Massachusetts Institute of Technology – Laboratory for Computer Science

IAPBlue   A Mini Course on Bluespec

`http://www.csg.lcs.mit.edu/IAPBlue`

Lab Exercises 2 14 Jan 2003

Remember that you can refer to the "How-To" handout from Monday for details on running the Bluespec compiler and simulating the output. More complete documention on the Bluespec compiler is available in the "Bluespec Compiler User Manual" on the IAPBlue website. The Bluespec language and standard libraries are documented in the "Bluespec Language Manual," also on the website.

Exercise 1 Divide by three

In Lab 1, problem 3b asked you to write a circuit which keeps accepting digits of a number and reports whether the number, as seen so far, is divisible by three. The file `Div3.bs` contains one possible solution for that problem. The `mkDiv3` module has a register `modsum` which keeps track of the remainder of the sum of the digits when divided by three (the "mod"). When a new digit comes in to the module, a `case`-statement considers the current remainder and the incoming digit to produce the new remainder, which is stored in `modsum`.

Now let's think about taking one bit at a time, rather than a whole hexidecimal digit.

Part a:

Write a function which takes a remainder value and a single bit and returns the new remainder. You might want to use a `case`-statement as in `mkDiv3` and in the shifter example from lecture.

Your function should have the following type:

```
computeRemainder :: Bit 1 -> Bit 2 -> Bit 2
computeRemainder nextbit modsum = ...
```

Part b:

Now we have a combinational circuit `computeRemainder` which can compute a new remainder given a single bit. If we are given a long bit vector, and we want to compute the remainder, we would need to chain many 1-bit circuits together, so that each is passing its computed remainder to the next one. To do this, we use the function `foldr`.

See, for example, the variable shifter from lecture. In that example, the `step` function was folded over a list to generate a chain of steps.

Use `foldr` to fold the function `computeRemainder` over an arbitrarily-sized bit vector. Your new function should have the following type:

```
computeRemainderN :: Bit n -> Bit 2 -> Bit 2
```

In order to fold over the bits in a bit vector, you will need to convert the bit vector into a list of single bits. To do this, use the `listBits` functions in the file `ListBits.bs` which we have provided with this lab.

Can you write a module around this function which takes bit vectors and reports whether they are divisble by three?

Part c:

As we saw in Lecture 3, the combinational cascade of steps in the variable shifter could be pipelined by adding a FIFO between each step.

Using a similar technique, pipeline the `computeRemainderM` circuit.

Exercise 2 Register Array

The `Array` interface that was presented in lecture has the following general definition:

```
interface Array i a =
    upd :: i -> a -> Action
    sub :: i -> a
```

This interface can be instantiated for any index `i` and value `a`.

Let's consider a specific instance of `Array` which has an index type of `(Bit 2)` and a value type of `(Bit n)`. This means that there are at most four entries in the array.

```
    upd :: Bit 2 -> Bit n -> Action
    sub :: Bit 2 -> Bit n
```

Part a:

Write a module `mkArray` which defines the `(Array (Bit 2) (Bit n))` interface using only `Reg` primitives. The module should have the following type:

```
mkArray :: Module (Array (Bit 2) (Bit n))
mkArray = module ...
```

You will need to instantiate four registers to hold the four array values. The interface to your module will need to contain `case`-statements which select the register to read or write based on the index.

The registers in your array can be uninitialized, initialized to zero, or you can add an argument to `mkArray` which is the initial value of the array:

```
mkArrInit :: (Bit n) -> Module (Array (Bit 2) (Bit n))
```

Exercise 3 FIFO

Now that we have practiced with `case` and `foldr` and arrays of registers, let us implement a FIFO in Bluespec using only register primitives.

The `FIFO` interface that we saw in lecture has the following form:

```
interface FIFO t =
    enq   :: t -> Action
    first :: t
    deq   :: Action
    clear :: Action
```

Part a:

Before implementing the FIFO, let's make a prediction about the conflicts between interface methods. Write a conflict grid (like we saw in Lecture 2) showing which interface methods you would expect to conflict, which you would expect to be conflict free, and which you would expect to be sequentially composable.

For example, should `enq` and `deq` be allowed to happen at the same time? Write down what you think a good implementation of FIFO should allow.

Part b:

Write Bluespec code for a 2-place FIFO using registers. (The code you wrote for an array of registers might be useful.) To make type-checking simpler, let's assume that the FIFO holds bit vectors. Your module should have the following signature:

```
mkFIFO :: Module (FIFO (Bit n))
```

Part c:

Generate code for `mkFIFO` with the Bluespec compiler and inspect the header of the generated Verilog or C code. There should be a line that looks like this:

```
// Method conflict free info:
```

Following that line is any composability relationships that the compiler determined for the FIFO's interface methods. If no information is listed, then the compiler believes that all interface methods conflict. For example, the following information is from `mkGCD` from Lab 1:

```
// Method conflict free info:
// [result_ < start_]
```

This says that the `result_` method is sequentially composible with `start_` only in the order `result_` followed by `start_`. The symbol `<>` is used to indicate that the methods can be composed in either direction.

From this information, draw the conflict matrix that the compiler determined for your FIFO design. How well does it correspond to what you had hoped to achieve in part (a)? It is likely much more restrictive (unless your prediction was not hopeful enough). Do you think that, given the language constructs that we have shown you so far, you could write a FIFO which has your hopeful conflict properties? (Hint: Don't spend too much time trying to writing it.)

Part d:

If you have time, try generalizing your FIFO implementation to an arbitrary depth FIFO. Passing the size as a parameter to `mkFIFO` is tricky, so for this exercise you're welcome to write a function which makes a FIFO module of fixed size, but which has one number inside that you can conveniently change in the code to change the size of the FIFO that is generated. If you still want to try tackling the problem of how to pass that size as a parameter, you're welcome to.

Part e:

If you have more time, write an array of registers or a FIFO which supports an additional `find` method. Find has the following signature:

```
find :: ((Bit n) -> Bool) -> Bit n
```

The `find` method takes a function and applies it *in order* to each element of the array or FIFO, returning the first element for which the function is true.

We will use this method in lecture tomorrow to define bypass FIFOs for a 5-stage CPU.

First start by writing `find` for a specific size array or FIFO. Then, if you're feeling very adventurous, try writing `find` for a FIFO of arbitrary depth.