# The Design and Verification of a Sorter Core

Koen Claessen[1], Mary Sheeran[1], and Satnam Singh[2]

[1] Chalmers University of Technology
[2] Xilinx, Inc.

**Abstract.** We show how the Lava system is used to design and analyse fast sorting circuits for implementation on Field Programmable Gate Arrays (FPGAs). We present both recursive and periodic sorting networks, based on recursive merging networks such as Batcher's bitonic and odd-even mergers. We show how a design style that concentrates on capturing *connection patterns* gives elegant generic circuit descriptions. This style aids circuit analysis and also gives the user fine control of the final layout on the FPGA. We demonstrate this by analysing and implementing four sorters on a Xilinx Virtex-II$^{TM}$ FPGA. Performance figures are presented.

## 1 Introduction

This paper describes the application of various formal and informal techniques to the design, implementation, optimisation and verification of a high speed sorter core realised on a large field programmable gate array (FPGA). Customers who buy FPGA cores expect them to have high performance and to have been carefully verified. Examples of the application of high speed sorting are graphics algorithms for rendering and ray tracing.

The design of the sorter core is based on recursively described butterfly networks which are composed to realise periodic sorters. There are a large number of different but similar sorting network designs, resulting in circuits of varying performance (depending on the lengths of intermediate wires). Having a design language that is well-suited to describing these networks has helped us to explore the design space far more effectively than is possible using conventional hardware description languages. We present four designs and instrument their performance. The user of the sorter core specifies the required speed performance, area requirements, latency and pipelining behaviour. Based on these requirements the sorter core selects one of the four sorter implementations presented here.

To produce an efficient sorter network on an FPGA, one must carefully manage the intermediate wire lengths and also make effective use of the available silicon resource. We demonstrate how a layout combinator based style of description allows us to generate a compact layout without the tedious calculations that are necessary in a conventional HDL.

The design environment used to describe and verify our circuit cores was developed at Xilinx and at Chalmers University, and is called Lava [3]. Circuit descriptions in Lava are written in the functional language Haskell [6].

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 355–368, 2001.

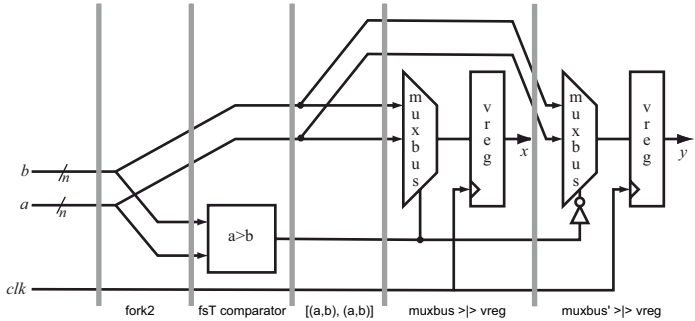


**Fig. 1.** Architecture of the 2-sorter

## 2 Design and Verification of a 2-Sorter

The basic element of the sorters and mergers that we present in the following sections is a two-sorter circuit. This circuit takes a pair of $n$-bit numbers and returns the pair sorted into ascending order. The butterfly networks implemented on FPGAs are pipelined. This is achieved by registering the output of each 2-sorter.

Figure 1 shows the implementation architecture for the 2-sorter. This implementation uses a comparator to determine which of `a` or `b` is greater. The result of the comparator is used as the select signal to two bus-multiplexors. The outputs of the bus-multiplexors are registered to allow the construction of pipelined sorters.

The top-level Lava description of the two sorter is:

```
twoSorter clk = fork2 >-> fsT comparator >-> condSwap clk
```

This Lava description describes the circuit shown in Fig. 1 by composing in series several sub-circuits. The serial composition infix combinator is written as `>->`. This connects the output of the circuit on the left to the input of the circuit on the right. Furthermore, the circuits are laid out horizontally with a left to right information flow. Note how combinators compose *behaviour* and *layout*. Lava also provides combinators for right to left serial composition (`<-<`), top to bottom serial composition (\/), bottom to top (/\) and overlaid layout (`>|>`). There are also combinators for four sided tiles and many kinds of elaborate layout and wiring for real circuits have been successfully described using convenient combinators.

