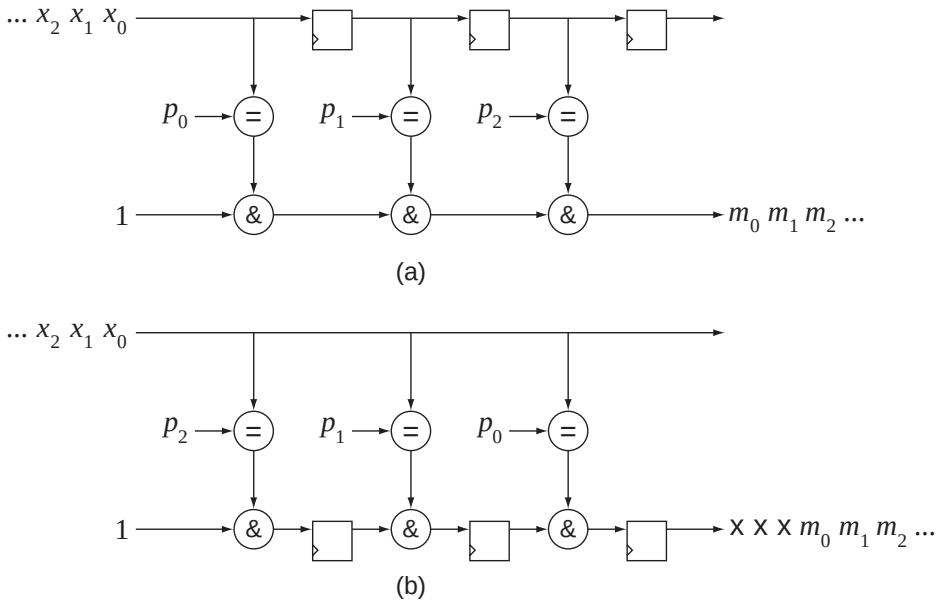
4 Regular 2-Sided Composition

This tutorial shows how regular circuits, i.e. circuits with repeated structures, can be described using Lava. The type of regular circuits that are discussed in this tutorial are those that are made up from repeating structures that can be considered to have two sides, that contain the inputs and the outputs. The repeating structures in these type of circuits are composed in series and parallel, as shown in the previous two tutorials. The next tutorial shows how to describe circuits composed from 4-sided elements.

The steps in describing regular pattern matcher designs are given here to demonstrate the features of Lava for describing 2-sided regular designs. Figure 4.1 shows the system flow diagrams of two regular pattern matcher designs that produce the same results from the same input data and pattern, but with different timing.

Figure 4.1

Pattern matcher designs that match a 3 value pattern *p* with a serial data stream *x*. The = operator compares two values and produces a boolean TRUE if they are equal. The & operator is the boolean AND operator.



Design (a) is the straightforward implementation of the boolean equation

$$m_t = 1 \quad (x_t = p_0) \quad (x_{t-1} = p_1) \quad (x_{t-2} = p_2) \ldots \quad (x_{t-n} = p_n)$$

where *t* is time and *n* is the size of the pattern. The = operators within the paratheses represent boolean equality, the dots between them represent boolean AND and the 1 is boolean TRUE. The first value of the pattern *p* is compared with the value of the input stream *x*, and the other values of the pattern are compared with the previous values of input stream. If the comparisons are all TRUE, then a match has been found in the input stream, and the output *m* is asserted as TRUE.

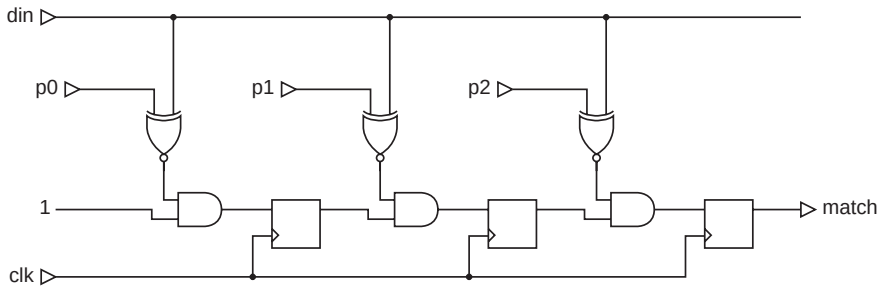
Design (b) also matches the pattern with the input stream but the output is produced 3 cycles later. It is a less obvious implementation than design (a). The advantage of this design is that the registers break the combinational path through the AND operators. Assuming that the delay on the combinational path is larger than the wires, this design has a lower overall combinational delay. For this reason, this design is used throughout the rest of the tutorial. Two different Lava implementations will be described, the first matches a 1-bit pattern with a 1-bit data stream and the second matches an *n*-bit pattern with an *n*-bit data stream.

