Hardware Description and Verification
Lava Exam

Mary Sheeran

May 14, 2011

Introduction

The purpose of this take-home exam is to give you further practice in writing Lava definitions of recursive circuits. It places particular emphasis on connection patterns. You will gain experience of exploring a design space using Lava, and will again use formal verification and other circuit analyses.

Important information

This is not a normal assignment, but part of the course examination. You are to do all of the work yourself. You are *not* allowed to cooperate or copy solutions (whole or partial). You should not even discuss your solution with anyone else. Act as if you were sitting in an exam hall with someone watching you. Cheating is unprofessional, and morally wrong. We will take a very dim view of any cheating that we discover, but feel hopeful that there will not be any.

If you have questions about how to interpret the assignments, or have problems with the tools, do not hesitate to contact us. Also, if you get completely stuck, **please** contact us (and not your friends). This is a **much** better idea than just giving up! In that case, any assistance that we give will be noted and may influence your grade.

The required assignments are numbered **A1-A10**. It will be possible to pass by doing a good job on assignments **A1-A6** and one of **A8** and **A10**. Although this document is long, your answers can be short! You can increase your grade on this exam by up to 20% by doing the extra assignment, **A11**.

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. You can save some time by starting from the file `LavaExamRemoved11.hs`, which is provided.

In comments in the same Lava file, 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, verification or other analysis. Give us as much information about what you have done as possible, as this eases the problem of judging the quality of your work. Remember, though, that we don't expect perfection. Hand in whatever you have got at the deadline, even if you are not at all happy with it.

You may use definitions and other code fragments from the Lava tutorial and from the file `Efile11.hs`, which is available, together with important papers and documents (including this one) on the Assignments page on the course web page.

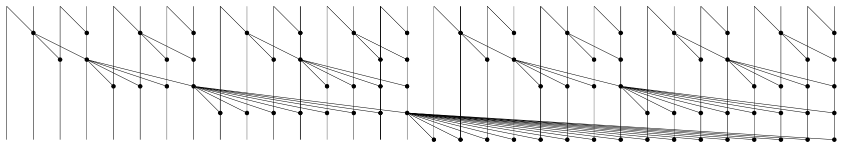


Figure 1: The Sklansky parallel prefix network for 32 inputs

Adders and parallel prefix circuits

Lecture 4 on Lava showed a sequence of steps (or a derivation) from a very simple ripple carry binary adder to a more sophisticated adder that uses a so-called prefix network to calculate the carries in the adder. By first calculating all the carries and then using those carries to calculate the sum bits, one can speed up binary addition significantly. This idea is due to Brent and Kung and their classic paper is provided with this exam (on the Assignments page).

The adders discussed in Lecture 4 on Lava are named `adder0` to `adder7`. All code referred to here is in file `Efile11.hs`. One of the steps in the derivation of the final prefix adder relies on the fact that the function `fullAdd1` does indeed correctly define a full adder.

A1. Verify by exhaustive simulation in Lava that `fullAdd1` implements exactly the same function as the built-in full adder in Lava (see Lava.Patterns). *(1 points)*

A2. Perform the same check using formal verification (and SMV) in Lava. *(1 points)*

A3. Formally verify that the "dot" operator, `dotOp`, is indeed associative, that is that it satisfies a property of the form $a \circ (b \circ c) = (a \circ b) \circ c$. *(2 points)*

Hint: Write a property that checks associativity for any binary operator, and then apply it to `dotOp`.

A4. Using formal verification in Lava, check that `adder0` and `adder6` have identical behaviour for a range of input sizes. *(2 points)*

Given an input $[a_1, a_2, \ldots, a_n]$, the problem of calculating, in parallel, all of a_1, $a_1 \circ a_2$, $a_1 \circ a_2 \circ a_3$, and so on, up to $a_1 \circ a_2 \circ \ldots \circ a_n$ for an associative operator $\circ$ is called the *parallel prefix problem.* It is this problem that we need to solve in order to make a fast adder based on Brent and Kung's dot operator. The prefix network is used for the calculation of the carries. We are going to look at a number of different prefix networks. Note that in our final adder descriptions (see for example `adder6`), the parts of the circuit that prepare inputs for the carry computation (the `gpC` components) and that compute the final sums (the `sumC` components) are each very small and entail only one gate delay. So the final cost of the adder, both in area and delay, is completely dominated by the prefix network.