The `fork2` circuit simply duplicates its input, as shown in Fig. 1. The `fsT` combinator takes a circuit as a parameter and applies it to the first element of a pair leaving the second element unchanged. The comparator is implemented by using a subtracter laid out vertically (its definition is not given in this paper). The next stage uses `fsT` with `comparator` to perform a comparison on the first element of the forked value, namely `(a,b)`. The next three stages implement a conditional swap circuit which will swap the values `(a,b)` if `a` is larger than `b`.

The source for the conditional swap circuit is shown below:

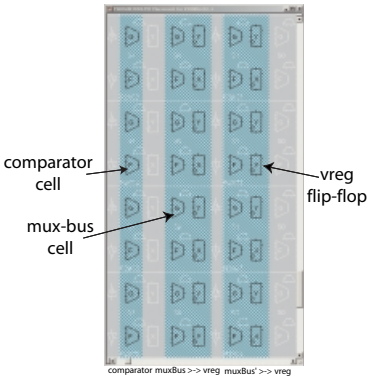


**Fig. 2.** FPGA floorplan of an 8-bit 2-sorter

```
condSwap clk  =
  fork2List >-> hpar [muxBus >|> vreg clk, muxBus' >|> vreg clk]
```

This circuit is build using the `hpar` combinator which lays out a list of circuits horizontally. In this case, the two circuits laid out next to each other are bus-multiplexors that use a select signal to determine which input to transfer to the output. The `fork2List` combinator duplicates its input into a two element list. Each bus-multiplexor has its output connected to a register bus and the two circuits are overlaid so that they occupy the same locations on the FPGA (there is enough space in each cell to perform the multiplexing and registering). To avoid having to explicitly realise the inverter required for the second multiplexor a variant `muxBus'` is used which switches in the opposite sense from `muxBus`. This optimisation allows for a very compact floorplan as shown in Fig. 2 for a Xilinx Virtex ™ FPGA.

The verification of the two-sorter was conducted using an in-house system developed for the formal verification of intellectual property cores [8]. A technology specific equivalence checker has been produced which takes as input two EDIF netlists and tries to establish their equivalence using sequential equivalence checking. Using this tool we have shown that the Lava implementation of the 2-sorter has the same behaviour as a golden VHDL behavioural description.

## 3 Describing Networks

Many circuits have a clear recursive or iterative pattern of construction. Typical examples are multipliers, mergers, sorters, interconnection networks and transforms (notably the Fast Fourier Transform). These are important functions, and indeed it turns out that many of the functions that we would like to implement as cores for Digital Signal Processing (DSP) display this kind of regularity.

For this reason, we have developed design, verification and implementation methods that are particularly suited to these regular circuits. The overall goal

is to make a fast route to efficient FPGA implementations. Here, we consider how to describe recursive and iterative networks, taking mergers and sorters as examples. This means that we are particularly interested in *butterfly networks* like that shown on the right of Fig. 3, and in variants on them. As a first step, we introduce some useful connection patterns.

Often, we want to feed each of the pairs in a list of even length to different copies of a two-input two-output component, producing a list of the same length. We define:

```
evens f []       = []
evens f (a:b:cs) = f [a,b] ++ evens f cs
```

Here, `f` is the component in question, given as a parameter to the circuit. `[]` is the empty list, giving a base case. In the step, `a` and `b` are the first two elements of the input list, and `f [a,b]` is the output of the `f` component. That list is concatenated (using `++`) to the result of applying `evens f` to the rest of the input list, `cs`.

A circuit in which `g` works on the bottom half of the input list and `h` on the top half is written `parl g h`. The special case in which the two components are the same arises often and so gets its own abbreviation.

```
two g = parl g g
```

We can simulate this circuit on the Lava prompt. As an example, we simulate `two reverse`, where `reverse` is the wiring pattern that reverses its inputs.

```
Lava> simulate (two reverse) [1..16]
[8,7,6,5,4,3,2,1,16,15,14,13,12,11,10,9]
```

We also introduce the pattern `ilv`, for *interleave*. Whereas `two f` applies `f` to the top and bottom halves of a list, `ilv f` applies `f` to the odd and even elements (see Fig. 3). We define it in terms of the wiring pattern *riffle*, which performs the perfect shuffle on a list. Think of taking a pack of cards, halving it, and then interleaving the two half packs (as you do before you deal out the cards). If you now *unriffle* the pack, you reverse the process, returning the pack to its original condition. (This is somewhat more difficult to accomplish with aplomb at the poker table.)

```
Lava> simulate riffle [1..16]
[1,9,2,10,3,11,4,12,5,13,6,14,7,15,8,16]

Lava> simulate unriffle [1..16]
[1,3,5,7,9,11,13,15,2,4,6,8,10,12,14,16]
```

