Lab Exercises for CλaSH

Christiaan Baaij

1 Introduction

It is recommended that you use the following template for all the exercises. A copy can be found on the website: `http://goo.gl/WMqqd`

```
{-# LANGUAGE TemplateHaskell #-}
import CLasH.HardwareTypes

type Word = Unsigned D16

mealyT (State s) i = (State s',o)
  where
    s' = -- The value of the new/updated state
    o  = -- The value of the output

initState = -- initial value of the state

{-# ANN machine TopEntity #-}
machine :: Comp Word Word
machine = mealyT ^^^ initState

testInput = -- Input to test your machine with

main = print (simulate machine testInput)
```

2 Exercise 1: Shift Register

A shift-register remembers N consecutive inputs. The output of the shift-register is the value that will be shifted out on the next cycle.

a Update the definition of `mealyT` to behave like a shift-register.

b Change `initState` so that the shift register can remember 4 consecutive values.

c Define `testInput` and verify that your circuit behaves correctly by executing `main`.

3 Exercise 2: FIFO Buffer

A FIFO buffer behaves similarly to a shift-register, the difference being that the last value is not evicted every cycle. There is an extra control signal that tells if the a value should be shifted out or not. It also has a status flag indicating that the buffer is full.

a Change the type of `machine` to:

```
machine :: Comp (Word,Bool) (Word,Bool)
```

The extra input `Boolean` value indicates if the last value should be evicted. The extra output `Boolean` value indicates if the buffer if full.

b Update the definitions of `mealyT` and `initState` to behave like a FIFO buffer.

c Define `testInput` and verify that your circuit behaves correctly by executing `main`.

4 Exercise 3: Pattern Matcher

A pattern matcher remembers N consecutive inputs, tests whether they match a specific pattern, and increments a counter for every matched sequence. The output of the pattern-matcher is the number matched patterns.

a Change the definition of `mealyT` and `initState` to behave like a pattern matcher. You can use the following expression to define the pattern:

```
$(vTH [0::Word,1,0,1])
```

b Define `testInput` and verify that your circuit behaves correctly by executing `main`.

c Instead of matching against a pattern, update `mealyT` to test if a sequence of values forms a palindrome.

d Update `mealyT` so that the first N-1 values are not tested if they form a palindrome.

e Change the definition of `mealyT` to be parametrizable in its functionality: that is, it should be parametrizable to be e.g. a pattern matcher or a palindrome tester.

Create two copies of the `machine` function that each specialize the new `mealyT` definition:

- One that performs pattern matching
- One that performs palindrome testing

Create or reuse test inputs to verify that both machines operate correctly.

5 Cheat Sheet

5.1 Type-Level Natural Numbers

CλaSH has natural numbers on the type-level. They are used, amongst others, in the fixed-size Vector types of CλaSH. Here is an example of how you can use them:

```
copiesOfFalse :: Vector D8 Bool
copiesOfFalse = vcopy False
```

The above code creates a Vector of length 8 as indicated by the `D8`.

There are also terms that correspond directly to these type-level integers, as can be observed in the following example:

```
copiesOfTrue = vcopyn d8 True
```

The above code creates a Vector of length 8 (notice that we use `vcopyn` and not `vcopy`). The term `d8` (notice the lower-case 'd') is the term that corresponds to the type `D8` (notice the capital 'D'). The type of `copiesOfTrue` is hence `Vector D8 True`.

5.2 Library Functions

Some standard library functions for you to use:

```
-- | Create a Component out of a Mealy-machine-like functions
(^^^) :: (State s -> i -> (State s,o)) -> s -> Comp i o

-- | Return the first element of a list
vhead :: Vector n a -> a

-- | Return everything but the first element of a Vector
vtail :: Vector n a -> Vector (Pred n) a

-- | Return the last element of a Vector
vlast :: Vector n a -> a

-- | Return everything but the last element of a Vector
vinit :: Vector n a -> Vector (Pred n) a

-- | Add an element to the beginning of a Vector (infix)
(+>) :: a -> Vector n a -> Vector (Succ n) a

-- | Add an element to the end of a Vector (infix)
(<+) :: Vector n a -> a -> Vector (Succ n) a

-- | Add element to the start of the Vector, throw away the last one (infix)
(+>>) :: a -> Vector n a -> Vector n a

-- | Add element to the end of the Vector, throw away the first one (infix)
(<<+) :: Vector n a -> a -> Vector n a

-- | Map a function 'f' over a vector 'xs'
vmap :: (a -> b) -> Vector n a -> Vector n b

-- | Zip two vectors with a binary operation
vzipWith :: (a -> b -> c) -> Vector n a -> Vector n b -> Vector n c

-- | Reduce a vector to a single value (left associative)
vfoldl :: (a -> b -> a) -> a -> Vector n b -> a

-- | Reduce a vector to a single value to a (right associative)
vfoldlr :: (b -> a -> a) -> a -> Vector n b -> a

-- | Create a Vector with 'n' copies of an element, where 'n' is derived from the context
vcopy :: a -> Vector n a

-- | Create a Vector with 'n' copies of an element, where 'n' is the first argument
vcopyn :: n -> a -> Vector n a

-- | Get an element out of a vector at the specified index (infix)
(!) :: Vector s a -> Index s -> a

-- | Replace an element in a vector with an other element
vreplace :: Vector s a -> Index s -> a -> Vector s a
```

```
-- | Get the maximum index of a vector
maxIndex :: Vector s a -> Index s

-- | Reverse the elements of a vector
vreverse :: Vector s a -> Vector s a
```