDL. Later, we will instead print the netlist in CNF (for input to a SAT-solver) or in SMV input format (for input to a model checker).

Looking at this VHDL code, you can see that it is odd, in that it passes the clock to every combinational gate! If you don't feel like doing this, you could use the module `VhdlNew` and the accompanying gate definitions available in the

```
-- Generated by Lava 2000

use work.all;

entity
  fullAdd
is
port
  -- clock
  ( clk : in bit

  -- inputs
  ; carryIn : in bit
  ; a : in bit
  ; b : in bit

  -- outputs
  ; sum : out bit
  ; carryOut : out bit
  );
end entity fullAdd;

architecture
  structural
of
  fullAdd
is
  signal w1 : bit;
  signal w2 : bit;
  signal w3 : bit;
  signal w4 : bit;
  signal w5 : bit;
  signal w6 : bit;
  signal w7 : bit;
  signal w8 : bit;
begin
  c_w2      : entity id    port map (clk, carryIn, w2);
  c_w4      : entity id    port map (clk, a, w4);
  c_w5      : entity id    port map (clk, b, w5);
  c_w3      : entity xor2  port map (clk, w4, w5, w3);
  c_w1      : entity xor2  port map (clk, w2, w3, w1);
  c_w7      : entity and2  port map (clk, w2, w3, w7);
  c_w8      : entity and2  port map (clk, w4, w5, w8);
  c_w6      : entity xor2  port map (clk, w7, w8, w6);

  -- naming outputs
  c_sum     : entity id    port map (clk, w1, sum);
  c_carryOut : entity id    port map (clk, w6, carryOut);
end structural;
```

Figure 2.3: The VHDL code for the full adder in `fullAdd.vhd`.

file `gates.vhd` in directory `Lava2000/Vhdl`. Now, each of the VHDL generation functions has a clocked and unclocked version (`writeVHdlClk`, `writeVhdlNoClk` etc.). You should import the module `VhdlNew` if you want to use these functions. (It has been assumed that your project directory is called `work`.) The full adder is a purely combinational circuit, so it makes sense to produce a circuit without a clock. (We will return to clocks, D flip-flops etc. in a later chapter.) The following example

```
test1 = writeVhdlInputOutputNoClk "fullAddNew" fullAdd
          (var "cin", (var "a", var "b")) (var "sum", var "cout")
```

produces the VHDL code in Figure 2.4

## 2.5 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 they should behave:

```
Main> simulateSeq swap [(low, high), (low, low), (high, low)]
[(high, low), (low, low), (low, high)]
Main> simulateSeq 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 highest one on the right hand side.

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

2.4 Define and simulate a *multiplexer* in Lava. A multiplexer circuit has as an input a pair of a signal and a pair `(x, y)`. The output is equal to `x` if the signal is low, and to `y` if the signal 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 2.5):

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

```
library ieee;

use ieee.std_logic_1164.all;

entity
  fullAddNew
is
port
  (



    cin : in std_logic
  ; a : in std_logic
  ; b : in std_logic


  ; sum : out std_logic
  ; cout : out std_logic
  );
end fullAddNew;

architecture
  structural
of
  fullAddNew
is
  signal w1 : std_logic;
  signal w2 : std_logic;
  signal w3 : std_logic;
  signal w4 : std_logic;
  signal w5 : std_logic;
  signal w6 : std_logic;
  signal w7 : std_logic;
  signal w8 : std_logic;
begin
  c_w2      : entity work.wire  port map (cin, w2);
  c_w4      : entity work.wire  port map (a, w4);
  c_w5      : entity work.wire  port map (b, w5);
  c_w3      : entity work.xorG  port map (w4, w5, w3);
  c_w1      : entity work.xorG  port map (w2, w3, w1);
  c_w7      : entity work.andG  port map (w2, w3, w7);
  c_w8      : entity work.andG  port map (w4, w5, w8);
  c_w6      : entity work.xorG  port map (w7, w8, w6);


  c_sum     : entity work.wire  port map (w1, sum);
  c_cout    : entity work.wire  port map (w6, cout);
end structural;
```

Figure 2.4: The VHDL code produced by `test1` for the full adder.

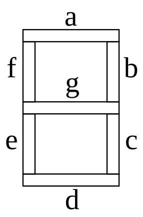

Figure 2.5: Digital display.

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 recursion and *connection patterns*. We will also see how we use numbers in Lava.

## 3.1 Recursion over Lists

A *bit adder* takes a pair of inputs. The first part is a carry bit, the second part is a binary number, represented as a list of 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 Lava by recursion over the list of bits. There are two cases. Either the list is empty, denoted as `[]`, and there is nothing to add. Or the list has at least one element `a`, and we can split the list up in two parts, `a`, the least significant bit, and `as`, the remaining bits, written `a:as`. In this case, we will use a half adder to add `a` and the carry, and *recursively* add the resulting carry to the rest of the binary number.

```
bitAdder (carryIn, [])   = ([], carryIn)

bitAdder (carryIn, a:as) = (sum:sums, carryOut)
  where
    (sum, carry)      = halfAdd (carryIn, a)
    (sums, carryOut) = bitAdder (carry, as)
```

A more complicated circuit is the circuit `adder` that takes a carry and a pair of binary numbers, and adds them up. This is called a binary adder. The recursive structure is almost the same, but we are doing *simultaneous* recursion over both binary numbers.

```
adder (carryIn, ([], []))     = ([], carryIn)
```

```
  adder (carryIn, (a:as, b:bs)) = (sum:sums, carryOut)
    where
      (sum, carry)     = fullAdd (carryIn, (a, b))
      (sums, carryOut) = adder (carry, (as, bs))
```

[Note: This adder is actually predefined in the module `Arithmetic`.]

### 3.1.1 Generating VHDL for a binary adder

To generate a VHDL netlist for the adder that we have just defined, we need to specify the *size* of the circuit, that we need to fix the lengths of its input lists. This is because we have written a *generic* circuit description using pattern matching over lists, but a netlist must have a fixed size. For example, to fix the lengths of the two binary numbers to be added to 4, we write

```
test2 = writeVhdlInputOutputNoClk "adder" adder
          (var "cin", (varList 4 "a", varList 4 "b"))
          (varList 4 "sum", var "cout")
```

Typing `test2` at the Lava prompt then produces the VHDL file shown in Figure 3.1. It is also possible to parameterise the definition with the adder size:

```
test3 n = writeVhdlInputOutputNoClk "adder" adder
               (var "cin", (varList n "a", varList n "b"))
               (varList n "sum", var "cout")
```

making it very easy to produce large netlists.

## 3.2 Connection Patterns

Looking at the two circuit definitions in the previous section, `bitAdder` and `adder`, we can see that they have a lot in common. Even though the *gates* that they use are different, their *structure* is very similar.

In Lava, we can capture these common structures in *connection patterns*. Connection patterns are higher-order functions that build circuits from other (smaller) circuits.

A very common connection pattern is the *serial* composition `serial` of two circuits (see figure 3.2). It is a circuit *parametrized* by two circuits `circ1` and `circ2`. This means that `serial circ1 circ2` is a circuit, which feeds its input `a` to `circ1`, connects the output `b` of it to the input of `circ2`, and results in that output `c`.

```
library ieee;

use ieee.std_logic_1164.all;

entity
  adder
is
port
  (


    cin : in std_logic
  ; a_0 : in std_logic
  ; a_1 : in std_logic
  ; a_2 : in std_logic
  ; a_3 : in std_logic
  ; b_0 : in std_logic
  ; b_1 : in std_logic
  ; b_2 : in std_logic
  ; b_3 : in std_logic


  ; sum_0 : out std_logic
  ; sum_1 : out std_logic
  ; sum_2 : out std_logic
  ; sum_3 : out std_logic
  ; cout : out std_logic
  );
end adder;

architecture
  structural
of
  adder
is
  signal w1 : std_logic;
  signal w2 : std_logic;
  signal w3 : std_logic;
  signal w4 : std_logic;
  signal w5 : std_logic;
  signal w6 : std_logic;
  signal w7 : std_logic;
  signal w8 : std_logic;
  signal w9 : std_logic;
  signal w10 : std_logic;
  signal w11 : std_logic;
  ...
  signal w28 : std_logic;
  signal w29 : std_logic;
begin
  c_w2      : entity work.wire  port map (cin, w2);
  c_w4      : entity work.wire  port map (a_0, w4);
     ...
  c_w29     : entity work.andG  port map (w21, w24, w29);
  c_w27     : entity work.xorG  port map (w28, w29, w27);


  c_sum_0   : entity work.wire  port map (w1, sum_0);
  c_sum_1   : entity work.wire  port map (w6, sum_1);
  c_sum_2   : entity work.wire  port map (w13, sum_2);
  c_sum_3   : entity work.wire  port map (w20, sum_3);
  c_cout    : entity work.wire  port map (w27, cout);
end structural;
```

Figure 3.1: The VHDL code produced for a 4-bit adder (with parts omitted for brevity).

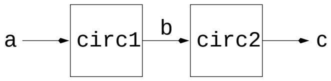

Figure 3.2: Serial composition of `circ1` and `circ2`.

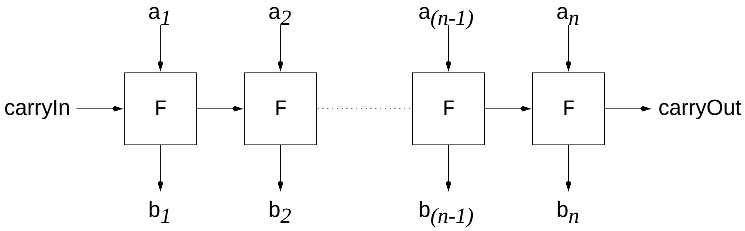

Figure 3.3: The pattern `row` $F$, connecting $n$ instances of $F$.

```
serial circ1 circ2 a = c
  where
    b = circ1 a
    c = circ2 b
```

More interesting connection patterns become possible when we consider recursive circuit structures. For example, instead of the half adder circuit in the `addBit` definition, we can plug in any other circuit. The result consists of a *row* of smaller circuits (see figure 3.3).

Here is how we define the `row` connection pattern.

```
row circ (carryIn, [])   = ([], carryIn)

row circ (carryIn, a:as) = (b:bs, carryOut)
  where
    (b, carry)     = circ (carryIn, a)
    (bs, carryOut) = row circ (carry, as)
```

Once we have made this definition, we do not need to use recursion anymore to define circuits of this specific pattern. Note that the definition of `row` assumes that the component, `circ`, has a pair as input and produces a pair as output. This was why we chose the type of `fullAdd` also to be of this form. Also, if the components are to fit together properly into a linear array, it is necessary that it be possible to connect the second output of one component to the first input of the next. However, the types are not constrained any further than this. Note also that `row` itself also produces a "pair-to-pair" circuit, as does the related connection pattern *column* (see exercises 3.9 and 3.10).

Here are alternative definitions of `bitAdder` and `adder`:

```
bitAdder' (carry, inps) = row halfAdd (carry, inps)
```

```
  adder'    (carry, inps) = row fullAdd (carry, inps)
```

It turns out that one can get quite far with surprisingly few connection patterns. The module Lava2000/Modules/Patterns.hs contains a few useful patterns (including row). Using these patterns can lead to very concise circuit descriptions that are still easy to read for those familiar with the patterns. It is also convenient to mix the "named wire" style, which we saw in the recursive definitions earlier, with the use of connection patterns.

Even shorter definitions of the same circuits are:

```
  bitAdder' = row halfAdd
  adder'    = row fullAdd
```

Note that the *type* of adder' is slightly different from adder, see exercise 3.3.

## 3.3 Arithmetic

In Lava, we can not only deal with low-level wire types like bits, and gates like and2 and xor2, but also with more abstract wire types and gates. One of these types is integers (and indeed the lowest level wires in our circuits carry either bits or integers).

On these integers, we have operations corresponding to abstract gates over integers. A list of these gates can be found in appendix A.

A simple circuit using these arithmetic gates is called numBreak. It takes a number as input, and has a pair of a bit and a number as output. The bit in the pair corresponds to the value of the first binary digit of the number; the resulting number is the input number divided by 2.

```
  numBreak num = (bit, num')
    where
      digit = imod (num, 2)
      bit   = int2bit digit
      num'  = idiv (num, 2)
```

The circuit i2b converts a number into a bit, by transforming a 0 into low, and any other number into high.

We can use this arithmetical circuit to build a circuit that converts a number into a binary number, that is, a list of bits. The circuit takes a parameter, corresponding to the size of the list it has to produce, and has as input the number that needs to be converted.

The converter int2bin converts an integer to a binary number. It has an extra *parameter*, which specifies the number of bits the binary number should have. Note again that parameters of circuits are different from inputs; int2bin is not really a circuit, but `int2bin 16` is. We define this circuit by recursion over the size of the binary number.

```
  int2bin 0 num = []

  int2bin n num = (bit:bits)
    where
      (bit,num') = numBreak num
      bits       = int2bin (n-1) num'
```

Here, the actual circuit input is `num`, which is the kind of integer that flows in a Lava circuit, and so has type `Signal Int`. Other arithmetic gates include `plus`, `times`, etc.

Here are some example simulations of these circuits:

```
  Main> simulate numBreak 7
  (high,3)

  Main> simulate (int2bin 3) 7
  [high, high, high]

  Main> simulate plus (3,4)
  7
```

At present, VHDL netlist generation supports only bit level operations. It will give an error if you try to generate VHDL for a circuit that operates on integers. However, the integers can still be useful! For example, you can use them in testing your arithmetic circuits. Let us wrap our binary adder up in suitable conversions:

```
wrapAdd n (a,b) = out
  where
    as = int2bin n a
    bs = int2bin n b
    (ss,c) = adder (low,(as,bs))
    out = bin2int (ss ++ [c])
```

We supply it with two `n`-bit inputs, which we produce from the integer inputs `a` and `b`. For the output, we stick the carry onto the *end* of the list of sum bits, since that list is *least significant bit first.* This is done by forming the singleton list `[c]` and *appending* that list to the end of the list `ss`. (`++` is the Haskell operator that appends two lists.) Having made a single binary number, we convert the result back into an integer. We would then expect the resulting circuit to behave rather like `plus`, but with a limit on the size of the inputs that it can deal with. Note that we must fix the size of the parameter `n` in order to get a circuit that can be simulated.

```
Main> simulate (wrapAdd 4) (3,5)
8
```

Perhaps you can figure out why we get

```
Main> simulate (wrapAdd 2) (3,5)
4
```

.

## 3.4 Exercises

3.1 Define a bit subtractor, called `bitSubber`, which takes a bit and a binary number as input, and subtracts the bit from the binary number.

3.2 Define a binary adder, called `adder2`, which does not take in a carry bit, and throws away the resulting carry.

3.3 What is the difference between `adder` and `adder'`? Hint: look at the types of the inputs.

3.4 Define a circuit `bin2int`, which converts a bit vector into an integer.

3.5 Define the circuit `zipp`, which takes a pair of list as inputs and produces a list of pairs, one by one grouped together.