Note that unriffling the sequence from 1 to $n$ divides into its odd and its even elements. We use this fact to define `ilv` in terms of `two`.

```
ilv f = unriffle >-> two f >-> riffle
```

Now we are in a position to define a connection pattern for butterfly circuits.

```
bfly 1 f = f
bfly n f = ilv (bfly (n-1) f) >-> evens f
```

The smallest butterfly is just a single `f` component with two inputs and two outputs. A butterfly of size $n$, for $n$ greater than zero, consists of two interleaved

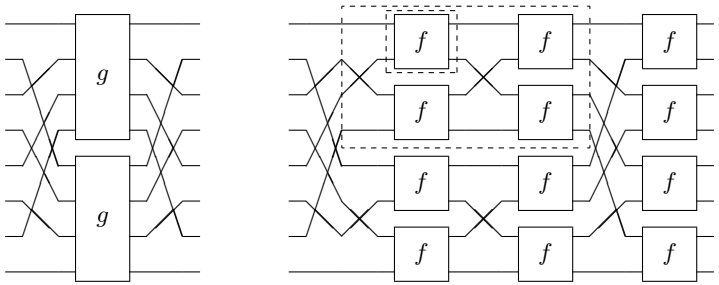


**Fig. 3.** `ilv g` and `bfly 3 f`

butterflies of size $n-1$, the output of which is fed into a stack of `f` components, which is made using `evens`.

This pattern corresponds to a divide and conquer algorithm: there are two recursive calls on sub-parts of the input, and then a final phase (the call of `evens`) in which the results are combined. This butterfly-shaped connection pattern is shown on the right in Fig. 3. The figure indicates one of the smaller butterflies using dotted lines. This butterfly is interleaved with another one. (Compare with the diagram on the left of the figure.) The call of `evens` can be seen on the right of the figure.

### 3.1 Batcher's Bitonic Merger and Sorter

One of the best known uses of the butterfly network is in the building of mergers and sorters based on a two-input two-output comparator, that is a two-sorter. It turns out that `bfly n cmp` sorts some input patterns, including those whose first half is sorted and second half is sorted into reverse order (provided that `cmp` is a two-sorter). This allows us to build a recursive sorter.

For lists of length 1, the base case, sorting is just the identity function (`id`). Otherwise, we make two recursive calls to smaller sorters, and reverse the output of the second, feeding the result into a merger. The merger (`bfly n cmp`) is known as *Batcher's bitonic merger* [1].

```
sortB 0 cmp = id
sortB n cmp = parl (sortB (n-1) cmp) (sortB (n-1) cmp >-> reverse) >->
              bfly n cmp
```

The recursive structure of Batcher's Bitonic Sorter is illustrated in Fig. 4. In part (a) of the figure a sorter is designed by recursively sorting the two halves of the input and then merging the result (after reversing one of the sub-sorts). Part (b) shows that a merger can be made from a butterfly of two sorters. The two sorter component is shown as a box with 2S written inside it. The merger shown here is Batcher's bitonic merger. Part (c) shows that one of the smaller sorters can be decomposed using exactly the same strategy i.e. two sub-sorters and a merger. Part (d) shows that the merger inside this sub-sorter is just a butterfly of size 2. The resulting sorting network is shown in Fig. 5.