Making parallel prefix circuits

Sklansky

In a Lava lecture, you already saw the most well-known way of computing parallel prefix, which is usually called after Sklansky, see Figure 1. It is a divide and conquer approach. We compute the parallel prefix of each half of the inputs, and then combine the last output of the lower parallel prefix with each of the outputs of the upper one.

In the file `Efile11.hs`, the definition of the Sklansky pattern is shown. It is built from components that we call fans (see the lecture slides). An Example of such an operator is `pplus`, see the file `Efile11.hs`.

You are provided with a Haskell file called `DrawPP11.hs` that allows you to draw pictures like Figure 1 from your Lava descriptions. This program is a hack, written by Mary late one

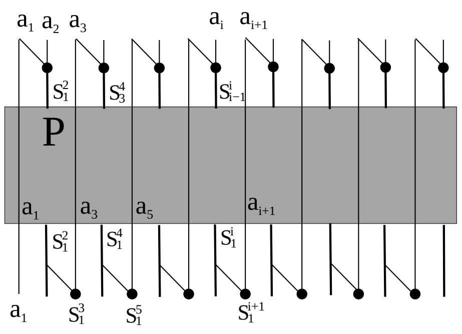


Figure 2: One way to build a larger prefix network out of a smaller one. This is the basis of the Brent Kung construction. S_i^j means $a_i \circ a_{i+1} \ldots a_j$.

night. It is not beautiful but it seems to work! Using `DrawPP11.hs` you can produce a file in FIG format for a given parallel prefix network (e.g. by loading the file into Lava and typing at the prompt `drawPP "skl" skl 32`). On Linux machines, the resulting file, `skl32.fig`, can either be viewed with `Xfig` using the command `xfig skl32.fig`, or converted to pdf using `fig2dev -L pdf skl32.fig skl32.pdf`. Rotate the diagram clockwise to make it easier to see. `Xfig` can also produce pdf and many other formats, using `Export...` in the File menu. Make sure, now, that this all works for you, and that you can view the resulting file, having converted it to pdf if you wish. Get in touch with us if you have problems with this.

The file `Efile11.hs` also provides a Haskell function called `check` that allows you to check that a pattern correctly constructs a prefix network (without calling any external tool):

```
Main> check skl 32
True
```

The file `Efile11.hs` also defines a non-standard fan component `delFan`, which can be used to measure the depth of a connection pattern. For example, the depth of the `skl` pattern can be measured thus:

```
> simulate (skl delFan) (replicate 16 0)
[1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,4]
```

A5. Adders `adder6` and `adder7` are nearly identical, differing only in the patterns (`ser` and `skl`) used to create the prefix networks that compute the carries. Define a function

```
adder8 :: PP (Bit,Bit) -> [(Bit, Bit)] -> ([Bit], Bit)
```

that is parameterised on this pattern. *(2 points)*

The Brent Kung parallel prefix network

The second well-known recursive pattern for parallel prefix is due to Brent and Kung. The Brent Kung paper describes it in terms of a forwards and a backwards tree, but we can also think of it as a recursive network. (Note that our generated diagrams are very like Brent and Kung's except that they are rotated by 180 degrees! In the Brent Kung pictures, data flows from bottom to top, while in ours data flows from top to bottom.) Note that the fanout per level in the classic Brent Kung network is only 2.