```
Main> simulate zipp ([low,high,low],[high,high,low])
[(low,high),(high,high),(low,low)]
```

Also define the circuit `unzipp`, which is the inverse of `zipp`.

3.6 Define the circuit `pair`, which takes a list as input and produces a list of pairs, with the neighbours grouped together.

```
Main> simulate pair ([low,high,low,high,high,low])
[(low,high),(low,high),(high,low)]
```

Also define the circuit `unpair`, which is the inverse of `pair`.

3.7 Define a connection pattern called `par` which turns two circuits, each taking in one input and having one output, into one circuit taking in a pair of inputs and having a pair of outputs.

3.8 Define, using recursion, a binary multiplier. What is the recursive structure?

3.9 Looking at the definition of `row`, define a connection pattern called `column` which carries the right part of the input and the left part of the output through.

(*) Can you define `column` in terms of `row`?

3.10 Define a connection pattern called `grid`, which puts a number of copies of circuits in a matrix. The left parts of the inputs are carried through from left to right, and the right parts of the inputs and outputs are carried through from top to bottom.

Hint: think of a grid as a row of columns (or a column of rows).

3.11 Can you think of a useful circuit that makes use of the `grid` connection pattern?

3.12 Looking at the recursive definition of an adder, define a simple subtractor. It will only have to subtract smaller numbers from bigger numbers. Can you use any of the connection patterns described in this chapter to make a non-recursive description?

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

```
swapper (swap, (a, b)) = (x, y)
  where ...
```

3.14 Define a comparator, a circuit that takes in two binary numbers of equal length and tells you if the left one is less than or equal than the right one.

3.15 Implement a binary sorter. It takes as an input two binary numbers of equal length, and outputs them in the correct order.

# Chapter 4

# Verification

In this chapter we describe how we can define properties of circuits, and how we can formally verify these properties using a SAT-solver or model checker.

## 4.1 Simple Properties

The main kind of properties of circuits we deal with in Lava are so-called *safety properties*. These are properties which can be defined in such a way that they state that some condition is *always* true (or, equivalently, never false).

Here is an example; a property that checks that the outputs of a half adder are never both true.

```
  prop_HalfAddOutputNeverBothTrue (a, b) = ok
    where
      (sum, carry) = halfAdd (a, b)
      ok           = nand2 (sum, carry)
```

Note that this property looks pretty much like a normal circuit definition, and in fact it is.

The actual verification question is: does this property circuit always yield true, no matter what the input is? To answer the question, we use the Lava operation `satzoo`, which is a call to a satisfiability solver (a propositional theorem prover). To get access to this function, import the module `Satzoo`.

```
  Main> satzoo prop_HalfAddOutputNeverBothTrue
Satzoo: ...
real    0m0.005s
user    0m0.000s
sys     0m0.000s
```

```
(t=)
Valid.
```

This process works in the following way. Just as we can generate VHDL from a circuit description, we can also generate a logical formula representing the circuit. This logical formula is then given to an external theorem prover which will prove (or disprove) the validity of the formula. The result is then taken back into Lava.

Here is another example; we formulate that a full adder does not care about the order of the two one-bit arguments that are not the carry-in, but will always produce the same result. This property is in general called *commutativity*.

```
  prop_FullAddCommutative (c, (a, b)) = ok
    where
      out1 = fullAdd (c, (a, b))
      out2 = fullAdd (c, (b, a))
      ok   = out1 <==> out2
```

Note that, since we are not interested in the exact shape of the output of the two full adders, we can just give a name to the whole output, in this case `out1` and `out2`. Another thing to notice is that we use the general equality `<==>`. We can also use the circuit `equal` for that.

```
Main> satzoo prop_FullAddCommutative
Satzoo: ...
real    0m0.046s
user    0m0.000s
sys     0m0.002s
(t=)
Valid.
```

## 4.2 Quantification

The commutativity property is not only true for full adders, but also in general for binary adders. Here is how we state that property:

```
  prop_AdderCommutative (as, bs) = ok
    where
      out1 = adder2 (as, bs)
      out2 = adder2 (bs, as)
      ok   = out1 <==> out2
```

Note that we use the adder `adder2` we defined in exercise 3.2 (the answer is on page 86).

The problem is that this property holds for *all* circuit sizes, but we can only verify it for specific sizes! This is because it is very hard to verify properties automatically for all sizes.

So, instead of verifying it for all sizes, we will pick a specific size and verify the property for those. Thus, we define a new property, which is explicit about what size of input we want to verify the property.

```
prop_AdderCommutative_ForSize n =
  forAll (list n) $ \as ->
    forAll (list n) $ \bs ->
      prop_AdderCommutative (as, bs)
```

This property means: "for all lists of size $n$ called `as`, and for all lists of size $n$ called `bs`, the property that the adder is commutative holds for `(as, bs)` as input".

Now, we can verify the property using `satzoo`. We can of course do this for more than one size.

```
Main> satzoo (prop_AdderCommutative_ForSize 2)
Satzoo: ...
real    0m0.026s
user    0m0.001s
sys     0m0.001s
(t=)
Valid.

Main> satzoo (prop_AdderCommutative_ForSize 32)
Satzoo: ...
real    0m0.375s
user    0m0.089s
sys     0m0.002s
(t=)
Valid.
```

What actually happens behind the scences when you do verifications like these is that a file called `circuit.cnf` in the CNF (= conjunctive normal form) format read by *Satzoo* is produced in the directory `Verify`. You should do a small verification and then go into the directory `Verify` and look at the resulting file `circuit.cnf`. In the same directory, you will find the file `circuit.cnf.out` that shows what the satisfiability solver output when given `circuit.cnf`. (Note that the SAT-solver actually checks that the negation of the formula is unsatisfiable, leading to the `Valid` answer inside Lava.)

The expression `(prop_AdderCommutative_ForSize 32)` means the function `prop_AdderCommutative_ForSize` applied to the parameter 32. The result of this application is the circuit (of fixed size) that we want to verify with `satzoo`.

Leaving out the brackets instead means passing two different (and wrongly typed) arguments to `satzoo`. At this, the Haskell type checker complains:

```
Main> satzoo prop_AdderCommutative_ForSize 2
ERROR - Type error in application
*** Expression     : satzoo prop_AdderCommutative_ForSize 2
*** Term           : satzoo
*** Type           : d -> IO ProofResult
*** Does not match : a -> b -> c
```

## 4.3 General Properties

General properties are properties that are parametrized by one or more circuits. They can be defined just like connection patterns. Here is a general property that poses the question if the two given circuits are equivalent.

```
  prop_Equivalent circ1 circ2 a = ok
    where
      out1 = circ1 a
      out2 = circ2 a
      ok   = out1 <==> out2
```

You will likely use this kind of *equivalence checking* often. As an example, we can check that our own *full adder* (the one defined in the *Getting Started* chapter) is the same as the one built into Lava (in the `Arithmetic` module). To do this, you should add `import Arithmetic` to import that module. Now, the built-in *full adder* is also called `fullAdd`, so we need to distinguish it from ours by including the module name:

```
Main> satzoo (prop_Equivalent (Arithmetic.fullAdd) fullAdd)
Satzoo: ...
real    0m0.005s
user    0m0.001s
sys     0m0.002s
(t=)
Valid.
```

The following property checks if a given circuit is commutative.

```
  prop_Commutative circ (as, bs) = ok
    where
      out1 = circ (as, bs)
      out2 = circ (bs, as)
      ok   = out1 <==> out2
```

Of course, the circuits that one uses to instantiate these properties have to be of the right shape (type).

### 4.3.1 Using SMV

The other tool that you will be using (as a Lava backend) to do verification is Cadence SMV [4]. This a model checker, and so makes most sense when verifying sequential circuits (circuits with state holding elements). However, even for combinational circuits, SMV can be used. For example, to verify that the two *full adders* are equivalent in SMV, we write

```
Main> smv (prop_Equivalent (Arithmetic.fullAdd) fullAdd)
Smv: ... (t=0.00system)
Valid.
```

Now, the input file for SMV is `Verify/circuit.smv`.

```
-- Generated by Lava2000

MODULE main
VAR i0 : boolean;
VAR i1 : boolean;
VAR i2 : boolean;
DEFINE w5 := i0;
DEFINE w7 := i1;
DEFINE w8 := i2;
DEFINE w6 := !(w7 <-> w8);
DEFINE w4 := !(w5 <-> w6);
DEFINE w10 := !(w7 <-> w8);
DEFINE w9 := !(w5 <-> w10);
DEFINE w3 := !(w4 <-> w9);
DEFINE w2 := !(w3);
DEFINE w15 := w7 & w8;
DEFINE w16 := w5 & w6;
DEFINE w14 := !(w15 <-> w16);
DEFINE w18 := w7 & w8;
DEFINE w19 := w5 & w10;
DEFINE w17 := !(w18 <-> w19);
DEFINE w13 := !(w14 <-> w17);
DEFINE w12 := !(w13);
DEFINE w20 := 1;
DEFINE w11 := w12 & w20;
DEFINE w1 := w2 & w11;
SPEC AG w1
```

Here, we check the CTL formula `AG w1`, asking SMV to prove that the output of the comparison of the two circuits is always true. (This works both for combinational circuits (as here) and for sequential circuits, as we shall see later.)

## 4.4 Exercises

4.1 Take a look at the two bit sorter you defined in exercise 2.2. To verify that it is correct, two properties need to be true:

- The left part of the output is smaller than the right part of the output,
- The output of the circuit contains the same bits as the input (but possibly in a different order).

State these two properties separately, and verify them using `satzoo`.

4.2 Some properties are so easy to verify that we can actually do it by simulating them for all inputs (using `domain`). There are a few of these properties in this chapter. Verify them by testing them for all inputs. Can you think of other such easy-to-verify properties?

4.3 Check that the various adders in the previous chapter are all commutative, for sizes up to 16 bits. What happens if you try to prove that the subtractor is commutative?

4.4 Check that the subtractor you defined in the previous chapter is really a subtractor. How do you formulate your property; what is the "definition" of subtraction? Make sure you do not mess up the sizes of the binary numbers.

4.5 Define a general property that states that a given circuit is associative. An operator $\circ$ is associative, if for every $x, y, z$ it holds that $(x \circ y) \circ z = x \circ (y \circ z)$. Are all the adders associative?

4.6 Verify that the carry-save adder you defined in the previous chapter is equivalent to a binary adder. Be careful how you formulate your property, since the inputs do not have the same shape.

4.7 Prove that, for an adder and subtractor of your choice, it holds that $x + (y - z) = (x + y) - z$. What extra condition should hold for $y$ and $z$? How do you express that?

4.8 (Haskell) How would you proceed if you want to verify a property for all sizes between, say, 1 and n?

# Chapter 5

# Sequential Circuits

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

## 5.1 The Delay Component

A new component in sequential circuits is the *delay* component. It is a circuit with one parameter (the initial output of the delay) and one input, which becomes its output in the next clock cycle.

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 a delay component to remember the previous input.

```
edge inp = change
  where
    inp'   = delay low inp
    change = xor2 (inp, inp')
```

We can simulate a sequential circuit by using the operation `simulateSeq`. It needs a circuit and a list of inputs. The list of inputs is interpreted as the different inputs at each clock tick.

```
Main> simulateSeq edge [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 change = out
  where
    out' = delay low out
    out  = xor2 (change, out')
```

As we can see, the definition of `out'` is dependent on `out`, whose definition is dependent on `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 circuits, they are essential to implement any interesting behavior.

Simulating `toggle` gives:

```
Main> simulateSeq toggle [high, low, low, high]
[high, high, high, low]
```

## 5.2 Multiple Delays

We have seen how we can delay a signal *one* time instant, so that we can refer to the signal's previous value. Sometimes, we want to delay a signal multiple time instances. We can do this by defining a parametrized circuit, called `delayN`. It has two parameters, $n$, the number of delays to use, and `init`, the initial values of these delays.

We use recursion over $n$ to define this circuit.

```
delayN 0 init inp = inp

delayN n init inp = out
  where
    out  = delay init rest
    rest = delayN (n-1) init inp
```

A useful sequential circuit that we can implement using `delayN`, is called `puls`. It has no inputs, one output, and one parameter $n$. Its output is normally low, except on the $n$-th, $2n$-th, $3n$-th, ... clock tick, where it outputs high.

We implement the circuit by creating $n-1$ delay components in a row, initialized by `low`, ended with one delay component initialized by `high`.

```
puls n () = out
  where
    out  = delayN (n-1) low last
    last = delay high out
```

Note that we need to use a loop back here. This implementation is not optimal, in the sense that it uses too many delay components; see exercise 5.6.

Simulating `puls 3` gives:

```
Main> simulateSeq (puls 3) [(), (), (), (), (), (), ()]
[low, low, high, low, low, high, low]
```

## 5.3 Counters

An n-bit counter is a circuit that outputs an n-bit binary number at every clock tick, starting with 0, and increasing it by 1 every clock tick. We implement this by keeping an internal state, which is a binary number. The circuit takes one parameter, which indicates the number of bits to use, and has no inputs.

```
counter n () = number'
  where
    number'             = delay (zeroList n) number
    (number, carryOut) = bitAdder (high, number')
```

We use the function `zeroList`, which creates a list of $n$ zeros, denoting the initial value. Note that the delay component not only works for bits, but also for example for pairs of bits and lists (as in this case).

Simulating `counter` gives:

```
Main> simulateSeq (counter 3) [(), (), ()]
[[low, low, low], [high, low, low], [low, high, low]]
```

A variant on this circuit is the *up-counter*, which takes an input, which indicates if the number should increase or not. In this case, we want the desired increase to take effect immediately, so we output the number *before* we delay it.

```
counterUp n up = number
  where
    number'             = delay (zeroList n) number
    (number, carryOut) = bitAdder (up, number')
```

Simulating `counterUp` gives:

```
Main> simulateSeq (counterUp 3) [high, low, high]
[[high, low, low], [high, low, low], [low, high, low]]
```

## 5.4 Sequentialization

In chapter 3, we have seen a combinational binary adder. As an input, it takes two $n$-bit binary numbers, and adds them up. For large $n$, this circuit can get quite large, which means it takes more circuit area and consumes more power, and will need a lower clock frequency to work properly.

We can make use of the regularity in the circuit to make a small version of the circuit that however needs several clock cycles to compute the result. If we apply this technique on the binary adder, we obtain a *sequential adder*. It takes one new digit of both binary numbers at each clock cycle. This is sometimes called *bit serial*.

We can implement this by storing the carry as an internal state, so that the current carry-in of the circuit is the previous carry-out.

```
adderSeq (a,b) = sum
  where
    carryIn        = delay low carryOut
    (sum,carryOut) = fullAdd (carryIn, (a,b))
```

Simulating adderSeq gives:

```
Main> simulateSeq adderSeq [(high,low), (high,high), (low,high)]
[high, low, low]
```

Because we find that many sequential circuits have this structure, we define a *sequential* connection pattern, called rowSeq which builds a row of circuits, just like row, but interprets the row *over time*.