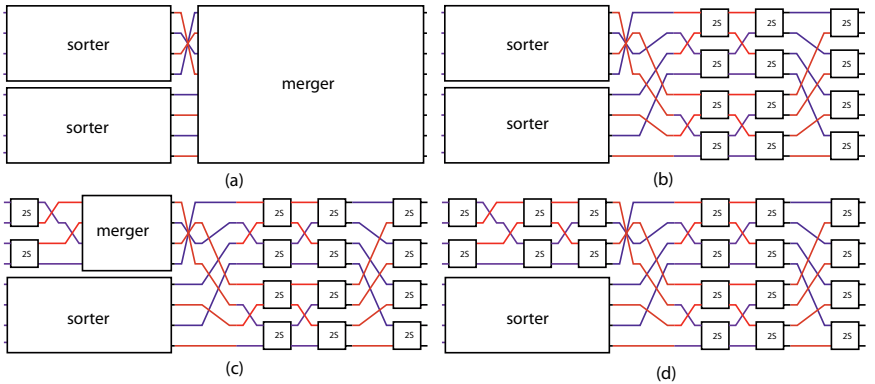


**Fig. 4.** Recursive structure of Batcher's Bitonic Sorter

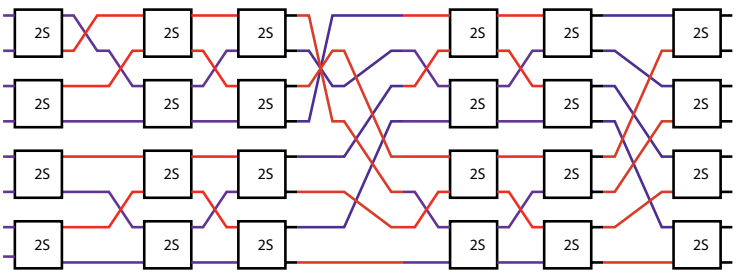


**Fig. 5.** Batcher's Bitonic Sorter `sortB`

Note that our sorter is parameterised on the comparator component. So, for instance, if `cmp` is a two-sorter on integers, then `sortB n cmp` is sorter on integer lists of length $2^n$, but later we will plug other circuit-level comparator components into the same `sortB` function. So we have really designed the connection pattern that must be used to connect comparators. We have not in any way tied ourselves down to comparators of a particular type. We will later use this fact more than once, when visualising merging networks and when verifying sorters.

### 3.2 Batcher's Odd Even Merger and Sorter

The other well-known merging network is Batcher's *odd even merger*. To describe it, we need a another connection pattern, `mid`. The circuit `mid f` passes the first and last inputs of its input list through unchanged, and applies `f` to the remaining elements. For example, simulating `mid reverse [0..7]` gives `[0,6,5,4,3,2,1,7]`.

Now, the connection pattern for the odd-even merge is defined as

```
mergeOE 1 cmp = cmp
mergeOE n cmp = ilv (mergeOE (n-1) cmp) >-> mid (evens cmp)
```

The subcircuit `mid (evens cmp)` on the right places components not on even pairs, but on *odd* pairs. For comparison, look back at the definition of `bfly`, the connection pattern for the bitonic merger. The pattern is almost identical! The big difference is that the butterfly has `evens cmp` as its last column, while the odd even merger has `mid (evens cmp)`. The reader is encouraged to sketch an odd even merger for 8 inputs, along the lines of our picture of the butterfly in Fig. 3.

The odd even merger sorts inputs whose top and bottom halves are sorted, so the definition of the corresponding recursive sorter is short and sweet.

```
sortOE 0 cmp = id
sortOE n cmp = two (sortOE (n-1) cmp) >-> mergeOE n cmp
```

## 4 Visualising Networks

We have introduced some merging and sorting networks, and made claims about their behaviour. How do we convince ourselves that we have got our mergers and sorters right?

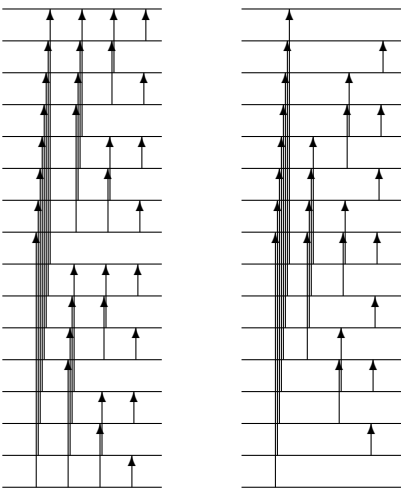

**Fig. 6.** The topology of the butterfly and odd even merge networks

A good way to check that one has described these standard networks correctly is to draw pictures of them and compare with reliable presentations in the literature. A standard way to visualise sorting or merging networks is shown in Fig. 6. Values in the network flow from left to right. The arrows indicate comparators, which operate on values flowing on the wires at each end of the arrow (possibly swapping them). These figures show Batcher's bitonic and odd even mergers for 16 inputs.

