Hardware Description and Verification: Lava Lab

Emil Axelsson, Thomas Hallgren and Mary Sheeran

April 25, 2012

What to submit

A submission will consist of a single Lava-runnable file containing the source code of the circuits and properties you have defined, and all the helper functions. Clearly mark in comments which definition is part of which answer. The assignments are numbered **A1** to **A9** and there is an extra assignment in case you find the lab too easy. You can pass the lab by doing a reasonable job on assignments **A1** to **A7**

A partial solution is provided in the file `LavaLabRemoved12.hs`. You are not required to use it, but we advise doing so, particularly if you have not seen functional programming before. Give the answers to the questions, motivation for decisions you have made, etc. Mention what actually happened (time, output etc.) when you did a simulation or verification. Give us as much information about what you have done as possible, so that we have an easier job of marking and giving feedback. But be brief!

Comments in Lava can be written in two ways:

```
-- comment until the end of the line

{- A lot of comments
   possibly spanning over
   more than one line -}
```

Points to note

Use SMV (the function `smv`) when doing verification.

Getting started: yet another binary adder

A1. The file Arithmetic.hs (one of the modules of Lava, see http://hackage.haskell.org/package/chalmers-lava2000) contains the following definition of a binary adder:

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

where `adder` is a function that takes a `carryIn` and two binary numbers and adds them all up. Here, it is called with that `carryIn` set to `low`. This means that this function uses a full adder to add this `low` value to the two least significant bits of the the two binary

numbers. Define `binAdder1`, a slightly different implementation of the same function, that uses a halfAdder to add these two least significant bits, and then calls the function `adder` to do the rest. [**Hint**: the file `LavaLabRemoved12.hs` defines the type `Binary`, so that the type of `binAdder1` is `(Binary,Binary) -> Binary`. Fill in the missing parts of the given function.]

Check by (exhaustive) simulation that you have got the function right. [**Hint**: you may find the functions `int2bin` and `bin2int` useful for this purpose. See page 20 of the Lava Tutorial (LT). Note also that the function `simulateSeq` allows you to run many tests in sequence on a combinational circuit.]

Defining a connection pattern

A2. Define a connection pattern `linArray` that forms a linear array of a two-input one-output component. [**Hint**: Visualise a `row` that has had all of its downwards pointing outputs removed. (See Page 18 of LT.) Think of the values that flow from left to right as an accumulating parameter that gathers up the answer.]

```
Main> simulate (linArray plus) (0,[1..7])
28
```

Binary multipliers

A standard way to do binary multiplication is to first compute the *partial products* and then add the resulting binary numbers.

A3. Define a function `prods_as_bin` that takes two (lsb first) binary numbers a and b and returns the list of binary numbers that must be summed to produce the product of a and b. The first number should represent $a_0 * b$, the second $2 * a_1 * b$, the third $2 * 2 * a_2 * b$ and so on, where a_i is bit i of a. This can be checked by simply running the function on symbolic values:

```
*Main> prods_as_bin (varList 3 "a", varList 3 "b")
[[andl[a_0,b_0],andl[a_0,b_1],andl[a_0,b_2]],
 [low,andl[a_1,b_0],andl[a_1,b_1],andl[a_1,b_2]],
 [low,low,andl[a_2,b_0],andl[a_2,b_1],andl[a_2,b_2]]]
```

[**Hint**: The suggested solution uses a *list comprehension*, see
http://en.wikipedia.org/wiki/List_comprehension or
http://www.cs.nott.ac.uk/~gmh/chapter5.ppt (with thanks to G. Hutton). You may use some other form if you prefer, but list comprehensions are very handy for this kind of thing.]

Now we will study several ways to do mulitplication by adding up these lists of binary numbers.

A4. Define `multB`, which adds up the numbers using a linear array of binary adders.

A5. Formally verify, using Lava and SMV, that the resulting circuit is indeed a binary multiplier. Do this by comparing to the built-in Lava multiplier called `multi`. [**Hint.** Remember that the multipliers take as input two lists of bits, so you will have to go to some trouble to set their sizes for the verification. See p.25 in LT. Don't be over-ambitious about the maximum size that you try to verify. I find that 9 or 10 bit inputs give no problem on my laptop, but I didn't manage 16 bits. At the same time, verify a good variety of sizes and shapes.]