```
rowSeq circ inp = out
  where
    carryIn         = delay zero carryOut
    (out, carryOut) = circ (carryIn, inp)
```

Worth noting is that we make use of the generic delay and zero component here. The structure is exactly the same as in the sequential adder.

Recalling the definition of a binary adder in terms of row, we can repeat it and implement a sequential adder in terms of rowSeq:

```
adder'    = row    fullAdd  -- combinational
adderSeq' = rowSeq fullAdd  -- sequential
```

In this way, using a connection pattern to define a combinational circuit helps us to define the sequential version of the circuit.

## 5.5 Variations on rowSeq

The sequential row connection pattern is sometimes useful, but certainly not always. If we use it to implement a sequential adder, as we did, we can also use it to add up "infinitely big" binary numbers. The addition never ends, so we can never start over adding two new numbers.

Therefore, it is handy to have a connection pattern, called `rowSeqReset`, which takes one extra input `reset`. When `reset` is high, the internal carry state will be reset to `zero`.

```
rowSeqReset circ (reset,inp) = out
  where
    carryIn         = delay zero carry
    carry           = mux (reset, (carryOut, zero))
    (out, carryOut) = circ (carryIn, inp)
```

We use the standard multiplexer component `mux` here, which chooses the left or right component of an input pair, depending on if the first incoming signal is low or high, respectively.

Now we can define a resettable sequential adder `adderSeqReset` as follows:

```
adderSeqReset = rowSeqReset fullAdd
```

Very often, it is the case that the internal carry state has to be reset periodically, that is, on every $n$-th, $2n$-th, ... clock tick. Therefore, we create a third sequential row variation, which takes a parameter $n$, which indicates the reset period.

```
rowSeqPeriod n circ inp = out
  where
    reset = puls n ()
    out   = rowSeqReset circ (reset, inp)
```

Now we can define a sequential adder `adderSeqPeriod` adding $n$-bit numbers as follows:

```
adderSeqPeriod n = rowSeqPeriod n fullAdd
```

## 5.6 Exercises

5.1 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 Lava and generate VHDL.

5.2 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.

5.3 Implement a clocked delay component `delayClk`. It has one parameter, the initial state, and it has an extra input `clk`. Only when `clk` is high, the output changes to the state and the state changes to the current input.

5.4 Define a circuit called `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, then the output stays low forever.

5.5 Define three different circuits that output high only on every 6th clock tick (so it happens on the 6th, 12th, 18th, ... etc.). Use 6 delay elements in the first circuit, 5 delay elements in the second, and 3 in the last.

Is it possible to define this circuit with less than 3 bit-level delay elements?

5.6 Define a puls generator `puls2` which has a parameter $k$, and generates a puls every $2^k$-th clock tick. Your design should use a minimal number of delay components (how many is that?).

5.7 Define an up-down counter. The counter gets a pair of inputs. If the left input is high, it counts up. Otherwise, if the right input is high, it counts down. Otherwise, the state stays the same.

5.8 Define a 0-to-9 counter. The counter has no inputs, and a 4-bit number as output. Initially, the output starts at 0, and increments at every clock tick, but after the output 9, it returns to 0.

Connect the display from exercise 2.6 to your counter.

5.9 Define a *synchronizer*, which has two inputs `go1` and `go2`, and one output `go`. The output only becomes true when both `go1` and `go2` have been high in the past or are high now since the last `go`. .

Here is an example simulation:

```
Main> simulateSeq synchronize
                  [(low,high),(high,low),(high,high),
                         (high,low),(low,low),(low,high)]
[low, high, high, low, low, high]
```

5.10 Define a circuit called `outputList`, which has one parameter, a list of values, no inputs, and one output. The circuit outputs the elements in the parameter list one by one at every clock tick repeatedly. Here is an example simulation:

```
Main> simulateSeq (outputList [low, low, high])
                                       [(),(),(),(),(),()]
[low, low, high, low, low, high]
```

# Chapter 6

# Sequential Verification

In this chapter we describe how we can verify properties of sequential circuits. We restrict ourselves to sequential safety properties.

## 6.1 Sequential Safety Properties

Let us take a look at how to define properties about sequential circuits. In principle, we can use the same techniques as we did with combinational circuits. Let us take a look at some examples.

Here is how we can compare the two sequential adders from section 5.4.

```
prop_SameAdderSeq inp = ok
  where
    out1 = adderSeq inp
    out2 = adderSeq' inp
    ok   = out1 <==> out2
```

Here is another example; the composition of `edge` and `toggle` from section 5.1 gives the identity circuit. This means that the input is the same as the output.

```
prop_ToggleEdgeIdentity inp = ok
  where
    mid = toggle inp
    out = edge mid
    ok  = out <==> inp
```

The properties we can describe in this way are called *sequential safety properties*. Recall that safety properties are properties which can be described as a circuit with one output, which should always be true (or never be false) for the property to hold.

Examples of properties which are not safety properties are for example *liveness* properties. These can assert that a certain condition must hold at some point in the future, for example.

## 6.2 Sequential Logic

Apart from the techniques we used to define combinational properties, there are also special techniques we can apply to define sequential properties.

- When we want to refer to values of signals at different time instances, we can use a `delay` to get access to previous values. But be careful about what initial value you choose for this use of delay.
- When we want a certain property only to be true when a certain condition holds, which does not necessarily hold all the time, we can use logical implication. Implication is implemented by the Lava gate `impl`, and also by the binary operator `==>`.

Here is an example. Suppose we want to define the following property about the `toggle` circuit: "if the input is high, then the current output is different from the previous output".

The way we define this in Lava is:

```
prop_ToggleTogglesWhenHigh inp = ok
  where
    out    = toggle inp
    out'   = delay low out
    change = xor2 (out, out')
    ok     = inp ==> change
```

First, we compute the output `out` from `toggle`. Then, we use a delay component to get access to the previous output `out'`. We define the situation `change` in which these outputs differ. And then we say: "if the input is `high`, then the outputs differ".

## 6.3 Verification

After defining these properties, we would like to formally verify them. Verification of a sequential property means that we have to prove that the property holds at all times. In Lava, we do this by *induction* over time. It works as follows.

Firstly, we have to do the *base case*: proving that the property holds for the *first* time instance. Since looking at just one time instance does not involve time at all, we can use the same techniques as we did in the combinational case.

Then, we do the inductive *step*. We want to prove that *if* the property holds at time $t$, it also holds at time $t+1$. We do this as follows: we create an *arbitrary* time instance by filling the states of the circuits with fresh variables. Then, we run the circuit once on that state, obtaining an output and new state values. Then we assert that the output is true, and run the circuit on the new state values. Finally, we need to prove that the new output is true.

After proving the base case and the inductive step, we have proved our property. Here is what happens in Lava:

```
Main> verify prop_ToggleEdgeIdentity
Proving: base 1 ... Valid.
Proving: step 1 ... Valid.
--
Result: Valid.

Lava> verify prop_ToggleTogglesWhenHigh
Proving: base 1 ... Valid.
Proving: step 1 ... Valid.
--
Result: Valid.
```

We give a more detailed explanation of induction in the next section.

## 6.4 Induction

To perform induction on a Lava property, we convert it to a logical formula relating input '$\mathsf{inp}$' and old state variables '$\mathsf{q_{old}}$' to output '$\mathsf{ok}$' and new state variables '$\mathsf{q_{new}}$'. Whenever we use a signal-level delay component in a circuit or property, we introduce one state variable.

In this translation, we also introduce a special input, called '$\mathsf{init}$', which is true only in the first time instance. So, after we have translated the property, we have a logical formula of the following form:

$$T(\mathsf{init}, \mathsf{inp}, \mathsf{q_{old}}, \mathsf{q_{new}}, \mathsf{ok})$$

This formula is usually called the *transition relation*.

A very simple way to prove that the output '$\mathsf{ok}$' is always true, would be to try proving the following:

$$T(\mathsf{init}, \mathsf{inp}, \mathsf{q_{old}}, \mathsf{q_{new}}, \mathsf{ok}) \Rightarrow \mathsf{ok} \tag{6.1}$$

Unfortunately however, this method does not work very often, because even when the property is always true in any run of the circuit, it might not be true in every possible configuration of the state variables. This is why we use induction.

First, we prove the *base case*, that is: 'ok' is true at the first time instance. In this case, we know that the variable 'init' is true, so we prove:

$$T(\mathbf{true}, \mathsf{inp}, \mathsf{q_{any}}, \mathsf{q_{new}}, \mathsf{ok}) \Rightarrow \mathsf{ok} \tag{6.2}$$

This is usually easy, since initially, we know the values of the state variables.

Then, we prove the *induction step*, that is: if 'ok' is true at time $t$, it is also true at time $t + 1$. So, we are looking at two time instances of the property.

$$\left.\begin{array}{r} T(\mathsf{init}_1, \mathsf{inp}_1, \mathsf{q}_1, \mathsf{q}_2, \mathbf{true}) \\ T(\mathbf{false}, \mathsf{inp}_2, \mathsf{q}_2, \mathsf{q}_3, \mathsf{ok}_2) \end{array}\right\} \Rightarrow \mathsf{ok}_2 \tag{6.3}$$

Note how we connect the different time instances 1 and 2 by reuse of the state variables '$\mathsf{q}_2$' as new states in the first time instance and as old states in the second time instance. Also note that we use **false** for the value of 'init' in the second time instance, because we know it is not the initial time instance. And we use **true** for the value of ok in the first time, since we may assume that the induction hypothesis holds.

If we have proven the two formulas 6.2 and 6.3, then we know that 'ok' must be true at all time instances. This is the basic notion of induction.

## 6.5 Induction With Depth

Unfortunately, the method of induction mentioned in the previous section is not *complete*. This means that there are properties which are true, which cannot be proven by simple induction.

Here is an example: Consider the `toggle` circuit from section 5.1 and the `puls` circuit from section 5.2. We might want to verify that these circuits do exactly the opposite if `toggle` always has a high input, and `puls` has a period of 2.

```
prop_Toggle_vs_Puls () = ok
  where
    out1 = toggle high
    out2 = puls 2 ()
    ok   = inv (out1 <==> out2)
```

This cannot be proven by normal induction, since the `puls` circuit has two delay components in a row, so it is not enough to look at two time instances at a time.

So, instead, we will look at more time instances in the induction proof. We introduce the concept of *induction with depth k*, which means that the base

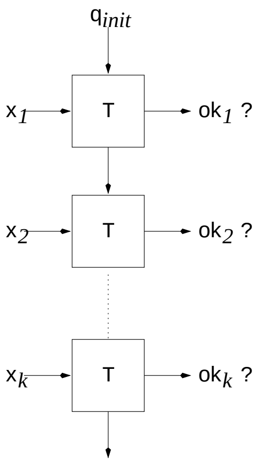

Figure 6.1: Base case for induction with depth $k$.

case proves that the first $k$ steps are okay, and the step case may assume that a sequence of $k$ steps went okay, in order to prove that the $k+1$-th step is okay.

Here is the concrete formula for the base case (see also figure 6.1):

$$\left.\begin{array}{r} T(\mathbf{true}, \mathsf{inp}_1, \mathsf{q}_1, \mathsf{q}_2, \mathsf{ok}_1) \\ T(\mathbf{false}, \mathsf{inp}_2, \mathsf{q}_2, \mathsf{q}_3, \mathsf{ok}_2) \\ \ldots \\ T(\mathbf{false}, \mathsf{inp}_k, \mathsf{q}_k, \mathsf{q}_{k+1}, \mathsf{ok}_k) \end{array}\right\} \Rightarrow \mathsf{ok}_1,\ \mathsf{ok}_2,\ \ldots,\ \mathsf{ok}_k \tag{6.4}$$

Note that we use the same trick of reusing the state variables of consecutive times to line up the time instances. Here is the concrete formula for the step case (see also figure 6.2):

$$\left.\begin{array}{r} T(\mathsf{init}_1, \mathsf{inp}_1, \mathsf{q}_1, \mathsf{q}_2, \mathbf{true}) \\ T(\mathbf{false}, \mathsf{inp}_2, \mathsf{q}_2, \mathsf{q}_3, \mathbf{true}) \\ \ldots \\ T(\mathbf{false}, \mathsf{inp}_k, \mathsf{q}_k, \mathsf{q}_{k+1}, \mathbf{true}) \\ T(\mathbf{false}, \mathsf{inp}_{k+1}, \mathsf{q}_{k+1}, \mathsf{q}_{k+2}, \mathsf{ok}_{k+1}) \end{array}\right\} \Rightarrow \mathsf{ok}_{k+1} \tag{6.5}$$

So, for any depth $k$, if we can prove the formulas 6.4 and 6.5, we have proved that 'ok' holds at every time instance. Note that if we choose $k = 1$, then we are back to normal induction again.

Here is what happens when we verify `prop_Toggle_vs_Puls` in Lava:

```
Main> verify prop_Toggle_vs_Puls
Prover: base 1 ... Valid.
Prover: step 1 ... Falsifiable.
Prover: base 2 ... Valid.
```

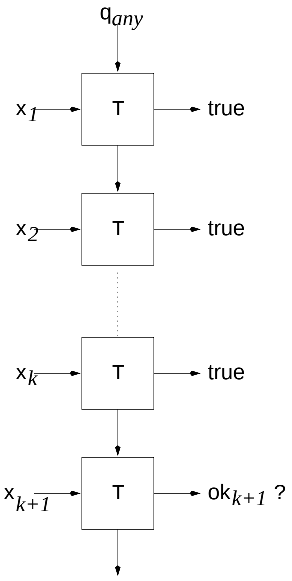

Figure 6.2: Inductive step for induction with depth $k$.

```
Prover: step 2 ... Valid.
--
Result: Valid.
```

So, the verifier realizes that induction depth 1 is not enough for the step to go through, and increases the induction depth automatically. It will keep increasing the depth until either the base case turns out to be false, or until it manages to prove both the base case and the step case.

If we want to specify a specific depth to do the induction for, we can use the operation `verifyWith`, which takes an extra list of verify options.

```
Main> verifyWith [Depth 2] prop_Toggle_vs_Puls
Prover: base 2 ... Valid.
Prover: step 2 ... Valid.
--
Result: Valid.
```

The operation `verify` is actually just a short-hand for `verifyWith [Depth 1, Increasing]`. With the option `Depth`, one can specify the induction depth. `Increasing` means that it will keep increasing the depth until it proves or disproves the property.

## 6.6 Induction With Restricted States

Unfortunately, even induction with depth is not a complete method. This means that there exists properies which are always true, but for which there exists no $k$ such that the property can be proven by induction with depth $k$.

An example of such a property is to check if a periodic sequential adder of period 2 is equivalent to a resettable adder which we reset every second clock tick.