We have generated these pictures using a simple non-standard interpretation. We replace the standard comparator by one where each "wire" carries both a number (as before) and a list of pairs of numbers, the comparisons done so far in the circuit. The comparator records which two numbers it compares, appending the pair to the comparisons list on one of its outputs. In addition, we add depth indicators to the circuit. A depth counter is incremented each time it flows through such a marker. Then, the new circuit is simulated on a sorted input list. We simply write some Haskell code to naively generate pictures from the resulting output. We exploit the fact that we have a full programming language available to code up lightweight analyses that help with the design problem at hand. Here, we exploit the fact that we have written generic descriptions of the connection patterns for the mergers. This enables us to plug in non-standard components for use in circuit analysis.

## 5 Verifying Mergers and Sorters

The recursive structure of the mergers and sorters encourages us to reason about them by induction. At present, we perform such induction proofs by hand (although it is our ambition to provide automatic assistance). For examples of hand proofs about sorting and permutation networks, see for example references [5,9, 10]. What we have available in the current Lava system is support for reasoning about *fixed size circuits*. What we tend to do in practice is to use this facility to check instances of more general conjectures – a sanity check that can be a great time-saver.

A network sorts correctly if (1) it always produces sorted output, and (2) if the output is a permutation of the input. In performing automatic proofs about fixed-size networks of comparators, we are fortunate in being able to use the zero-one principle [7]: A sorting network built only of wiring patterns and two-input two-output comparators sorts arbitrary numbers if it sorts numbers in {0,1}. This remarkable fact means that we only need to check that our networks, when built from combinational two-bit comparators (`twoBitSort`, which consists of just one *or* and one *and* gate) sorts lists of bits. Here, we again see the advantage of having generic connection patterns, into which we can plug many components.

A list of bits is sorted if implication holds between adjacent bits. A circuit permutes an input list of bits, if there are equally many ones in the input as there are in the outputs. If these two checks return true for all inputs, then we have a sorter. (An alternative would be to compare our sorters with a known correct sorter. We often perform such equivalence checking, but have chosen not to do so in this paper.)

To check the first sorting network property, we first define in what case a list of bits is sorted.

```
sorted []          = high
sorted [x]         = high
sorted (x1:x2:xs) = (x1 ==> x2) <&> sorted (x2:xs)
```

Then, we define the property, which is parameterised by a sorting network `sort`, and a natural number `n`.

```
prop_Sorts sort n =
  forAll (list (2^n)) $ \ inp ->
    sorted (sort n twoBitSort inp)
```

Read this property definition as: "For all lists of size $2^n$ called `inp`, the output of the sorting network `sort` is sorted, when given `inp` as input".

We can check these kinds of properties using external tools such as SAT-solvers and BDD-based tools. In order to do so, we have to specify at what size we want to verify the property. The logical level at which these tools work is not powerful enough to perform the verification for all sizes at once. We use symbolic evaluation to generate an input file to the external tool, which contains the unfolded definition of the circuit and the property. As an example, we present the verification of the sortedness property for size 4 (that is 16 inputs), using Prover Technology's propositional prover.

```
Lava> verify (prop_Sorts sortB 4)
Proving: ... (t=0.5) Valid.
```

Given a suitable circuit `count`, which outputs a binary number indicating how many of its inputs are ones, the permutation property can be easily formulated as below. We show the verification for size 5 (64 inputs) using the BDD-based tool VIS.

```
prop_Permutes sort n =
  forAll (list (2^n)) $ \ inp ->
    count inp <==> count (sort n twoBitSort inp)

Lava> vis (prop_Permutes sortOE 5)
Vis: ... (t=3.0) Valid.
```

Unfortunately, the verification of the two properties presented above runs out of steam at around 64 inputs, for both SAT-solvers and BDD-based tools. The problem is that the verification problem simply becomes too hard for current-day technology.

However, all is not lost. If we did a proof of the sorters by hand, we would use inductive reasoning. An inductive proof of the correctness of for example Batcher's odd even sorter clearly relies on the fact that the merger sorts two appended sorted lists of equal length. We can actually check this conjecture for fixed sizes of the merger!