A6. Make a binary adder, `binAdder2`, that produces its output as a pair of the least significant bit and the remaining bits. Now build a binary multiplier from this component, and formally verify it. This could be viewed as an optimisation of the previous multiplier in which some outputs are produced earlier because they won't be affected by the rest of the circuit. The resulting circuit is a bit smaller than the previous one because it eliminates some unnecessary components. (We also wanted you to play with different connection patterns.) [**Hint**: You'll need to generate the partial products a bit differently. Draw a picture to help your thinking.]

A7. One way to reduce delay when adding several numbers is to use carry-save numbers, see http://en.wikipedia.org/wiki/Carry-save_adder or Figure 1(b) in the paper at http://www.cse.chalmers.se/edu/course/TDA956/Papers/CarrySaveAdderMults.pdf.

In Lava, a carry-save number can be represented as a list of *pairs* of bits, as defined in the type synonym `type CSV = [(Bit,Bit)]`. You can think of carry-save numbers as having, at each position, either a pair of equal weight bits or a pair of bits one of which has twice the weight of the other. You may choose either of these views. Please tell us which one you have chosen. (There seem to be nice solutions for both choices, and the main thing is to be consistent. Mary chooses equal weight bits and this is reflected in the example below.)

Make a component `cbPlusc` that adds a carry-save number of length $n-1$ and a binary number of length n to give a carry-save number of length n.

```
Main> simulate cbPlusc  (zipp (int2bin 4 15, int2bin 4 15), int2bin 5 31)
[(high,low),(high,high),(high,high),(high,high),(high,high)]
```

[**Hint**: Think about what happens if you `map` a full adder over some suitable input.]

Now use this component repeatedly to build a multiplier whose output is in carry-save form. Formally verify this multiplier too. [**Hint**: It is a good idea to convert the output of the multiplier to binary to enable the formal verification. Why do you think this is?]

Tree multipliers

A8. Another way to reduce a list of partial products down to one binary number is to use a binary tree of binary adders. Implement and verify such a multiplier. [**Hint**: This is trickier than it sounds because it is not trivial to get the lengths of intermediate results right. I found it useful to pad the partial products with a zero at the msb end, and to use a binary adder that drops the `carryOut`; there are probably other solutions.]

A9. A much better idea is to make a tree from components that reduce three binary numbers to two; this is known as a Wallace tree. Inspired by your `cbPlusc` component, make a circuit that takes binary inputs a, b and c and produces x and y such that $a+b+c=x+y$. Using this component, make a tree that reduces a list of binary numbers down to two. (We will return later to log depth ways to add two binary numbers.) Verify the correctness of the tree. [**Hint**: If you use formal verification, this is again most easily done by starting with two binary numbers, feeding partial products into the tree, and converting the output of the tree to binary, so that you actually build a complete multiplier.]

Extra assignment for extra credit. Note that you should complete tasks A1 to A9 first.

Choose at least one of

A10. Yet another way to build a fast multiplier is to generate the partial products not as a list of binary numbers but as a list of lists of bits, with one list per bit-weight.

```
*Main> simulate prods_by_weight (int2bin 4 5, int2bin 5 3)
[[high],[high,low],[low,low,high],[low,low,high,low],
[low,low,low,low],[low,low,low],[low,low],[low]]
```

Build a cell that takes a pair containing some `carryIn` bits of weight k, and a list of bits of weight k, and produces a pair containing **2** bits of weight k and a list of `carryOut` bits of weight $k+1$. Using a `row` of these cells, build a *column compression* multiplier that, again, produces its output as a carry-save number. [**Hint.** There are many ways to build the cell. Anything that is functionally correct will do. If you are ambitious, you could try to build a Dadda-like multiplier, or one following the "overturned stairs" or TDM methods.]

A11. Use Lava to design and analyse a circuit that interests you.

If you have done all of this, you might want to investigate performing delay analysis on your multipliers using Non-Standard Interpretation (see lec. 1 and LT). It will mess up your code a bit because you will have to parameterise on either half- and full-adders or on gates, depending on what you want to count. But it is a good way to gain understanding of the various multiplier options.

Formalia

Submit via the Fire system. The rules are the same as for VHDL Lab. Work in pairs. Do not copy solutions or parts of solutions. But do discuss your ideas.

Good luck!