```
prop_AdderPeriod2 ab = ok
  where
    sum1 = adderSeqPeriod 2 ab
    two  = delay low (inv two)  -- 010101...
    sum2 = adderSeqReset (two, ab)
    ok   = sum1 <==> sum2
```

Verifying this property results in an infinite loop:

```
Main> verify prop_AdderPeriod2
Prover: base 1 ... Valid.
Prover: step 1 ... Falsifiable.
Prover: base 2 ... Valid.
Prover: step 2 ... Falsifiable.
...
```

The problem is that there exist a lot of state variable configurations that never occur when we run the circuit, but are logically possible. In some cases, these so-called *unreachable states* mess up the induction proof. Even assuming that the property we want to prove is true for a very large number $k$ of consecutive running steps (like we do in the induction step) is not enough to ensure we are in a reachable state. The reason for this is that we might be running around in the unreachable states in circles for these $k$ steps, so increasing $k$ does not help.

Instead, we will strengthen the induction step by saying that all $k+1$ states we visit in the formula must be distinct. In this way, we ensure that we are not running around in circles.

The new formula for the inductive step becomes:

$$\left.\begin{array}{r} T(\mathsf{init}_1, \mathsf{inp}_1, \mathsf{q}_1, \mathsf{q}_2, \mathbf{true}) \\ T(\mathbf{false}, \mathsf{inp}_2, \mathsf{q}_2, \mathsf{q}_3, \mathbf{true}) \\ \ldots \\ T(\mathbf{false}, \mathsf{inp}_k, \mathsf{q}_k, \mathsf{q}_{k+1}, \mathbf{true}) \\ T(\mathbf{false}, \mathsf{inp}_{k+1}, \mathsf{q}_{k+1}, \mathsf{q}_{k+2}, \mathsf{ok}_{k+1}) \\ \mathsf{q}_1 \neq \mathsf{q}_2,\ \mathsf{q}_1 \neq \mathsf{q}_3,\ \ldots,\ \mathsf{q}_{k-1} \neq \mathsf{q}_{k+1},\ \mathsf{q}_k \neq \mathsf{q}_{k+1} \end{array}\right\} \Rightarrow \mathsf{ok}_{k+1} \qquad (6.6)$$

For this method, proving formulas 6.4 and 6.6 for some $k$ is enough to prove the ‘ok’ holds at all time instances. Moreover, this is a complete method! This means that, if the property holds, there is always a $k$ such that we can prove it by induction with depth $k$ with restricted states.

To use induction with restricted states in Lava, we can use the option `RestrictStates`:

```
Main> verifyWith [RestrictStates,Increasing] prop_AdderPeriod2
Proving: base 1 ... Valid.
Proving: step 1 ... Falsifiable.
...
Proving: base 5 ... Valid.
Proving: step 5 ... Valid.
--
Result: Valid.
```

We needed induction depth 5 for this property. Note that we used the option `Increasing` also, otherwise the verification would have stopped at depth 1.

## 6.7 Exercises

6.1 Why is simulation not enough to do sequential verification?

6.2 Verify that the `edge` circuit and the circuit `evenSoFar` from exercise 5.1 always have opposite outputs if fed with the same inputs.

6.3 Verify that the three different implementations of a puls generator with period 6 in exercise 5.5 are equivalent. What is the induction depth that is needed?

6.4 Verify the obvious relationship between the `puls` circuit and the `puls2` circuit from exercise 5.6, for different values of $k$. What is the induction depth that is needed?

6.5 Verify that the up-part of the up-down counter you defined in exercie 5.7 is equivalent to the up-counter from section 5.3. Do this for different values of $n$.

6.6 Define and verify the following property: "if the input to `toggle` is the same twice in a row, then the current output is the same as the output two steps ago".

6.7 Consider the following general property: "As long as $A$ holds, then $B$ must hold". How would you define such a property? Hint: use the `always` circuit from 5.4.

6.8 Show that doing induction with depth 1 amounts to normal induction.

6.9 (*) Show that doing induction with depth $k$ is *sound*, that is, if we have proven the base case and the inductive step, then we have really proven that the property always holds.

6.10 (**) Show that doing induction with depth $k$ and restricted states is *sound.* You may use the fact that exercise 6.9 holds.

6.11 (**) Show that doing induction with depth $k$ and restricted states is *complete.*

# Chapter 7

# Time Transformations

In this chapter, we will see some techniques with which we can compare circuits that operate at different clock rates.

## 7.1 Timing Issues

So far, when we were comparing two circuits, we always assumed that they consumed their inputs and produced their outputs at the same rate. Let us take a look at an example where this is not the case: comparing a sequential adder against a combinational adder.

The sequential adder (see figure 7.1) takes in a pair of bits every clock tick, and outputs the sum, and remembers the carry for the next clock cycle. The carry is reset every $n$-th clock tick. Here is how we defined it:

```
adderSeqPeriod n =
  rowSeqPeriod n fullAdd
```

The combinational adder (see figure 7.2) takes in two $n$-bit binary numbers and produces the sum as a $n$-bit binary number in one clock tick. Here is how we define it:

```
adderCom abs = sum
  where
    (sum, carryOut) = row fullAdd (low, abs)
```

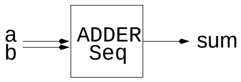

Figure 7.1: A sequential adder.

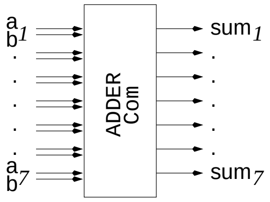

Figure 7.2: A combinational adder.

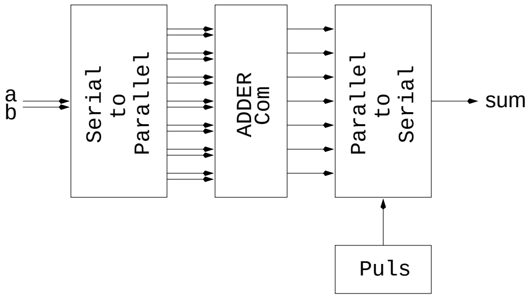

Figure 7.3: The slowed down combinational adder.

For convenience, we abstract away from the carry.

There are two basic methods for comparing these two circuits.

The first method involves slowing down the combinational adder, so that it takes more clock ticks to calculate the sum. So instead of taking $n$ pairs of bits at a time, it takes them in one-by-one, and when it has gotten all of them, it outputs the sums one-by-one. The circuits now operate at the same rate, and can be compared by conventional methods.

The second method involves speeding up the sequential adder, so that it computes several results in one clock tick. So instead of taking in one pair of bits at a time, it takes in $n$ pairs of bits, and produces $n$ sums in one clock cycle.

## 7.2 Slowing Down

The first technique we describe *slows down* the combinational circuit. So, instead of computing everything in one clock tick, we force it to take $n$ clock ticks instead. We do this by transforming the circuit into a circuit that looks just like the sequential version: it takes one input and produces one output at a time (see figure 7.3).

Since the inputs come in one-by-one, we have to wait for $n$ clock ticks until we have the full input available for the circuit. This is done by the *serial to parallel converter* (see figure 7.4). We can implement this component as follows:

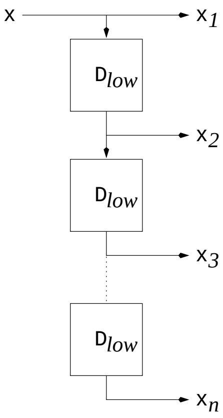

Figure 7.4: A serial to parallel converter.

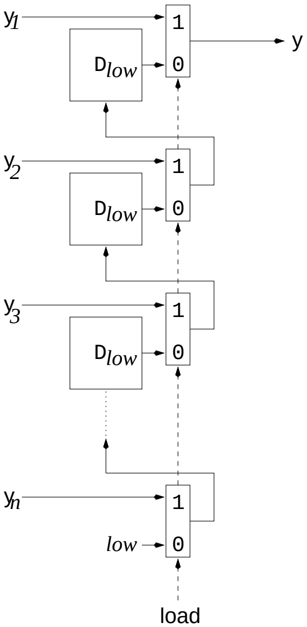

Figure 7.5: A parallel to serial converter.

```
serialToParallel 1 inp = [inp]

serialToParallel n inp = inp : rest
  where
    inp' = delay zero inp
    rest = serialToParallel (n-1) inp'
```

Then we have to take care of the outputs. At every clock tick, the combinational circuit produces $n$ outputs, but they only make sense on every $n$-th, $2n$-th, ... clock tick, because then we have the right input. Therefore, we need to add a component on the outputs that spreads out the outputs of the important clock ticks over the other clock ticks. This is done by the *parallel to serial converter* (see figure 7.5). We can implement this component as follows:

```
parallelToSerial (load, [inp]) = out
  where
    out  = mux (load, (low, inp))

parallelToSerial (load, inp:inps) = out
  where
    from = parallelToSerial (load, inps)
    prev = delay low from
    out  = mux (load, (prev, inp))
```

Then, we can put these components together in a new sequential adder:

```
adderSlowedDown n ab = sum
  where
    abs  = serialToParallel n ab
    sums = adderCom abs
    load = puls n ()
    sum  = parallelToSerial (load, sums)
```

The `load` input to the parallel to serial converter is a puls with period $n$. Let us take a look at how this sequential adder adds up binary numbers for $n = 4$.

| clock | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| input | $ab_1$ | $ab_2$ | $ab_3$ | $ab_4$ | $ab'_1$ | $ab'_2$ | $ab'_3$ | $ab'_4$ | $ab''_1$ |
| output | 0 | 0 | 0 | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s'_1$ | $s'_2$ |

As we can see, the results $s_i$ are delayed by $n - 1$ clock ticks. This is of course because the result is computed at the $n$-th, $2n$-th, ... clock tick. So, when we compare this with the original sequential adder, we have to slow the output of that one down with $n - 1$ delay components. Here is the property:

```
prop_AdderSeqSlowedDown n ab = ok
  where
```

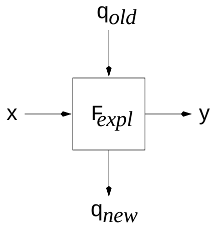

Figure 7.6: A combinational circuit with explicit states $\mathsf{q_{old}}$ and $\mathsf{q_{new}}$.

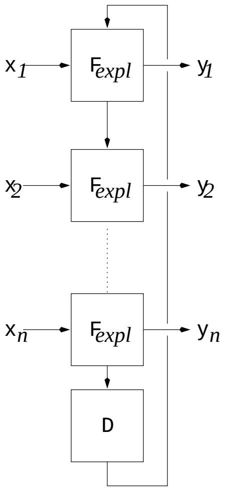

Figure 7.7: A time transformed sequential circuit $F$.

```
sum1  = adderSeqPeriod n ab
sum1' = delayN (n-1) low sum1
sum2  = adderSlowedDown n ab
ok    = sum1' <==> sum2
```

Unfortunately, this way of specifying the property introduces a lot of extra logic, and moreover, extra state. This makes the verification of these kind of properties very hard. In particular, the induction methods need an extremely high induction depth. In the next section, we will see a simpler and more direct method for specifying retiming properties.

## 7.3 Speeding Up

Another technique for retiming works as follows. Instead of slowing down the combinational circuit, we speed up the sequential circuit. Unfortunately, this cannot be done by adding retiming components around the circuit. Instead, we *transform* the circuit into another circuit. This is done by a built-in Lava operation, called `timeTransform`.

The idea is that we make the state of the sequential circuit explicit by turning a sequential circuit $F$ into a combinational circuit $F_{\mathsf{expl}}$, that takes in the old state as an extra input, and has the new state as an extra output (see figure 7.6).

The next step is to create a column of $F_{\mathsf{expl}}$, where we thread the states through as carry. The last step is to make the state implicit again by adding delay components and a loop back (see figure 7.7).

All this is implemented by Lava's primitive operation `timeTransform`. So, we can make a new adder from the sequential adder, by using time transformation:

```
adderSpedUp abs = sums
  where
    sums = timeTransform (adderSeqPeriod n) abs
    n    = length abs
```

The function `length` computes the length of a list, so that we know what period the sequential adder requires.

The property of comparing the two different adders now looks as follows:

```
prop_AdderSeqSpedUp abs = ok
  where
    sum1 = adderSpedUp abs
    sum2 = adderCom abs
    ok   = sum1 <==> sum2
```

Because this is a property that has a list as an input, we need to be explicit about the length of the list:

```
prop_AdderSeqSpedUp_ForSize n =
  forAll (list n) $ \abs ->
    prop_AdderSeqSpedUp abs
```

Verifying this by induction is easy, and needs induction depth 2 for any $n$.

## 7.4 Exercises

7.1 Consider the following circuit:

```
highLow () = [high, low]
```

Verify that the circuit `toggle` behaves twice as slow as this circuit if its input is always high. Do this by slowing down and speeding up.

7.2 What goes wrong when we try using the slowing down method for comparing two *sequential* circuits that operate at different rate? Also see exercise 7.5.

Hint: what happens to the state of a circuit that is slowed down?

7.3 Does the speeding up method work when we use it for comparing two sequential circuits that operate at different rate?

7.4 Design a property connection pattern that verifies two circuits that operate at different rates equivalent. You may decide yourself what method to choose.

7.5 Can you find a method to fix the problem in exercise 7.2?

Hint: use clocked delays (see exercise 5.3).

# Chapter 8

# More connection patterns

In this chapter, we first review some standard connection patterns, and then consider the problem of describing *tree shaped circuits* and *butterfly circuits*. These are common circuit structures in digital signal processing.

## 8.1 Connection patterns revisited

In an earlier chapter, we saw the `serial` connection pattern, which connects two circuits in series. It is convenient to have an infix version, so that we can write `f ->- g`, instead of `serial f g`, see figure 8.1. Note that serial composition is associative:

```
f ->- (g ->- h)   ===   (f ->- g) ->- h
```

Sometimes we want to compose a list of circuits. We call this `compose`.

```
compose []           inp = inp
compose (circ:circs) inp = out
  where
    x   = circ inp
    out = compose circs x
```

Note that we could have written this definition in a different style, using the serial connection pattern.

```
compose1 []           inp = inp
compose1 (circ:circs) inp = out
  where
    out = (circ ->- compose1 circs) inp
```

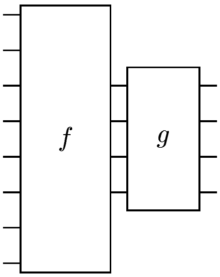

Figure 8.1: `f ->- g`

We could go even further and drop the circuit inputs (`inp`) from each side of the definitions. The identity circuit (which just returns its input) is written `id`. This is a definite change of style to one in which the emphasis is on connection patterns.

```
compose2 []           = id
compose2 (circ:circs) = circ ->- compose2 circs
```

All of these styles are equally good, and the choice is really just a matter of taste. In fact it is quite convenient to be able to mix styles, sometimes choosing one and sometimes the other.

Out of `compose`, we can easily make a connection pattern, called `composeN`, that composes several copies of the same circuit in sequence.

```
composeN n circ = compose (replicate n circ)

Main> simulateSeq (composeN 5 inc) [0,2,4,6]
[5,7,9,11]
```

Here `inc` is the circuit that adds one to its integer input.

We also saw the `par` connection pattern: `par f g` takes a pair of inputs, passing the first to `f` and the second to `g`, and combining the results into a pair. The infix version of `par f g` is written `f -|- g`.

A version of `par` that "does" `f` to the first half of a list and `g` to the second half also turns out to be useful. We call this pattern `parl`. First, we define a helper function, `halveList`, which divides a list in two.