The property that we then want to check has a list of bits as input, and a single bit as output. That bit should be 1 if the property holds of the given merger (which is the parameter `merge`). So, if `out` is the output of the merger when it has a bit-sorter as its component, we check that `out` is sorted if the two halves of `inp` are.

```
sortsTwoSorted merge n inp = ok
  where
    (inpL, inpR) = halveList inp
    out          = merge n twoBitSort inp
    ok           = (sorted inpL <&> sorted inpR) ==> sorted out
```

To make this into a property that can be checked by a verification tool dealing with fixed sized circuits, we must set the size of the list:

```
prop_SortsTwoSorted merge n =
  forAll (list (2^n)) $ \ inp ->
    sortsTwoSorted merge n inp
```

It is read as "for all lists of size $2^n$ called `inp`, the merger sorts two appended sorted lists". This property is rather easy to check for for example Prover Technology's propositional prover. For example, size 7 (that is 128 inputs) takes only about 3 seconds.

These results seem to indicate that monolithic proofs of sorting networks are hard for both SAT-solving and BDD-based methods, and that splitting the proofs up in smaller lemmas can help. Concluding, we were able to verify the two sorting properties of our sorting networks for small sizes (up to 64 inputs). The lemmas about the mergers have been verified for more than 256 inputs.

## 6 Periodic Networks

We have seen how to give elegant descriptions of two well-known recursive sorters. Now, we turn our attention to the so-called periodic sorters. These are sorters that can be made by composing a number of identical circuits. The best known of these sorters is odd even transposition sort. For $2N$ inputs, and `cmp` a comparator, $N/2$ copies in series of the circuit

```
evens cmp >-> mid (evens cmp)
```

makes a sorter. But this is a rather large sorter! Interestingly, there are mergers that can be composed in series to give sorters that are only about twice as big as the recursive sorters that we have seen. We are attracted by the prospect of building very regular, very fast sorters out of a single merger component. We can then concentrate optimisation efforts on that merger, aiming for a small footprint on the FPGA. Also, having a single merger makes it possible to experiment with space/time trade-offs, by reusing a single component repeatedly over time.

We first check to see whether or not the bitonic merger can be composed in sequence to make a sorter. (The combinator `hrep` puts $n$ copies of circuit in serial composition, horizontally.)

```
bflyCompose n cmp = hrep n (bfly n cmp)
```

Verifying `prop_Sorts bflyCompose n` fails already when `n` is 2, indicating that for example `[low,high,low,high]` is a counterexample. (This list passes through each butterfly unchanged.) So we need to look further.

Sheeran has earlier studied the periodic balanced merger of Dowd et al [4,9]. The Ruby descriptions can be directly translated into Lava. We introduce a new and somewhat mysterious connection pattern `vee` related to `ilv`, and build a connection pattern similar to the butterfly, but replacing `ilv` by `vee`.

The wiring pattern `alt` swaps every second of the even pairs of a list. So, for example, the list `[1..16]` is permuted to

```
[1,2,4,3,5,6,8,7,9,10,12,11,13,14,16,15]
```

by the application of `alt`.

```
vee f = alt >-> ilv f >-> alt

vfly 1 f = f
vfly n f = vee (vfly k f) >-> evens f
```

Here, we need a generated picture to help understanding, see Fig. 7, which illustrates `vfly 4 cmp`. It becomes clear why this is called the balanced merger. This rather beautiful and symmetrical merger can be composed to make a sorter.

```
sortV n cmp = hrep n (vfly n cmp)
```

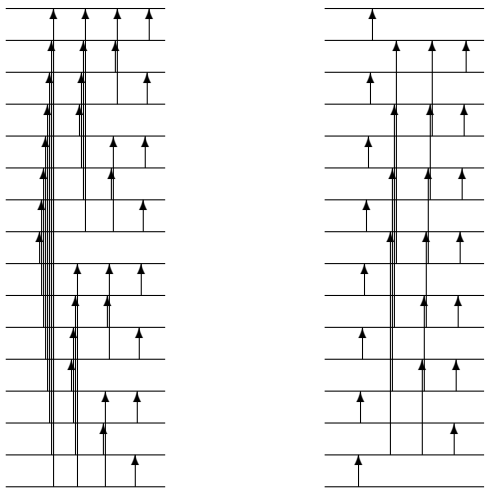

**Fig. 7.** The balanced merger and a periodic variant of the odd even merger

The same `prop_Sorts` that we used before confirms that `sortV 4`, that is four copies of `vfly 4` in series, does indeed construct a sorter for 16 inputs (using Prover Technology's basic propositional prover). The proof for 32 inputs takes too long. On the other hand, we can quickly confirm that the periodic balanced merger sorts two interleaved sorted lists, even for 128 inputs.