4.1 A 1-Bit Pattern Matcher Design

The 1-bit pattern matcher design is shown in Figure 4.2. The equals operator is implemented with an XNOR gate and the AND operator with an AND gate. The first thing to do is to describe the repeating structure. This is the XNOR gate in series with the AND gate, which are both in series with the register. The schematic shows this structure repeated 3 times, but we are going to write a Lava program that can produce an implementation with a repeated structure of any size.

Figure 4.2

1-bit pattern matcher design that matches a serial data stream with a pattern of length 3. The equals operator is implemented as an XNOR gate and the logical AND as a single AND gate.



It is possible to implement both the XNOR and AND gates of the repeating structure in a single LUT of the Virtex architecture. The simplest way to describe this combinational logic is therefore to initialise a LUT with the logic function.

The initialising function is:

```
xnorAndFunc :: Bool -> Bool -> Bool -> Bool
xnorAndFunc i j k = (i == j) && k
```

The XNOR function is achieved by the boolean equals operator, written as ==. This initialises a 3-input LUT, which is composed in series with the register. Since they can share the same slice, we use the >|> combinator:

```
lut3 xnorAndFunc >|> fd clk
```

This structure is repeated by a combining it *n* times in series. The only complications are the pattern input *p* and the `clk` signal, these must be excluded from the series composition. As mentioned before, arguments can be excluded from a composition by partial application. The wires to be included in the series composition are the top wire that passes through, the lower input to the AND gate and the output from the register. These two inputs and outputs need to be combined as a pair for the series composition. To get this wiring we can use `fsT` and a bit of custom wiring. The function for the repeating structure can be written in Lava as follows:

```
patternMatcherCell :: Bit -> Bit -> (Bit, Bit) -> (Bit, Bit)
patternMatcherCell clk p
  = pre p >=> fsT (lut3 xnorAndFunc >|> fd clk)
    where pre p (a,b) = ((p,b,a),b)
```

All that remains now is to compose *n* of these is series. Lava provides combinators that compose a list of circuits in series. In this case we will use the horizontal version `hser`. To describe the circuit in Figure 4.2 we can write the following in Lava:

```
hser [patternMatcherCell clk p0,
      patternMatcherCell clk p1,
      patternMatcherCell clk p2]
```

To describe a circuit of size *n*, we need to be able to create a list of `patternMatcherCell` circuits of size *n*. The question is, how do we create a list that provides the correct pattern inputs *p*? Haskell actually provides an expression called *list comprehension* that provides a convenient way of creating lists from other lists. Using list comprehension, it is possible to create the correct sized circuit from a list of the pattern inputs. The Lava for this is as follows:

```
hser [patternMatcherCell clk p | p <- ps]
```

The expression `[patternMatcherCell clk p | p <- ps]` creates a list of `patternMatcherCell clk p` taking each element `p` of the list `ps` in turn. The complete function for the pattern matcher is as follows:

```
patternMatcher :: Bit -> [Bit] -> Bit -> Bit
patternMatcher clk ps din
  = match
    where
    (match, dout) =
              hser [patternMatcherCell clk p | p <- ps] (vcc, din)
```

When we implement a pattern matcher circuit, we need to create a set of input and output ports including ports for the pattern inputs. Lava provides the function `inputBitvec`, which is a variation of the function `inputBit`, to create a set of input ports that connect to a list of wires. Using this and the pattern matcher circuit we can implement the circuit in Figure 4.2 by writing the following Lava:

```
patternMatcherCircuit
  = (outputBit "match" match)
    where
    din = inputBit "din"
    clk = inputClock "clk"
    (match) = patternMatcher clk p din
    p = inputBitvec "p" (0 `to` 2)

main :: IO ()
main
  = do nl <- netlist "patternMatcher" patternMatcherCircuit
       writeNetlist nl virtex [vhdl, edif]
```

The syntax of the `` `to` `` function is inspired by the syntax of VHDL vector types. It means create wires with indices that take the values between and including the left expression to the right expression. Conversely, the `` `downto` `` function creates wires with indices from the left expression down to the right expression. The correct sized pattern matcher circuit will be created according to the parameters supplied to the pattern input port.

4.2 An N-Bit Pattern Matcher Design

A 3-bit instance of the N-bit pattern matcher design is shown in Figure 4.3. This design matches 3-bit data with a 3-bit pattern of length 3. The equals operator matches 3 bits of data in parallel and is implemented also as a parallel 1-bit pattern matcher. It is therefore a regular 2-sided circuit, and so the overall circuit is a composition of two nested regular circuits.