```
halveList inps = (left,right)
  where
    left  = take half inps
    right = drop half inps
    half  = length inps `div` 2
```

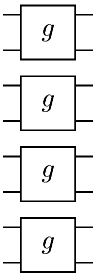

Figure 8.2: `map g`

```
Main> simulate halveList [high,low,high,low]
([high,low],[high,low])
```

Then, we define the circuit `append`, which takes a pair of lists of length $m$ and $n$, and joins them together (or *concatenates* them), to give a list of length $m+n$. This circuit is defined in terms of Haskell's built-in infix list concatenate operator (`++`).

```
append (a,b) = a ++ b
```

Lastly, we define `parl`:

```
parl circ1 circ2 =
  halveList ->- (circ1 -|- circ2) ->- append

Main> simulate (parl reverse id) [1..16]
[8,7,6,5,4,3,2,1,9,10,11,12,13,14,15,16]
```

Sometimes, we want to perform an operation of each element of a list of signals or bus. For this we use the connection pattern `map`, which you will have seen if you have used a functional programming language. For example, `map inv` inverts each of a list of bits.

```
Main> simulate (map inv) [high, low, high, low]
[low,high,low,high]
```

Buses need not contain only lists of bits. They can be more structured, so that our circuit descriptions can match the *logical* structure of the circuit. For example, the circuit `map fullAdd` makes perfect sense.

```
Main> simulate (map fullAdd)
        [(low,(high,low)),(high,(high,high)),(low,(high,high))]
[(high,low),(high,high),(low,high)]
```

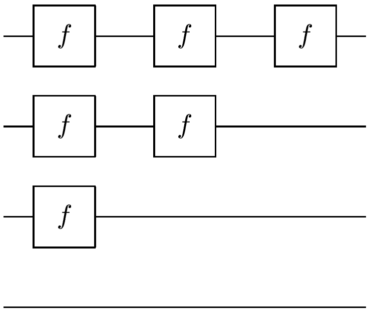

Figure 8.3: `tri f`

Figure 8.2 shows a `map` in the case where the input is a 4-list (of pairs or 2-lists).

Strangely enough, the connection pattern that places zero copies of a circuit on the first signal in a bus, one copy on the next, two on the next, and so on, is one that arises often in hardware design. It is a sort of mixture of `map` and `composeN`. We call it `tri` for *triangle.* You should understand why when you look at the diagram in figure 8.3. We leave the definition of `tri` as exercise 8.3.

An example of the use of triangle is

```
Main> simulate (tri inc) (replicate 10 0)
[0,1,2,3,4,5,6,7,8,9]
```

The connection patterns that we have seen in this section are all useful in many different kinds of circuits. Now let us consider how to describe tree shaped circuits.

## 8.2 Tree shaped circuits

Circuits in the shape of trees, like that shown in figure 8.4, can be used to systematically apply a function that combines data values together to a collection of data. A binary tree circuit first combines each half of the input values, using two smaller trees and then combines the two remaining results. One example of such a circuit is an adder tree that adds up a list of numhers.

The outline of the recursive definition of a tree connection pattern is:

```
tree circ [inp] = ... inp ...
tree circ inps  = ... tree circ ... tree circ ... circ ... inps
```

We call the parameter `circ` the *component circuit.* The first line in this outline defines what should be done when we get down to the base case of the recursion. The second line should use two copies of `tree circ` and combine their results

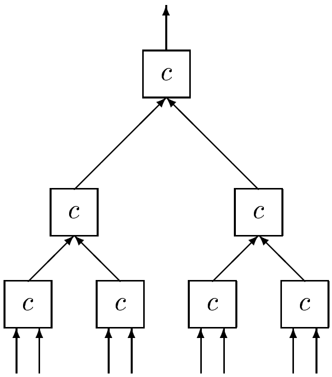

Figure 8.4: A tree shaped circuit

using `circ`. Exactly how these definitions should look depends partly on what the component `circ` looks like, and in particular on its type.

For example, if `circ` is a binary function taking a pair of inputs and returning a single output, then it makes sense to make the following definition of a binary tree connection pattern, `binTree`.

```
binTree circ [inp] = inp
binTree circ inps  =
  (halveList ->- (binTree circ -|- binTree circ) ->- circ) inps
```

This gives the behaviour that we expect: a binary tree of `circ` components gets built.

An example use of a tree connection pattern is when we want to build a circuit that adds up a lot of numbers. One way of doing this to make a so-called *adder tree*. To do this, we need a binary adder that adds two $n$ bit numbers, to give an $n + 1$ bit number. This means that we must include the carry out in the result. The resulting adder is therefore slightly different from those that we saw earlier. We call it `binAdder`.

```
binAdder (as, bs) = cs ++ [carryOut]
  where
    (cs, carryOut) = adder (low, (as, bs))
```

And here is the definition of our adder tree `addTree`:

```
addTree = binTree binAdder
```

To test it, we wrap the circuit in converters from integer to binary and back.

```
wrapAddTree n =
  map (int2bin n) ->- addTree ->- bin2int
```

```
Main> simulate (wrapAddTree 8) [3,4,5,6,10,9,8,7]
52
```

Beware, this adder tree works only for input lists whose length is a power of two. Exercise 8.5 asks you to define an adder tree that works for any size.

## 8.3 Describing Butterfly Circuits

Butterfly circuits are circuits with a particular recursive structure. Figures 8.6 and 8.7 show two such circuits and also indicate their recursive structures by showing, by means of dotted boxes, where to find sub-circuits that themselves have the same recursive structure. It turns out that these two circuits are in fact equivalent: the same network of components can be recursively described in two completely different ways. And indeed it turns out that there are many more ways to describe the same network. We will study some of them.

Butterfly circuits are used for example to build routing networks from switches, and in building efficient sorting circuits. Perhaps the best known butterfly-like circuit is the standard Cooley-Tukey algorithm [5] for computing the Fast Fourier Transform (FFT). We will not consider the FFT here. The twiddle-factors complicate matters a bit. The circuit is not quite as uniform as those that we consider. However, the interested reader is referred to [3], which shows how to describe and compare various FFT circuits in an older version of Lava. For more details about how the verification is actually done, see [2].

In this section, we first introduce two new connection patterns, and then show that butterfly circuits can be made with just these two patterns and serial composition.

The first of these patterns we call `two`. The circuit `two circ` contains two copies of `circ`. The first of these operates on the first half of the input list, and the second on the second half. Each copy of `circ` should have a list as output, and the two resulting lists are appended. This pattern is easily defined in terms of `parl`, which was introduced earlier in this chapter.

```
two circ = parl circ circ

Main> simulate (two reverse) [1..16]
[8,7,6,5,4,3,2,1,16,15,14,13,12,11,10,9]

Main> simulate (two (two reverse)) [1..16]
[4,3,2,1,8,7,6,5,12,11,10,9,16,15,14,13]
```

Related to `two`, we also introduce the pattern `ilv`, for *interleave*. Whereas `two f` applies `f` to the top and bottom halves of a list, `ilv f` applies `f` to the odd and even elements. We define it in terms of the wiring pattern *riffle*, which performs

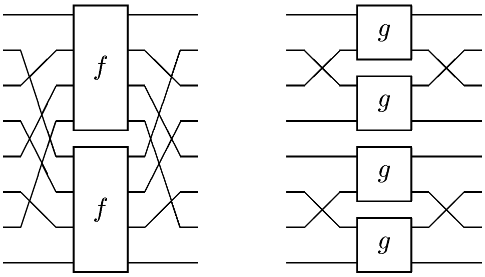

Figure 8.5: `ilv f` and `two (ilv g)`

the perfect shuffle on a list. Think of taking a pack of cards, halving it, and then interleaving the two half packs. If you now *unriffle* the pack, you reverse the process, returning the pack to its original condition. (This is somewhat more difficult to accomplish with aplomb at the poker table.)

```
Main> simulate riffle [1..16]
[1,9,2,10,3,11,4,12,5,13,6,14,7,15,8,16]

Main> simulate (riffle ->- unriffle) [1..16]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]

Main> simulate unriffle [1..16]
[1,3,5,7,9,11,13,15,2,4,6,8,10,12,14,16]
```

Note that unriffling the sequence from 1 to $n$ divides into its odd and its even elements. We use this fact to define `ilv` in terms of `two`.

```
ilv circ = unriffle ->- two circ ->- riffle

Main> simulate (ilv reverse) [1..16]
[15,16,13,14,11,12,9,10,7,8,5,6,3,4,1,2]

Main> simulate (ilv (ilv reverse)) [1..16]
[13,14,15,16,9,10,11,12,5,6,7,8,1,2,3,4]
```

Figure 8.5 shows `ilv f` and `two (ilv g)`. We leave the definition of `riffle` and `unriffle` as exercise 8.6.

We have seen from our examples that it makes sense to apply `two` and `ilv` repeatedly. We will do this so often in the butterfly circuits, that it is useful to define special functions.

```
twoN 0 circ = circ
twoN n circ = two (twoN (n-1) circ)
```

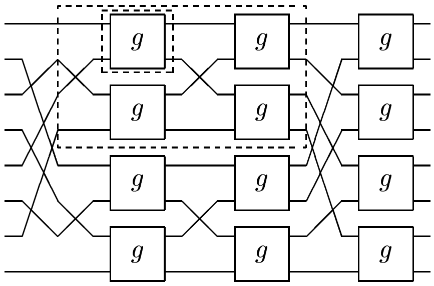

Figure 8.6: `bfly 3 g`

```
ilvN 0 circ = circ
ilvN n circ = ilv (ilvN (n-1) circ)
```

Clearly, there are similarities between these two definitions. We might just as well have defined a function that takes a *connection pattern* as input.

```
iter 0 comb circ = circ
iter n comb circ = comb (iter (n-1) comb circ)
```

Now, we can use `iter n two f` instead of `twoN n f` and `iter n ilv f` instead of `ilvN n f`.

Now we are in a position to define a connection pattern for butterfly circuits, that is circuits, like those shown in figures 8.6 and 8.7, that have a very particular recursive structure. Because the circuits are recursive, the corresponding connection pattern is defined using recursion.

```
bfly 0 circ = id
bfly n circ = ilv (bfly (n-1) circ) ->- twoN (n-1) circ
```

The smallest butterfly is just the identity. A butterfly of size $n$, for $n$ greater than zero, consists of two interleaved butterflies of size $n - 1$, the output of which is fed into a stack of `circ` components, which is made using `twoN`. This connection pattern is shown in figure 8.6, which shows `bfly 3 g`.

The larger dashed box shows one instance of `bfly 2 g`, and there is another instance just below it. These two smaller butterflies are interleaved, so there is actually an *unriffle* to their left and a *riffle* to their right. (Make sure to find these wiring patterns, and look again at the definition of `ilv`.) The two interleaved butterflies feed their outputs into four `g` components, one above the other, that is `twoN 2 g`. And if you look inside the `bfly 2 g` in the outer dashed box, you will find that it again has the same recursive structure.

Strangely enough, the same connection pattern (that is the same netlist and the same order of inputs and outputs, though a possibly different layout) can be described using a different pattern of recursion.

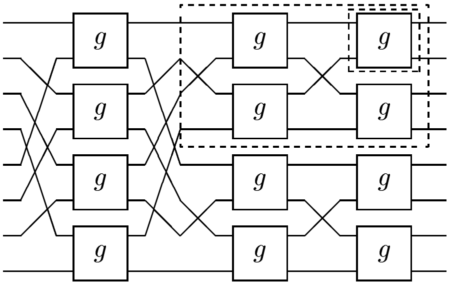

Figure 8.7: `bfly1 3 g`

```
  bfly1 0 circ = id
  bfly1 n circ = ilvN (n-1) circ ->- two (bfly1 (n-1) circ)
```

This time, we start with a repeatedly interleaved stack of basic components, whose outputs are fed into two smaller butterflies, which are combined using `two`. Figure 8.7 shows this recursive decomposition.

It turns out that `ilv (bfly n circ)` is the same as `bfly n (ilv circ)`. (See the question below about `two ilv g` if you want to figure out why.) This means that we can define the butterfly network using a single recursive call, but with a larger component:

```
  bfly2 0 circ = id
  bfly2 n circ = ilvN (n-1) circ ->- bfly2 (n-1) (two circ)

  bfly3 0 circ = id
  bfly3 n circ = bfly3 (n-1) (ilv circ) ->- twoN (n-1) circ
```

The surprising thing is that all of these connection patterns give equivalent circuits (for the same size and component).

The original butterfly defintions (`bfly` and `bfly1`) can also be expressed using a tree-like combinator. Take a look at the connection pattern `listTree`, which is a version of `binTree` which works for a component circuit `circ` processing lists.

```
  listTree circ [inp] = [inp]
  listTree circ inps  = (two (listTree circ) ->- circ) inps
```

You should think about the types involved in this definition.

Replacing that `two` by `ilv`, we get `ilvTree`, a sort of interleaved tree.

```
  ilvTree circ [inp] = [inp]
  ilvTree circ inps  = (ilv (ilvTree circ) ->- circ) inps
```

If we have a component that takes a pair as input and produces a pair as output, then we can describe a stack of such components by using pairing, unpairing and map as follows (see exercise 3.6 and the answer on page 86 for `pair` and `unpair`).

```
pmap circ = pair ->- map circ ->- unpair

Main> simulate (pmap swap) [1..16]
[2,1,4,3,6,5,8,7,10,9,12,11,14,13,16,15]
```

Then, for inputs of length $2^n$, `ilvTree (pmap circ)` is the same as `bfly n circl`, where `circl` is the same as `circ` except that it relates a 2-list to a 2-list.

So what kinds of circuits can we build with these remarkably recursive structures? Well, it turns out that `bfly 3 id` is a complicated way to write the identity function (on lists of length $8n$.) And `bfly n swapl` reverses a list of length $2^n$.

```
swapl [a,b] = [b,a]

Main>  simulate (bfly 4 swapl) [1..16]
[16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]

Main> simulate (ilvTree (pmap swap)) [1..16]
[16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
```

If we choose our basic component to be the perfect shuffle on lists of length 4, the circuit that we call `s2`, then we find that a butterfly of such components performs the perfect shuffle!

```
s2 [a,b,c,d] = [a,c,b,d]

Main> simulate (bfly 3 s2) [1..16]
[1,9,2,10,3,11,4,12,5,13,6,14,7,15,8,16]
```

But all of these examples were just wiring functions. What happens when we add some functionality to the component?

## 8.4 Batcher's Bitonic Merger

One of the best known uses of the butterfly network is in the building of mergers and sorters based on a two-input two-output comparator. Let us start with two abstract comparators that work on integer inputs. One sorts into ascending order, and the other into descending.