By experimenting with variants on the `ilv` connection pattern in the odd even merge, we have found another appealing periodic merger. You can think of `ilv` as marking the odd and even elements of the input and output lists with different colours, and then operating separately on each colour. What if we first divide the list up into pairs, and then mark alternate pairs with different colours? Then, one of the components operates on elements 0,1,4,5,8,9 etc. while the other operates on the remaining elements. We define

```
que f = ilv unriffle >-> two f >-> ilv riffle
```

You are encouraged to sketch this connection pattern.

Now, we replace each `ilv` in the odd even merge pattern by `que`.

```
qfly 1 f = f
qfly n f = que (qfly (n-1) f) >-> mid (evens f)
```

The resulting comparator network is shown on the right in Fig. 7. It is, we believe, a periodic network proposed by Canfield and Williamson [2]. Note that the lengths of the stretched comparators are shorter than in any of the networks that we have seen. This is the best periodic merger that we have found so far. We call the resulting sorter `sortQ`. Again, this network sorts two interleaved sorted lists, and this can be checked using a SAT-solver.

Why, though, does this merger compose to form a sorter? The key insight is that the merger not only transforms lists satisfying `ilv sorted` to lists satisfying `sorted`, but also lists satisfying `ilv (ilv sorted)` to lists satisfying `ilv sorted`. In general, the merger increases sortedness by "removing one ilv". So a sequence of $n$ mergers takes an unsorted list (of length $2^n$), that is a list that repeatedly interleaves $2^n$ sorted singleton lists ($n$ calls of `ilv`) to a sorted list (no calls of `ilv`). A useful lemma in this proof is the fact that

```
que (ilv f) = ilv (ilv f)
```

Our recursive descriptions of networks are very different from the standard ways of describing networks in the extensive literature on sorting networks. We have a language with which to describe networks! We have found that this connection pattern oriented style of description has enabled both informal and formal proofs about our networks. In addition, this style of description gives us an elegant way to control the final layout when implementing circuits on FPGAs. This is the topic of the following section.

## 7 Implementation

Four of the sorters presented in the previous section have been implemented on a Xilinx Virtex-II XC2V3000 FPGA to allow us to evaluate area and speed performance. We can measure the size of designs in terms of look-up table (LUT) and register pairs. The XC2V3000 FPGA has 256 LUT rows and 112 LUT columns. The 2-sorter is 3 LUTs wide and $n$ LUTs tall when used to compare two $n$-bit numbers.

The layout produced by each of the sorters results in a solid rectangular area which is the result of tightly tiled 2-sorters. No gaps are left between columns and the routing software manages to find enough wiring resources to connect up the stages of the butterfly network. The FPGA floorplan of the sorters `sortB` and `sortQ` is shown in Fig. 8. The sorter `sortV` is like `sortB` but wider, and the sorter `sortOE` is like the sorter `sortQ` but narrower.

Note that we have to add extra delays on some of the wires of `sortOE` and `sortQ` for the pipelined version of the sorter by changing the `mid` combinator.

The results of placing and routing the four pipelined sorters are shown in Table 1. Each of these sorters sorts 32 16-bit numbers. All the sorters have a footprint which is 256 LUTs high. The width (in LUTs) of the sorters is shown in the table. The footprint of the design is 256 times the width and we use this area metric rather than the gate count.

Based on the information in the table a sorter core can use any of these sorters (or variants) to satisfy speed, area, pipelining and latency requirements. The `sortB` sorter offers the most compact pipelined sorter. If a faster pipelined