To describe this circuit in Lava, we first describe the repeating structure for the equals operator, which is similar to the repeating structure of the 1-bit pattern matcher:

```
patternMatcherSubCellN :: Bit -> Bit -> Bit -> Bit
patternMatcherSubCellN p din
  = pre p din >=> lut3 xnorAndFunc
  where
  pre p din cin = (p, din, cin)
```

This circuit block takes three inputs and produces one output. The first input `p` is the pattern, the second `din` is the data input and the third `cin` is the output from the previous stage. The series composition involves the `cin` input and the output, the other inputs are parallel inputs. We can compose this as before using list comprehension, except now, there is an additional complication. When generating the list of circuit blocks for the composition, we need to take two inputs at a time from the two lists of inputs. This can be done by combining the two lists into one list of tuples, containing the elements

from each list, and using this list in the list comprehension. There is a function to do just this:

```
zip a b
```

Which takes:

```
a = [a0, a1, a2, ...]
b = [b0, b1, b2, ...]
```

and produces:

```
[(a0, b0), (a1, b1), (a2, b2), ...]
```

We can write the series composition as:

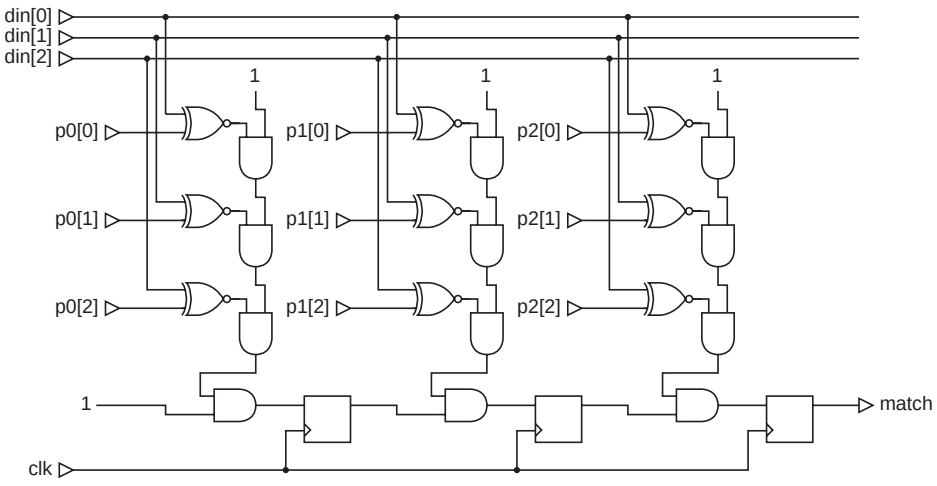
```
vser [(patternMatcherSubCellN p din) | (p, din) <- zip ps dins]
```

and the complete code for the main repeating structure:

```
patternMatcherCellN :: Bit -> [Bit] -> [Bit] -> Bit -> Bit
patternMatcherCellN clk ps dins
  = pre >=>
    fsT (vser [(patternMatcherSubCellN p din) | (p, din) <- zip ps dins])
    /\ (and2Gate >|> fd clk)
  where
  pre w = (vcc, w)
```

Note that the blocks are placed the from bottom to top using the `vser` operators and `/\`, not from top to bottom as shown in Figure 4.3.

Figure 4.3 3-bit pattern matcher design that matches a serial data stream with a pattern of length 3. The equals operator is implemented as a parallel version of 1-bit pattern matcher.



The complete circuit can now be composed in the same way, the function for this is shown below:

```
patternMatcherN :: Bit -> [[Bit]] -> [Bit] -> Bit
patternMatcherN clk ps dins
  = hser [(patternMatcherCellN clk p dins) | p <- ps] vcc
```

The only thing to note is that the pattern inputs are a represented by a list of lists. In the above function, `ps` is a list of lists, and `p` is a list.

TO DO:

1) VHDL interface -- all the code ?

2) floorplan

4.3 Exercises

1. The Lava below describes a 1-bit specialized pattern matcher design. A pattern is supplied at compile time as a boolean constant. Partially applying the pattern to the LUT initialisation function produces an initialisation expression for a 2-input LUT. This design uses less resources than the design with a pattern input since it does not require any circuitry to supply the pattern. The pattern remains fixed at run time, unless the circuit is reconfigured dynamically.

```
patternMatcherCellSpec :: Bit -> Bool -> (Bit, Bit) -> (Bit, Bit)
patternMatcherCellSpec clk p
  = pre >=> fsT (lut2 (xnorAndFunc p) >|> fd clk)
  where pre (a,b) = ((b,a),b)

patternMatcherSpec :: Bit -> [Bool] -> Bit -> Bit
patternMatcherSpec clk ps din
  = match
    where
    (match, dout) = hser [patternMatcherCellSpec clk p | p <- ps] (vcc, din)
```

Provide the Lava code for an N-bit specialized pattern matcher design.

2. Modify the N-bit pattern matcher design so that it uses the carry logic to implement the equals operator.

3. Provide a specialized version of the modified N-bit pattern matcher design.