```
compUp   [x,y] = [imin (x,y), imax (x,y)]
compDown [x,y] = [imax (x,y), imin (x,y)]
```

```
Main> simulate (two compUp) [1,2,4,3]
[1,2,3,4]

Main> simulate (ilv compDown) [1,2,4,3]
[4,3,1,2]
```

It turns out that `bfly n compUp` sorts (into ascending order) a list whose first half is ascending and second half is descending or vice-versa. We call such lists *inc-dec* and *dec-inc* lists. (The merger sorts many other lists too, the so-called *bitonic* lists, but we don't need to worry about them.) This network is known as *Batcher's bitonic merger* [1]. Also, `bfly n compDown` sorts *inc-dec* and *dec-inc* lists into descending order.

```
Main> simulate (bfly 3 compUp) [1,3,5,7,8,6,4,2]
[1,2,3,4,5,6,7,8]

Main> simulate (bfly 3 compDown) [1,3,5,7,8,6,4,2]
[8,7,6,5,4,3,2,1]
```

Knowing that the merger sorts *inc-dec lists* allows us to build a recursive sorter. In fact, we can parameterise the circuit on the comparator (the `comp` parameter), and define both an up and a down sorter at the same time. `sorter n compUp` sorts into ascending order, while `sorter n compDown` sorts into descending order.

```
sorter 0 comp [inp] = [inp]
sorter n comp inps  = outs
  where
    sortL  = sorter (n-1) comp
    sortR  = sorter (n-1) (comp ->- swapl) -- reversed comparator
    merger = bfly n comp                   -- bitonic merger
    outs   = (parl sortL sortR ->- merger) inps

Main> simulate (sorter 3 compUp) [8,7,1,2,3,4,6,5]
[1,2,3,4,5,6,7,8]

Main> simulate (sorter 3 compDown) [8,7,1,2,3,4,6,5]
[8,7,6,5,4,3,2,1]
```

Note that our sorter is parameterised on the comparator or two-sorter component. So we have really designed the connection pattern that must be used to connect comparators. We have not in any way tied ourselves down to comparators of a particular type. So, as long as we provide a comparator component of the right type, then we get back a function of the same type that acts as a sorter.

The next step is to refine the comparator component, by choosing a concrete representation for the integer data. Examples of such representations are parallel least significant bit first binary, or serial signed twos complement. The point is that whatever refinement we choose, we can simply plug in the new component into our sorter function. This is an example of how Lava allows us to design connection patterns and then reuse them. Exercise 8.9 asks you to build a sorter based on the comparator for binary numbers that you designed in an earlier exercise.

An interesting property of sorting circuits made from comparators is that they obey the *zero-one principle.* If such a sorter works correctly on lists of integers containing only zeros and ones, then it works correctly for arbitrary integers. So, we can test an integer sorter by checking that it works on bits! In exercise 2.2, you were asked to define `twoBitSort`, which sorts a pair of bits. Here, we need the circuit `twoBitSortl` that sorts a two-list of bits:

```
twoBitSortl [a,b] = [min, max]
  where
    (min, max) = twoBitSort (a, b)
```

Now, all we need to do is to plug this component into our sorter.

```
Main> simulateSeq (sorter 2 twoBitSortl) (domainList 4)
[[low,low,low,low],[low,low,low,high]
,[low,low,low,high],[low,low,high,high]
,[low,low,low,high],[low,low,high,high]
,[low,low,high,high],[low,high,high,high]
,[low,low,low,high],[low,low,high,high]
,[low,low,high,high],[low,high,high,high]
,[low,low,high,high],[low,high,high,high]
,[low,high,high,high],[high,high,high,high]
]
```

If, after studying these examples, you find that you have developed an interest in butterfly networks, you might like to look at a paper that poses a puzzle about butterfly networks of switches [18]. Do let us know if you solve the puzzle, because we have not managed to do so!

## 8.5 Exercises

8.1 Is parallel composition (`-|-`) associative?

8.2 Are the circuits `(a->-b) -|- (c->-d)` and `(a-|-c) ->- (b-|-d)` the same or not?

8.3 Define the connection pattern `tri`.

8.4 What does a triangle of delay elements do to its inputs? When might such a circuit be useful?

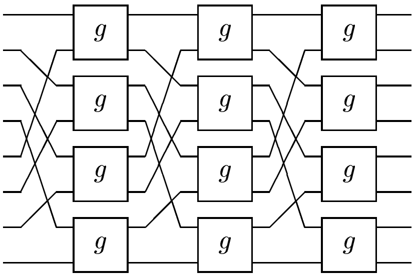

Figure 8.8: the shuffle-exchange network

8.5 The binary adder shown in this chapter works only when the binary numbers to be added are of the same length. Define a binary adder that adds two binary numbers, whatever their lengths. Use this to make a general adder tree that works for any size.

8.6 Define the wiring pattern `riffle` that corresponds to the perfect shuffle of a pack of cards.

8.7 Define the wiring pattern `unriffle` that is the inverse of `riffle`.

8.8 Verify that the sorter defined in this chapter works on list of bits, for several different sizes. How do you state the property? Hint: look at exercise 2.2.

8.9 Define a comparator that works on binary numbers and use it to make a binary number sorter.

8.10 If you have a pack of cards of size $2^n$ and riffle it repeatedly, how many riffles does it take before you are back where you started?

8.11 Consider the circuits `two (ilv f)` and `ilv (two f)`. Are they the same or not?

8.12 How would you show (using pencil and paper) that the two connection patterns `bfly` and `bfly1` are the same?

8.13 (*) Give an *iterative* rather than recursive description of the butterfly network. Hint: think of the number of `two` and `ilv` combinators in each stack of basic components. You might find a list comprehension useful.

8.14 It turns out that for two-input two-output components the butterfly network is also the same as the so-called shuffle-exchange network, which consists of a sequence of identical blocks, each of which is `riffle ->- twoN n circ`.

Figure out how many such columns you need (assuming that `circ` has two inputs and two outputs). Define the shuffle-exchange network in Lava.

Check that it is really the same as the butterfly network. In what circumstances might a circuit designer prefer the shuffle-exchange network?

8.15 We saw that `bfly n swapl` reverses its input list. Can you make `riffle` by plugging a two-input two-output wiring component into a butterfly? If not, why not?

# Chapter 9

# Synthesizing Lava Circuits

In this chapter, we present a number of examples where we generate a Lava circuit from a different kind of specification. We assume that the reader is familiar with the Haskell programming language [10].

## 9.1 State Machines

A very common way of specifying a sequential system is by constructing a *state machine*. A state machine consists of four parts: a set of states, a set of inputs, a set of initial states and a transition function. The transition function maps a state and an input to a set of next states. Usually, we draw state machines as pictures. An example of a state machine is pictured in figure 9.1.

In Haskell, here is how we might specify a datatype for representing state machines. We parametrize over the types of the states and the inputs.

```
data StateMachine state inp
  = StateMachine
      { states     :: [state]
      , inputs     :: [inp]
      , initial    :: [state]
      , transition :: state -> inp -> [state]
      }
```

Here is how we can describe the state machine in figure 9.1:

```
theStateMachine =
  StateMachine
    { states     = ["A", "B", "C"]
    , inputs     = ['a', 'b']
    , initial    = ["A"]
```

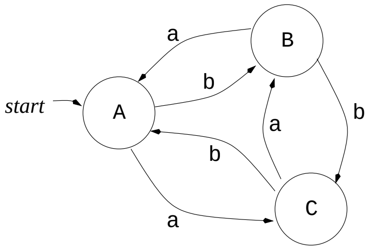

Figure 9.1: An example of a state machine.

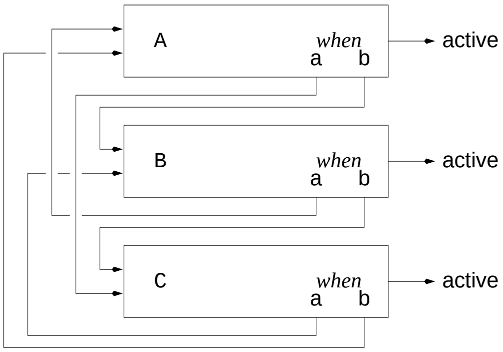

Figure 9.2: A schematic translation of the state machine of figure 9.1.

```
  , transition = \state inp ->
                   [ next | (state', inp', next) <-
                              [ ("A", 'a', "C")
                              , ("A", 'b', "A")
                              , ...
                              ]
                          , state == state'
                          , inp == inp'
                          ]
  }
```

Note that the somewhat clumsy definition of the transition function would be easier in an application where the states and inputs actually *mean* something.

Given a specification in terms of a state machine, we would like to be able to translate in into a circuit. One reason for this might be because we want a prototype implementation of the state machine. Another reason might be because we want to verify that a given circuit implementation is equivalent to the translated version.

One method of translating a state machine into a circuit is pictured in figures 9.2 and 9.3. The idea is that every state in the state machine maps to a component

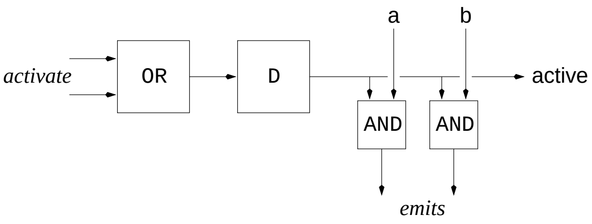

Figure 9.3: A more detailed view of the component belonging to a state.

in the circuit. The component has a delay element that keeps track of if we are in that state. The component receives messages from other component that activate it, and, depending on the inputs, also sends messages to other components activating them.

An advantage of this translation method is that we can be in several states at the same time, allowing for non-deterministic execution of our state machine. A disadvantage is that, even when our state machine is deterministic, we still have one delay component per state, which is often too much.

The type of circuits we are translating state machines to is a circuit from input signals to a list of indicators for each state.

```
type StateCircuit
  = [Signal Bool] -> [Signal Bool]
```

From these two type, we can declare the type of our translation function, which takes a state machine into a state circuit.

```
stateMachine :: (Eq inp, Eq state)
             => StateMachine state inp -> StateCircuit
stateMachine machine inSignals = outSignals
  where
    ...
```

First, we define the function `inSignal` which maps an input from the state machine to the corresponding signal wire.

```
    inSignal input =
      head [ sig
           | (input',sig) <- inputs machine `zip` inSignals
           , input == input'
           ]
```

Then, we create a list of the components, which we use as a lookup table in the rest of the translation.

```
    components =
    [ component state
    | state <- states machine
    ]
```

A component for a certain state consists of a pair `(active, emits)`, where `active` is the indicator signal for the state, and `emits` is a lookup table, representing what signal to send to what state.

```
component state = (active, emits)
  where
    init   = state `elem` initial machine
    active = delay (bool init) (activating state)

    emits  =
      [ ( state'
        , and2 (active, inSignal input)
        )
      | input  <- inputs machine
      , state' <- transition machine state input
      ]
```

The declaration of `active` uses one delay component, whose initial value depends on this state being an initial state or not, and whose next value depends on the signals the other components are sending to it (computed using the function `activating`).

The list `emits` is constructed as follows. For every input signal, we use the transition function to check what next states we have. We then send a signal to the component of state if and only if we are active, and we have that input as an incoming signal.

Here is how we define the function `activating`.

```
activating state =
  orl [ activate
      | (_, emits)         <- components
      , (state', activate) <- emits
      , state == state'
      ]
```

For all components, we look at what messages it wants to send, and filter out the signals going to the right state. Then, we take the or of all these signals.

Finally, we can create the list of state indicators, by taking the first output of the components.

```
outSignals =
  [ active
  | (active, _) <- components
  ]
```

Here is how we can make the circuit for the state machine we specified earlier.

```
theCircuit (a, b) = (inA, inB, inC)
  where
    [inA, inB, inC] =
      stateMachine theStateMachine [a, b]
```

## 9.2 Behavioral Descriptions

Another way of specifying the behavior of a circuit is by a *behavioral description language*. Examples of these kind of languages are behavioral VHDL, Verilog, Esterel, etc. The idea is to write a program in such a language, and then transform the program to a circuit with the same behavior.

We show how to compile programs in a very simple description language to a circuit. We call the language *Pace*. Here is a Haskell datatype respresenting Pace programs:

```
data Pace out
  = Skip
  | Emit out
  | Wait
  | IfThenElse (Signal Bool) (Pace out, Pace out)
  | While (Signal Bool) (Pace out)
  | Pace out :>> Pace out
  | Pace out :|| Pace out
```

A Pace program can send out messages of type `out`. Running a Pace program takes a number of clock cycles. Here is the informal semantics of Pace constructs:

- `Skip`: This program does not send any messages, and takes no time to execute.
- `Emit msg`: This program sends out the message `msg`, and takes no time to execute.
- `Wait`: This program does not send any messages, and takes 1 clock cycle to execute.
- `IfThenElse cond (p1, p2)`: If the signal `cond` is high, it executes `p1`, and sends the messages `p1` sends, and takes as long time as `p1` takes. If `cond` is low, the same, but for `p2`.
- `While cond p`: If `cond` is high, then it executes `p`, and sends the messages `p` sends, waits for the amount of time `p` takes to finish, and then tries to execute the program again. If `cond` is low, it finishes right away without sending any messages. For this program to be valid, `p` must at least take one clock cycle to execute if `cond` is high.

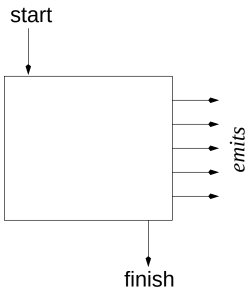

Figure 9.4: The shape of a circuit representing a Pace program.

- `p1 :>> p2`: (*sequential composition*) The program executes `p1`, waits for the time it takes to finish, and then executes `p2`.
- `p1 :|| p2`: (*parallel composition*) The program executes `p1` and `p2` in parallel, waiting for both to finish until it finishes.

Here is an example of a Pace program, where we describe a toggle:

```
togglePace change =
  While high
    ( While (inv change)
        ( Wait
        )
  :>> Emit ()
  :>> Wait
  :>> While (inv change)
        ( Emit ()
      :>> Wait
        )
  :>> Wait
    )
```

We can read the program as follows. Forever: wait until `change` is not low, then emit a message, and wait. Then, wait until `change` is not low, and emit a message all the time, then wait. The type of messages this Pace program is using, is `()`, because there is only one message.

We can give a more formal semantics to this language by giving a translation from a program to a circuit. And then we get an implementation for free!

We are going to define a function `circuit`, which takes a Pace program to a Pace circuit.

```
type PaceCircuit out
  = Signal Bool -> (PaceEmits out, Signal Bool)
```

```
type PaceEmits out
  = [(out, Signal Bool)]

circuit :: Pace out -> PaceCircuit out
```

A Pace circuit (see figure 9.4) takes in one input, called `start`, which is used to activate the program, and has two outputs, a list `emits`, and a signal `finished`, which the circuit uses to indicate that it is done. The list `emits` is a lookup table, which relates output messages and signals.

We start with `Skip`. Here, we just connect `start` to `finish`, so that we finish immediately.

```
circuit Skip start = ([], finish)
  where
    finish = start
```

In the case of `Emit`, we connect the `start` to the right output, and we finish immediately.