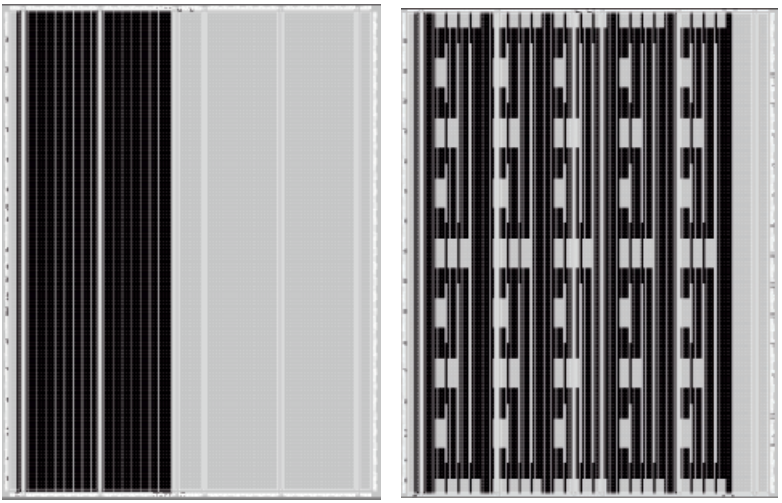


**Fig. 8.** FPGA floorplan of the sorters (a) `sortB` (b) `sortQ`

**Table 1.** Performance results

| Sorter | Speed | Latency (ticks) | Gate Count | Area of XC2V3000 | Width |
|---|---|---|---|---|---|
| `sortB` | 127 MHz | 14 | 153,350 | 43% | 45 |
| `sortOE` | 147 MHz | 14 | 136,886 | 41% | 55 |
| `sortV` | 121 MHz | 20 | 248,070 | 70% | 75 |
| `sortQ` | 152 MHz | 20 | 222,870 | 66% | 99 |

sorter is required then `sortOE` operates 20MHz faster but requires 10 extra columns. The periodic `sortV` sorter has little merit as shown since it is slow and wide. However, by using just one stage (instead of five) and feeding the output back into the input we get a very compact sorter (15 columns) but the penalty is that this sorter can not be pipelined. The fastest pipelined sorter is `sortQ` but this sorter also has the largest footprint. If a high speed non-pipelined sorter is needed then just one stage of `sortQ` can be used with the output fed back into the input which reduces the width to 18 columns. When minimising latency is an issue the sorter core will select `sortB` when saving area is more important than speed and `sortOE` when speed is more important than area.

We have shown in Lava that `prop_Sorts` is a valid safety property for all the pipelined sorters, for input sizes up to 128 bits.

## 8 Conclusion

The butterfly descriptions presented here are concise, and they result in highly optimised layouts. It is virtually impossible to produce such netlists from conventional hardware description languages. This is either because a synthesis based

approach usually has little or no support for floorplanning or if a structural approach is used then the complexity of calculating $(x, y)$ co-ordinates makes non-trivial layouts infeasible.

The butterfly-based sorters presented here have been built from a rich set of reusable combinators. This allowed us to quickly explore the design space of recursive and periodic sorters, and to make measurements of area and performance. Our earlier work on circuit description in this style indicated that large classes of circuits can be covered by surprisingly small sets of combinators.

What we have presented here is a *whole solution*, in the sense that it allows elegant circuit descriptions, formal and informal reasoning about the circuits, and a route to fast implementations on FPGAs.

The actual verifications in the development of the sorting networks which are described in this paper are the following. The actual two-sorter component (working on binary numbers) which we use on the FPGAs has been verified against a behavioural description in VHDL, for binary number sizes up to 8 bits. The various sorting networks have been verified to be correct sorting networks using the zero-one principle for sizes up to 64 inputs. Lastly, we have shown that the pipelined sorters always produce sorted outputs, for sizes up to 128 inputs.



## References

1. K.E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32, 1969.
2. E.R. Canfield and S.G. Williamson. A sequential sorting network analagous to the Batcher merge. In *Linear and Multilinear Algebra, 29*, pages 43–51, 1991.
3. Koen Claessen and Mary Sheeran. A tutorial on Lava: A hardware description and verification system. Available from `http://www.cs.chalmers.se/~koen/Lava`, 2000.
4. M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *JACM*, 36:738–757, 1989.
5. Geraint Jones and Mary Sheeran. The study of butterflies. In Graham Birtwistle, editor, *Proc. 4th Banff Workshop on Higher Order*. Springer Workshops in Computing, 1991.
6. Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.
7. D.E. Knuth. *Sorting and Searching, vol. 3 of The Art of Computer Programming.* Addison-Wesley, 1973.
8. Carl Johan Lillieroth and Satnam Singh. Formal verification of FPGA cores. *Nordic Journal of Computing*, 6:299–319, 1999.
9. Mary Sheeran. Sorts of butterflies. In Graham Birtwistle, editor, *Proc. 4th Banff Workshop on Higher Order*. Springer Workshops in Computing, 1991.
10. Mary Sheeran. Puzzling permutations. In *Proc. Glasgow Functional Programming Workshop*, 1996.