Figure 2 shows a recursive view of the Brent Kung pattern. The inputs are at the top, with the least significant input on the left. The smaller network P is *wrapped* in two more layers of operators, above and below it. If the inputs to the network are $[a_1, a_2, \ldots, a_n]$, then the topmost layer of operators should apply to the 2-lists $[a_1, a_2]$, $[a_3, a_4]$, up to either $[a_{n-1}, a_n]$ or $[a_{n-2}, a_{n-1}]$, depending on whether n is even or odd respectively. Call the results of those

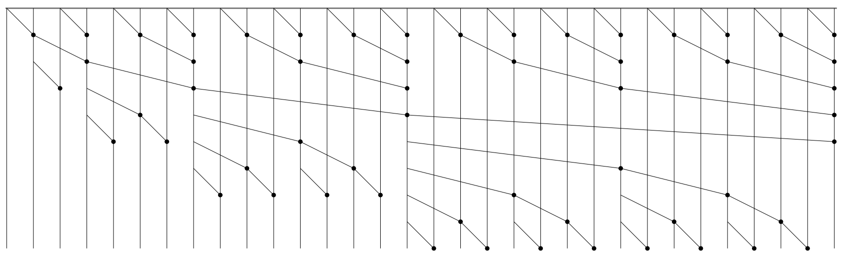


Figure 3: The Brent Kung pattern (for 32 inputs) as drawn from the Lava description

applications $[b_2, b_4, b_6 \ldots]$. The smaller P network applies to exactly this list, to produce the list $[c_2, c_4, c_6 \ldots]$. Both of these lists are shown as thicker lines in the diagram, entering and leaving the box labelled P. The inputs $[a_1, a_3, a_5, \ldots]$ also pass over the P network, and are shown as thinner lines. The final (bottom-most) layer of operators should then operate on the 2-lists $[c_2, a_3]$, $[c_4, a_5]$ and so on, to give d_3, d_5 etc. The output of the entire network is $[a_1, c_2, d_3, c_4, d_5 \ldots]$.

A6. Define the Brent Kung pattern in Lava:

```
bKung :: PP a
bKung f [a] = ...
bKung f as  = ...
```

Use the function `check` to verify that your definition is correct for a range of input sizes. Also measure the depth of your circuit by using the `delFan` component. Formally verify a fast adder built using the Brent Kung prefix pattern (for example using your `adder8` function). *(6 points)*

Hint: See `Efile11.hs` for some hints (but you are free to implement `bKung` any way you like). If you get it right, the picture for 32 inputs should look like Figure 3.

A7. It is possible to define a function to count the number of operators in a prefix pattern as follows:

```
pattSize patt n = simulate ((patt (mkFan opCount)) ->- sum) (replicate n 0)
```

The idea is to count up the operators along each "wire" and then add them up using `sum`. Define the operator

```
opCount :: (Signal Int, Signal Int)
```

to make this work. *(2 points)*

Generalising Brent Kung

We have been considering (standard) versions of Brent Kung that work on sub-blocks of the input of length 2.

A8. Make a more general Brent Kung pattern that takes a parameter `k` to indicate how wide those blocks should be. (4 points)

```
bKungG :: Int -> PP a
```

Hint: Use small serial prefix networks at the top, and fans at the bottom. One result that I got for that and block width 3 is shown in Figure 4.

A9. Generalise the above pattern to allow the small serial prefix networks across the top to be replaced by any prefix pattern (which is to be a parameter).

```
bKungG2 :: Int -> PP a -> PP a
```

(4 points)

Figure 5 shows a general recursive structure for prefix networks, and it may guide your design of the generalised Brent Kung patterns. The T_i are prefix networks and the F_i are fans. Here, though, we are assuming that all blocks across the top have equal width, k, with the possible exception of the rightmost one. An example of a prefix pattern where the small networks across the top have width 4 and are Sklansky networks is shown in Figure 6

A10. You now have the means to describe a great variety of prefix networks. File `Efile11.hs` also contains definitions of the Ladner Fischer construction, in case you want to use that as a building block. Assuming a 64 input network, say, explore the design space to find a network that is "lagom", that is reasonably shallow, not too big and with fanout that is not too large. (Decreasing depth increases both the number of operators and the fanout, so it is a question of finding a sweet spot that is a good compromise between conflicting demands.) Document your chosen network by giving its Lava definition. *(6 points)*

Extra assignment for PhD students. Can be used by other students to gain extra credit (up to 20%).