```
circuit (Emit out) start = (emits, finish)
  where
    emits  = [(out, start)]
    finish = start
```

When we execute a `Wait`, we connect `start` and `finish`, but with a delay, so that it takes one clock cycle to finish.

```
circuit Wait start = ([], finish)
  where
    finish = delay low start
```

To transform an `IfThenElse`, we first transform the two subprograms `prog1` and `prog2`. We start `prog1` if `start` is high and if the condition is true, and we start `prog2` if `start` is high and the condition is false. We collect all emitted messages, and finish if either one of them finishes.

```
circuit (IfThenElse cond (prog1, prog2)) start = (emits, finish)
  where
    (emits1, finish1) = circuit prog1 start1
    (emits2, finish2) = circuit prog2 start2

    start1 = and2 (start, cond)
    start2 = and2 (start, inv cond)

    emits  = emits1 ++ emits2
    finish = or2 (finish1, finish2)
```

To transform a While, we first transform the subprogram prog. Then, we introduce an auxiliary signal called active, which is high exactly when we should consider starting prog, that is when the whole while loop is started or when prog has finished. We actually start prog when we are active, and the condition is true. We finish the while loop when we are active but the condition is false.

```
circuit (While cond prog) start = (emits, finish)
  where
    (emits, finish') = circuit prog start'

    active = or2 (start, finish')
    start' = and2 (active, cond)
    finish = and2 (active, inv cond)
```

Transforming sequential composition just connects the finish of the first with the start of the second, and collects the emitted messages.

```
circuit (prog1 :>> prog2) start = (emits, finish)
  where
    (emits1, finish1) = circuit prog1 start
    (emits2, finish)  = circuit prog2 finish1

    emits = emits1 ++ emits2
```

And lastly, transforming parallel composition starts both circuits when started, collects the emitted messages, and *synchronizes* the finish signals for finishing. We use the synchronize circuit, defined in exercise 5.9 (the answer is on page 90).

```
circuit (prog1 :|| prog2) start = (emits, finish)
  where
    (emits1, finish1) = circuit prog1 start
    (emits2, finish2) = circuit prog2 start

    emits  = emits1 ++ emits2
    finish = synchronize (finish1, finish2)
```

Now we have made this translator, we can use it to turn a Pace program plus a list of output messages we are interested in into a circuit, outputting these messages.

```
compile :: Eq out => Pace out -> [out] -> [Signal Bool]
compile prog outputs = signals
  where
    start      = delay high low
    (emits, _) = circuit prog start
    signals =
      [ orl [ sig
            | (out',sig) <- emits
            , out == out'
            ]
      | out <- outputs
      ]
```

We first create a top-level `start` signal, which is to be high on the first clock tick, and then low forever, then filter out the signals we are interested in from the resulting circuit. Note that we have to take the or for these signals, since there might be several parts of the Pace program emitting the same signal.

Here is how we can create a toggle circuit from the given Pace program:

```
toggle' change = out
  where
    [out] = compile (togglePace change) [()]
```

We compile the Pace circuit, and say that we are only interested the `()` messages.

## 9.3 Exercises

9.1 In the circuit produced by the state machine translation, all inputs will only have effect on the outputs in the *next* clock cycle. Sometimes, however, it might be desirable to change state depending on the current input *right away*. In this way, you are not interested in the initial state.

Show how to change the definition of `stateMachine` to incorporate this change.

9.2 Verify that the toggle circuit derived from the Pace program is equivalent to a direct definition of a toggle circuit.

9.3 Describe the `synchronize` circuit from exercise 5.9 in terms of a state machine, and generate a circuit for it. Verify that the implementation in your answer to 5.9 is correct!

9.4 Describe the `synchronize` circuit from exercise 5.9 in terms of a Pace program, and generate a circuit for it. Verify that the implementation in your answer to 5.9 is correct!

# Chapter 10

# Types

In this chapter, we will describe what role types play in the Lava system.

## 10.1 Signals and Circuits

The circuits in Lava are *functions* from input signals to output signals. The basic signals in Lava are `low`, `high`, and integer signals. The type of the first two signals is `Signal Bool`, and that of integer signals is `Signal Int`. The notation for this is:

```
low  :: Signal Bool
high :: Signal Bool
3    :: Signal Int
42   :: Signal Int
-17  :: Signal Int
```

The types of circuits are written using the symbol `->`. Examples are:

```
and2    :: (Signal Bool, Signal Bool)   -> Signal Bool
times   :: (Signal Int, Signal Int)     -> Signal Int
halfAdd :: (Signal Bool, Signal Bool)   -> (Signal Bool, Signal Bool)
adder2  :: [(Signal Bool, Signal Bool)] -> [Signal Bool]
```

As we can see, the types for pairs are written using `(`, `,` and `)`, and the types for lists are written using `[` and `]`.

Types do not have to be explicitly written in Lava; they are automatically *derived* and *checked*. So, if we make a type error, for example by giving a list of signals rather than a pair of signals to an and gate as input, we get:

```
Main> and2 [high, low]
```

```
ERROR: Type error in application
*** Expression     : and2 [low,high]
*** Term           : [low,high]
*** Type           : [Signal Bool]
*** Does not match : (Signal Bool,Signal Bool)
```

## 10.2 Connection Patterns

To be able to deal with types in the presence of connection patterns, we need two features: *polymorphism* and *higher-order functions*.

- *Polymorphism* means that some circuits or connection patterns do not care about what kind of type we are using, as long as it matches with another (unknown) type.
- *Higher-order functions* allow us to have functions as parameters to other functions.

Here is an example: the type of the `row` connection pattern.

```
row :: ((c,a) -> (b,c)) -> (c,[a]) -> ([b],c)
```

From this we can see that `row` expects a circuit of the following type as a parameter:

```
(c,a) -> (b,c)
```

The connection pattern does not care however what exactly `a`, `b` or `c` is, as long as the two uses of `c` are the *same*. This has to be the case since `c` is the type of the carry, and the carries are matched up in the row. But apart from that, `a`, `b` and `c` can be *any* type, a signal, a pair of signals, a list of pairs of signals, etc.

## 10.3 Overloading

We have seen a number of circuits and functions that behave differently when we use them in different contexts. This is called *overloading*. We use overloading because it is convenient, we do not have to have different versions of operations around, and we can write general operations and circuits using overloaded operations.

An example is the constant `zero`, which is a generalized version of `low`. It behaves as follows.

```
Main> zero
ERROR: Unresolved overloading

Main> zero :: Signal Bool
low

Main> zero :: (Signal Bool, Signal Bool)
(low, low)

Main> zero :: Signal Int
0

Main> zero :: (Signal Bool, Signal Int)
(low, 0)
```

In the first example, we see that Lava complains because it has no idea in what kind of context you want to use `zero`. In a Lava program, this can usually be figured out, but we can be explicit about the shape of the result by using the `::` notation.

A similar constant we have seen is `domain`. It creates a list of all the possible values of a certain type. Here is how it behaves.

```
Main> domain :: [Signal Bool]
[low, high]

Main> domain :: [(Signal Bool,Signal Bool)]
[(low, low), (low, high), (high, low), (high, high)]
```

And so forth. Other examples of overloaded operators are `var` and `random`.

All these overloaded operations have a special version that works for lists. The reason for this is that, in the case of lists, we want to know how *long* they should be. How else can we create a list with only low bits in it, or sum up all the possible lists in a certain domain?

Here are some examples of how the special list versions behave in different contexts.

```
Main> zeroList 3 :: [Signal Bool]
[low, low, low]

Main> zeroList 2 :: [(Signal Bool,Signal Bool)]
[(low, low), (low, low)]

Main> domainList 2 :: [[Signal Bool]]
[[low, low], [low, high], [high, low], [high, high]]
```

```
Main> varList 3 "apa" :: [Signal Bool]
[apa_1, apa_2, apa_3]
```

Examples of a circuits that are overloaded are `delay`, `mux`, and `equal`.

# Appendix A

# Quick Reference Guide

In this appendix we present an overview of options, operations, predefined circuits and connection patterns in Lava.

## A.1 The `lava` command

Here are the command-line options for the `lava` command.

```
-hsize         set memory size to size for interpreter
-c module      compile
-ghc module    compile using GHC (default)
-hbc module    compile using HBC
-u             update internal modules after change
-x executable  use <executable> instead of compiler
```

## A.2 Logical Gates

Here are the logical gates defined in the Lava system. Some binary gates have a corresponding binary operator (for example, `and2` can also be written as `<&>`).

```
-- Nullary gates :: Signal Bool
low         -- constant low
high        -- constant high

-- Unary gates :: Signal Bool -> Signal Bool
id          -- identity
inv         -- inverse, negation
```

```
-- Binary gates :: (Signal Bool, Signal Bool) -> Signal Bool
and2,  <&>  -- logical and
nand2       -- inverse of logical and
or2,   <|>  -- logical or
nor2        -- inverse of logical or
xor2,  <#>  -- logical exclusive or
xnor2, <=>  -- inverse of exclusive or
equiv, <=>  -- logical equivalence
impl,  ==>  -- logical implication

-- n-ary gates :: [Signal Bool] -> Signal Bool
andl        -- logical and
nandl       -- inverse of logical and
orl         -- logical or
norl        -- inverse of logical or
xorl        -- logical exclusive or
```

## A.3 Arithmetical Gates

Here are the arithmetical gates defined in the Lava system. Some binary gates have a corresponding binary operator (for example, `plus` can also be written as `+`).

```
-- Nullary gates :: Signal Int
n           -- constant integer signal

-- Unary gates :: Signal Int -> Signal Int
id          -- identity
neg,   -    -- negation

-- Unary conversion
int2bit     -- integer signal to boolean signal
bit2int     -- boolean signal to integer signal

-- Binary gates :: (Signal Int, Signal Int) -> Signal Int
plus,  +    -- addition
times, *    -- multiplication
sub,   -    -- subtraction
idiv,  /    -- integer division
imod,  %%   -- modulo
imin        -- minimum
imax        -- maximum

-- Binary gates :: (Signal Int, Signal Int) -> Signal Bool
```

```
gte,   >>== -- greater than or equal

-- n-ary gates :: [Signal Int] -> Signal Int
plusl        -- addition
timesl       -- multiplication
```

## A.4 Generic Gates

Here are some generic gates defined in the Lava system.

```
equal, <==> -- equality
delay, |->  -- delay component
mux         -- multplexer, if-else-then
```

Furthermore, Lava defines some operations which can be used on some of these types:

```
domain     :: [a]
domainList :: Int -> [[a]]

zero       :: a
zeroList   :: Int -> [a]

var        :: String -> a
varList    :: Int -> String -> [a]
```

## A.5 Module: Patterns

You get access to the following wiring circuits and connection patterns if you include

```
import Patterns
```

at the top of your Lava program.

```
swap      :: (a, b) -> (b, a)
swapl     :: [a] -> [a]
copy      :: a -> (a, a)

riffle    :: [a] -> [a]
unriffle  :: [a] -> [a]

zipp      :: ([a],[b]) -> [(a,b)]
unzipp    :: [(a,b)] -> ([a],[b])
```

```
pair      :: [a] -> [(a,a)]
unpair    :: [(a,a)] -> [a]

halveList :: [a] -> ([a],[a])
append    :: ([a],[a]) -> [a]

serial    :: (a -> b) -> (b -> c) -> (a -> c)
(->-)     :: (a -> b) -> (b -> c) -> (a -> c)
compose   :: [a -> a] -> (a -> a)
composeN  :: Int -> (a -> a) -> (a -> a)

par       :: (a -> b) -> (c -> d) -> ((a,c) -> (b,d))
(-|-)     :: (a -> b) -> (c -> d) -> ((a,c) -> (b,d))
parl      :: ([a] -> [b]) -> ([a] -> [b]) -> ([a] -> [b])

two       :: ([a] -> [b]) -> ([a] -> [b])
ilv       :: ([a] -> [b]) -> ([a] -> [b])
twoN      :: Int -> ([a] -> [b]) -> ([a] -> [b])
ilvN      :: Int -> ([a] -> [b]) -> ([a] -> [b])
iter      :: Int -> (b -> b) -> (b -> b)

bfly      :: Int -> ([b] -> [b]) -> [b] -> [b]
tri       :: (a -> a) -> ([a] -> [a])

pmap      :: ((a,a) -> (b,b)) -> [a] -> [b]

mirror    :: ((a,b) -> (c,d)) -> ((b,a)     -> (d,c))
row       :: ((c,a) -> (b,c)) -> ((c,[a])   -> ([b],c))
column    :: ((a,c) -> (c,b)) -> (([a],c)   -> (c,[b]))
grid      :: ((a,b) -> (b,a)) -> (([a],[b]) -> ([b],[a]))
```

## A.6 Module: Arithmetic

You get access to the following arithmetical circuits if you include

```
import Arithmetic
```

at the top of your Lava program.

```
halfAdd  :: (Signal Bool,Signal Bool)
                   -> (Signal Bool,Signal Bool)
fullAdd  :: (Signal Bool,(Signal Bool,Signal Bool))
                   -> (Signal Bool,Signal Bool)
```

```
bitAdder :: (Signal Bool,[Signal Bool])
                   -> ([Signal Bool],Signal Bool)
adder    :: (Signal Bool,([Signal Bool],[Signal Bool]))
                   -> ([Signal Bool],Signal Bool)
binAdder :: ([Signal Bool],[Signal Bool])
                   -> [Signal Bool]

bitMulti :: (Signal Bool,[Signal Bool])
                   -> [Signal Bool]
multi    :: ([Signal Bool],[Signal Bool])
                   -> [Signal Bool]

numBreak :: Signal Int -> (Signal Bool,Signal Int)
int2bin  :: Int -> Signal Int -> [Signal Bool]
bin2int  :: [Signal Bool] -> Signal Int
```

## A.7 Module: SequentialCircuits

You get access to the following often used sequential circuits if you include

```
import SequentialCircuits
```

at the top of your Lava program.

```
edge         :: Signal Bool -> Signal Bool
toggle       :: Signal Bool -> Signal Bool
delayClk     :: a -> (Signal Bool,a) -> a
delayN       :: Int -> a -> a -> a
always       :: Signal Bool -> Signal Bool
puls         :: Int -> () -> Signal Bool
outputList   :: [a] -> () -> a

rowSeq       :: ((a,b) -> (c,a)) -> (b -> c)
rowSeqReset  :: ((a,b) -> (c,a)) -> ((Signal Bool,b) -> c)
rowSeqPeriod :: Int -> ((a,b) -> (c,a)) -> (b -> c)
```

Note that these functions are not completely polymorphic in `a`, but there are certain restrictions.

## A.8 Interpretations

Here are the various interpretations for circuits that Lava provides.

```
-- simulations
```

```
simulate circuit input
simulateSeq circuit inputs
simulateCon circuit inputs
test circuit

-- VHDL
writeVhdl name circuit
writeVhdlInput name circuit input
writeVhdlInputOutput name circuit input output

-- verification
verify property
verifyWith options property
fixit property
```

Possible verification opions are:

```
Name name
ShowTime
Sat level
NoBacktracking
Depth depth
Increasing
RestrictStates
```

## A.9 Errors

Here, we list a number of error messages that might occur when running the Lava system.

-
  ```
  ERROR: Garbage collection fails to reclaim
         sufficient space
  ```

  This means that Lava does not have enough memory to execute the circuit. Try to start up Lava with more memory, do this by saying `lava -h9999999`. You can increase the number if you need more.

  If this does not work, you might have an error in your circuit definition. Do you have a circular definition somewhere?

- `Program error: evaluating a delay component`

  You get this error when you try to use combinational simulation `simulate` to simulate a sequential circuit. Use `simulateSeq` instead.

- `Program error: evaluating a symbolic value`

  You get this error when you have used the `forAll` or `var` property constructors, and then later tried to simulate the circuit.

- `Program error: combinational loop`

  You get this error when you have defined a circuit which has a loop in it, on which there is no delay. In general, these circuits are hard to give meaning to, and are therefore not allowed in normal Lava simulation. You have probably made a mistake somewhere.

  You might try the *constructive* simulation `simulateCon` when this happens.

- `Program error: combining incompatible structures`

  You get this error when you use a `delay` component or `mux` component on structures of a different shape, for example two lists of different lengths. This is not allowed, since the length of a list needs to be known when you evaluate the circuit.

- `Program error: there is no equality defined for this type`

  Sigh ... you get this error when you use the Haskell equality == on a signal type. You probably want to use signal equality `<==>` instead.

- `Program error: short circuit`

  This happens when you have a *bad* combinational loop in your circuit, and you constructively simulate it using `simulateCon`. A real circuit would have oscillated. An example is the following circuit:

  ```
  shortCircuit () = out
    where
      out = inv out
  ```

- `Program error: undriven output`

  This also happens when you have a *bad* combinational loop in your circuit. The output wire is not driven by any component. An example is the following circuit:

  ```
  undrivenOutput () = out
    where
      out = and2 (out, out)
  ```

- `Program error: you can not enumerate symbolic values`

  You get this error when you use `..` on wires from a circuit instead of on constants. Use `..` only on constants!

- `Program error: INTERNAL ERROR ...`

  Oops! This probably means that there is a bug in the Lava system. Please report this bug by sending your program to us, so that we can fix it.

If you have some typical error that you would have liked to appear here, please e-mail us so that we can make this list more complete.

# 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 = and2 (a, b)
    max = or2  (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   = or2  (left, right)
    left  = and2 (inv c, x)
    right = and2 (c, y)
```

There is a built-in multiplexer in Lava, called `mux`. Using that one, we could define:

```
multiplexer' (c,(x,y)) = mux (c,(x, y))
```

2.5 A `threeBitAdder` can be defined as follows:

```
threeBitAdder (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))
```

3.2 We can make use of the adder we already have:

```
adder2 (as, bs) = cs
  where
    (cs, carryOut) = adder (low, (as, bs))
```

3.3 The `adder` circuit takes as an input a pair of lists of bits, whereas the `adder'` circuit gets a list of pairs of bits.

3.4 Here is a binary number to integer converter `bin2int`:

```
bin2int []     = 0
bin2int (b:bs) = num
  where
    num' = bin2int bs
    num  = bit2int b + 2 * num'
```

3.5 Here is how we can define `zipp`:

```
zipp ([],   [])   = []
zipp (a:as, b:bs) = (a,b) : rest
  where
    rest = zipp (as, bs)
```

And here is how we define `unzipp`:

```
unzipp []           = ([],   [])
unzipp ((a,b):abs) = (a:as, b:bs)
  where
    (as, bs) = unzipp abs
```

3.6 Here is how we can define `pair`:

```
pair (x:y:xs) = (x,y) : pair xs
pair xs       = []
```

We choose to ignore the last input if the number of elements is odd. And here is how we define `unpair`:

```
unpair ((x,y):xys) = x : y : unpair xys
unpair []          = []
```

3.7 This is how we can define parallel composition of circuits `par`:

```
par circ1 circ2 (a, b) = (c, d)
  where
    c = circ1 a
    d = circ2 b
```

3.9 Here is how we can define `column`:

```
column circ ([], carryIn)   = (carryIn, [])

column circ (a:as, carryIn) = (carryOut, b:bs)
  where
    (carry, b)     = circ (a, carryIn)
    (carryOut, bs) = column circ (as, carry)
```

Here is how we can define `column` in terms of `row`. First, we define a connection pattern called `mirror`, which swaps the left and right parts of input and output:

```
mirror circ (a, b) = (c, d)
  where
    (d, c) = circ (b, a)
```

And then, we use `row` and mirror the input to `row`:

```
column circ (as, carryIn) = (carryOut, bs)
  where
    (bs, carryOut) = row (mirror circ) (carryIn, as)
```

We could even say:

```
column circ = mirror (row (mirror circ))
```

3.10 We could define `grid` as:

```
grid circ (as, bs) = (cs, ds)
  where
    (cs, ds) = row (column circ) (as, bs)
```

Or, even shorter:

```
grid circ = row (column circ)
```

3.13 Here is how we define a `swapper`:

```
swapper (swap, (a, b)) = (x, y)
  where
    (x, y) = mux (swap, ((a, b), (b, a)))
```

4.1 The first property can be defined as:

```
prop_SorterHasSortedOutput (a, b) = ok
  where
    (x, y) = twoBitSort (a, b)
    ok     = or2 (inv x, y)  -- x <= y
```

The second property can be stated as:

```
prop_SorterHasSameBits (a, b) = ok
  where
    (x, y)  = twoBitSort (a, b)
    same    = (a, b) <==> (x, y)
    swapped = (a, b) <==> (y, x)
    ok      = or2 (same, swapped)
```

4.4 To check that the subtractor really subtracts, we can define:

```
prop_SubtractorSubtracts (as, bs) = ok
  where
    cs  = subtractor (as, bs)
    as' = adder2 (cs, bs)
    ok  = as <==> as'
```

4.5 Here is the general property of associativity:

```
prop_Associative circ (as, bs, cs) = ok
  where
    out1 = circ (as, circ (bs, cs))
    out2 = circ (circ (as, bs), cs)
    ok   = out1 <==> out2
```

4.8 One can define a general verify function, as follows:

```
verifyFor prop ns = sequence [ prop n | n <- ns ]
```

and verify a property by saying for example:

```
Main> verifyFor prop_AdderCommutative_ForSize [1..32]
...
```

5.1 We can define evenSoFar as follows:

```
evenSoFar inp = out
  where
    out  = delay high even
    even = xor2 (inp, even)
```

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

5.2 We can define flipFlop as follows:

```
flipFlop (set, reset) = state
  where
    state' = delay low state
    state  = and2 (up, inv reset)
    up     = or2 (state', set)
```

5.3 We can define delayClk as follows:

```
delayClk init (clk, inp) = out
  where
    out = delay init val
    val = mux (clk, (out, inp))
```

5.4 The circuit always can be defined as follows:

```
always inp = ok
  where
    sofar = delay high ok
    ok    = and2 (inp, sofar)
```

5.5 The circuits are:

```
pulsSix6 () = out
  where
    out = puls 6 ()                     -- 000001...

pulsSix5 () = out
  where
    a   = puls 2 ()                     -- 010101...
    b   = puls 3 ()                     -- 001001...
    out = and2 (a, b)

pulsSix3 () = out
  where
    a   = delay low (inv a)             -- 010101...
    b   = delay low (xor2 (b, c))       -- 001001...
    c   = delay low (nand2 (b, c))      -- 011011...
    out = and2 (a, b)
```

5.6 Using a counter, we can define puls2 as follows:

```
puls2 k () = out
  where
    number = counter k ()
    out    = norl number
```

5.7 We can define the circuit counterUpDown as follows:

```
counterUpDown n (up, down) = number
  where
   number' = delay (zeroList n) number
   number  = adder2 (diff, number')
   diff    = one : replicate (n-1) rest
   one     = or2 (up, down)        -- should I change?
   rest    = and2 (inv up, down)  -- +1 or -1?
```

5.9 Here is how we could define synchronize:

```
synchronize (go1, go2) = go
  where
    both = and2 (go1, go2)
    one  = xor2 (go1, go2)
    wait = delay low (xor2 (one, wait))
    go   = or2 (both, and2 (wait, one))
```

5.10 First, we define the following helper circuit outputDone. It does the same as output, but takes an extra parameter done, the signal to output at the time when the list is empty.

```
outputDone [] done () = done

outputDone (sig:sigs) done () = out
  where
    out  = delay sig rest
    rest = outputDone sigs done ()
```

Now, we can define the circuit outputList as follows:

```
outputList sigs () = out
  where
    out = outputDone sigs out ()
```

6.2 Here is a property that checks that:

```
prop_Edge_vs_Even inp = ok
  where
    out1 = edge inp
    out2 = evenSoFar inp
    ok   = inv (out1 <==> out2)
```

6.3 Here is a property that checks if they are equivalent:

```
prop_PulsSixEquivalent () = ok
  where
    out3 = pulsSix3 ()
    out5 = pulsSix5 ()
    out6 = pulsSix6 ()

    ok35 = out3 <==> out5
    ok56 = out5 <==> out6
    ok   = and2 (ok35, ok56)
```

These can be verified with induction depth 6 or 7.

6.4 Here is a property that checks if they are equivalent:

```
prop_PulsesEquivalent k () = ok
  where
    out1 = puls (2^k) ()
    out2 = puls2 k ()
    ok   = out1 <==> out2
```

6.5 Here is a property that checks that:

```
prop_CountingUp n up = ok
  where
    out1 = counterUp n up
    out2 = counterUpDown n (up, low)
    ok   = out1 <==> out2
```

6.6 Here is how we could define the property.

```
prop_ToggleTwiceStaysSame inp = ok
  where
    out     = toggle inp
    out'    = delay low out
    out''   = delay low out'
    sameOut = out <==> out''

    inp'    = delay low inp
    sameInp = inp <==> inp'

    ok      = sameInp ==> sameOut
```

First we compute the output from the input. Then, we define the outputs and inputs at several different points in time. And then we compute the implication.

7.1 Here are the properties which state this:

```
prop_ToggleHighLow_SlowedDown () = ok
  where
    load  = puls 2 ()
    out1  = highLow ()
    out1' = parallelToSerial (load, out1)
    out2  = toggle high
    ok    = out1' <==> out2

prop_ToggleHighLow_SpedUp () = ok
  where
    out1  = highLow ()
    out2  = timeTransform toggle [high,high]
    ok    = out1 <==> out2
```

Note that we do not need to use a serial to parallel converter in the first property since `highLow` does not have any interesting input.

7.2 Slowing down a circuit means that there are only a few important clock cycles, and we ignore all unimportant clock cycles. If we do not look at some outputs, we cannot say anything about how the circuit behaves in these outputs. The slowed down property might be true, but the circuits are not equivalent.

7.3 Yes, here there is no problem.

8.1 No, parallel composition is not associative. `(a,(b,c))` and `((a,b),c)` are not the same.

8.2 Yes, they are the same.

8.3 One possibility to define `tri` is

```
tri circ []          = []
tri circ (inp:inps) = inp : outs
  where
    outs = (map circ ->- tri circ) inps
```

There are many other ways of defining it. For example, you should try defining `tri` using `composeN`.

8.6 We give a definition that closely reflects our informal explanation of how a card sharp shuffles the pack. He halves it, zips the two halves together (to get a lot of pairs of cards) which he then pats carefully on the sides so as to unstick the pairs.

```
riffle = halveList ->- zipp ->- unpair
```

We use the circuit `zipp`, which we defined in exercise 3.5.

8.7 This definition is exactly the inverse of the definition of `riffle`:

```
unriffle = pair ->- unzipp ->- append
```

8.8 We need to verify two properties:

- The output of the sorter is sorted. We can verify this by checking that the first output is smaller than the second, the second output is smaller than the third, etc.
- The bits in the output are the same as the bits in the input, but maybe in a different order. We can verify this by counting the number of high inputs and high outputs, and checking that they are the same.

The details are left to the reader.

8.10 On inputs of length $2^n$, $n$ riffles in a row gets you back to where you started.

```
Main> simulate (composeN 4 riffle) (map int [1..(2^4)])
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

8.13 Here is how we define the butterfly circuit iteratively:

```
ibfly 0 circ = id
ibfly n circ =
  compose [ilvN (n-1-j) (twoN j circ) | j <- [0..(n-1)]]
```

9.1 We could make the following change to the local definition of `component`:

```
component state = (activated, emits)
  where
    init      = state `elem` initial machine
    activated = activating state
    active    = delay (bool init) activated
    ...
```

The rest of the definition stays the same.

# Bibliography

[1] K.E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32, 1969.

[2] Per Bjesse. Automatic verification of combinational and pipelined FFT circuits. In *Computer Aided Verification*. Springer Verlag, July 1999.

[3] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*. ACM, Sept. 1998.

[4] Cadence. The Cadence SMV Model Checker. Available from http://www.kenmcmil.com/smv.html.

[5] J.W. Cooley and J.W. Tukey. An algorithm for the machine computation of complex fourier series. In *Mathematics of Computation, 19*, pages 297–301, 1965.

[6] Alexandre Frey, Gérard Berry, Patrice Bertin, François Bourdoncle, and Jean Vuillemin. Jazz. Available from `http://www.cma.ensmp.fr/jazz`, 1998.

[7] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proc. IEEE*, 79(9), 1991.

[8] Steven Johnson. *Synthesis of Digital Designs from Recursion Equations.* The ACM Distinguished Dissertation Series, The MIT Press, 1984.

[9] M. P. Jones. The Hugs distribution. Currently available from `http://haskell.org/hugs`, 1999.

[10] Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[11] Wayne Luk, Geraint Jones, and Mary Sheeran. Computer-based tools for regular array design. In J McCanny, J McWhirter, and E Swartzlander, editors, *Systolic Array Processors*, pages 589 – 598. Prentice-Hall International, 1989.

[12] John Matthews and John Launchbury. Elementary microarchitecture algebra. In *Int. Conf. on Computer Aided Verification.* Springer Verlag, LNCS, 1999.

[13] John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Functional Programming Languages in Education.* Springer Verlag, LNCS, 1996.

[14] Mary Sheeran. $\mu$FP, an algebraic VLSI design language. PhD thesis, Programming Research Group, Oxford University, 1983.

[15] Mary Sheeran. $\mu$FP, a language for VLSI design. In *ACM Symp. on LISP and Functional Programming*, 1984.

[16] Mary Sheeran. Designing regular array architectures using higher order functions. In *Int. Conf. on Functional Programming Languages and Computer Architecture, LNCS 201.* Springer Verlag, 1985.

[17] Mary Sheeran. Describing and reasoning about circuits using relations. In *Workshop on Foundations of VLSI Design.* Kluwer, 1990.

[18] Mary Sheeran. Puzzling permutations. In *Proc. Glasgow Functional Programming Workshop*, 1996.

# Index