A11. Choose at least one of

(a) An alternative approach to making adders go fast is to use pipelining; see for example the slides numbered 10 in the following presentation: `http://perso.ens-lyon.fr/bogdan.pasca/resources/talks/addition_slides.pdf`. (The link is also given on the Assigments page.)

Define and verify a pipelined adder in Lava.

(b) Design and formally verify an interesting circuit in Lava. One possible choice would be to use the paper by Knowles called "A family of adders" as inspiration. (It is widely available on the web.)

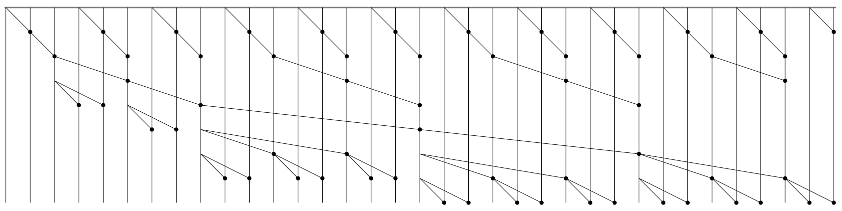
To submit

Use the Fire system as before.

You will be reviewed on the following:

1. The correctness and readability of your circuit descriptions
2. The correctness of your applications of formal verification
3. The clarity of your documentation (which should be brief).

Good luck!



35 lines, 8 stages, 60 operators, 3 maximum fanout.

Figure 4: Allowing fanout 3 in Brent Kung

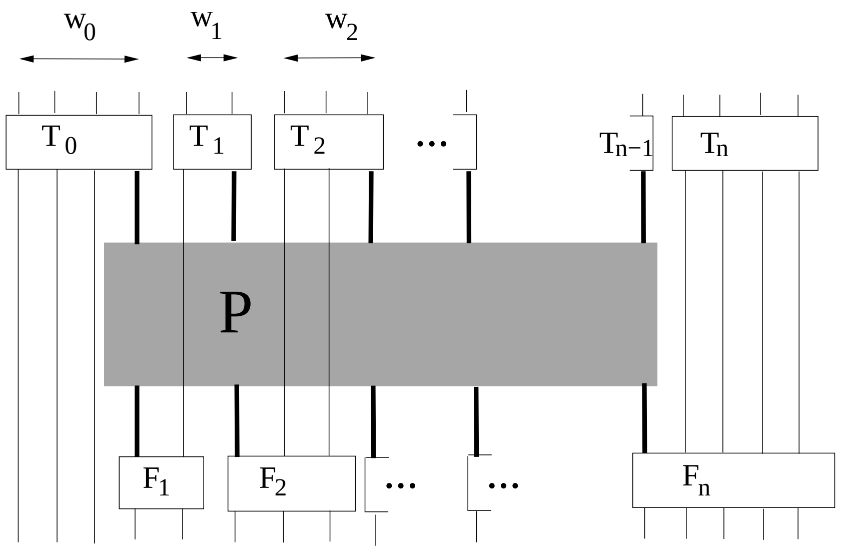
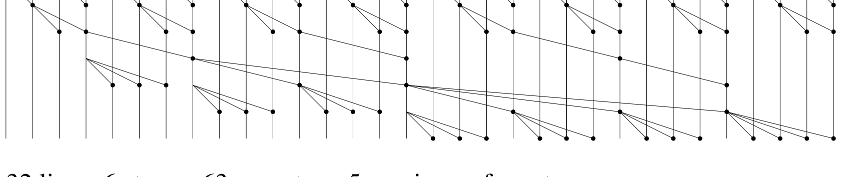


Figure 5: Recursive decomposition of a prefix network.



32 lines, 6 stages, 63 operators, 5 maximum fanout.

Figure 6: A network in which the small prefix networks (of width 4) across the top are Sklansky networks.