Journal of Universal Computer Science, vol. 11, no. 7 (2005), 1135-1158
submitted: 28/4/05, accepted: 9/6/05, appeared: 28/7/05 © J.UCS

Hardware Design and Functional Programming: a Perfect Match

Mary Sheeran
(Chalmers University of Technology
ms@cs.chalmers.se)

Abstract: This paper aims to explain why I am still fascinated by the use of functional languages in hardware design. I hope that some readers will be tempted to tackle some of the hard problems that I outline in the final section. In particular, I believe that programming language researchers have much to contribute to the field of hardware design.
Key Words: functional programming, hardware description languages, arithmetic and logic structures, circuit generation
Category: B.2.2, B.6.3

1 Introduction

At the beginning of the 1980s, many researchers independently came upon the idea of using a functional programming language to design and reason about hardware. Some of the references that I am familiar with are [Lahti, 1980, Sheeran, 1983, Boute, 1984, Johnson, 1984a, Johnson, 1984b, Patel et al., 1985, Morrison et al., 1985, Hunt Jr., 1985, O'Donnell, 1988, Verkest et al., 1988] but I have most likely omitted several more, for which I apologise. I thought then, and I still think that this was a great idea waiting to be discovered. And many others have discovered it along the way. In the old days, there was a feeling of excitement in the community. The interested reader is invited to dip into the long list of references that I have gathered here, to get a feeling for the nature and extent of the early work.

We felt sure that we would have an impact, and cheerfully ignored ridicule. In response to my first paper submission (which later appeared as [Sheeran, 1984]), I received a single referee's report: a very thin strip of paper which said "Functional correctness is not an issue in VLSI design. Layout is the problem.". When, as a shy doctoral student, I tried to explain my ideas to the leader of an industrial research lab, he eventually jumped red-faced out of his chair and shouted "How dare you tell my designers what to do!". And in the end, it must be said that we had little impact on industrial practice. Verilog and VHDL took over the world. In this invited paper, I would like to explain why I am more convinced than ever that functional programming and hardware design are a perfect match. Much has changed in the past 20 years. The problems facing hardware designers as we move to deep sub-micron technologies are unfathomably difficult. And functional programming languages, and our ideas about how to use them, have changed out of all recognition. My thesis is that our original idea was a good one. We were just a bit early!

1.1 μFP

My own route into this topic was via Backus' Turing Award paper [Backus, 1978]. I developed a variant of FP that operated on streams, and in which the combining forms had geometric as well as behavioural meaning. The following is the very first example that I published [Sheeran, 1984]:

```
Tal(0) = [1]
Tal(n) = α TA ∘ zip ∘  [apndl ∘ [0, Tal(n-1) ∘ mst],
                        [n,n,..n],
                        apndr ∘ [Tal(n-1) ∘ mst, 0] ]
  where  mst = reverse ∘ tl ∘ reverse
         TA  = 2 -> 3;1
```

It is the tally circuit from the classic text book [Mead and Conway, 1980], see Figure 1. It should take n inputs and produce $n+1$ outputs. If exactly j of the inputs are low (or zero), then the first (or leftmost) j outputs should also be low, followed by a single 1, and then $n-j$ low outputs (one for each high input). The component `TA` is a multiplexer (or mux) whose middle input is the address bit. It outputs its third input if that address bit is high, and its first input if it is low. The operator ∘ is function composition. `apndl` and `apndr` stand for *append left* and *append right* respectively, and they add an element to a sequence, either on the left or on the right. `tl` returns all but the first element of a non-empty sequence, and thus `mst` returns all but the last element of such a sequence.

The definition of the tally circuit is recursive. The base case is a circuit that outputs a single high output. For n inputs, the circuit constructs the two possible $n+1$-bit outputs, one beginning with low and one ending with it, and then selects between them using the n^{th} input (here called `n`) as the address bit for each of the muxes. My description contains two recursive calls of the smaller tally with $n-1$ inputs, and this now strikes me as strange. However, the combinator-oriented style did not permit the naming of intermediate values or wires. I also showed the transistor diagram of the original circuit, and with the innocence and arrogance of youth wrote "It is clear that we have described the circuit exactly.".

It should be remembered that μFP was developed before the idea of computer hardware description languages (CHDLs) had become widespread. Neither VHDL nor Verilog existed at this time, though both were about to come into existence. Many groups had long been experimenting with the design of

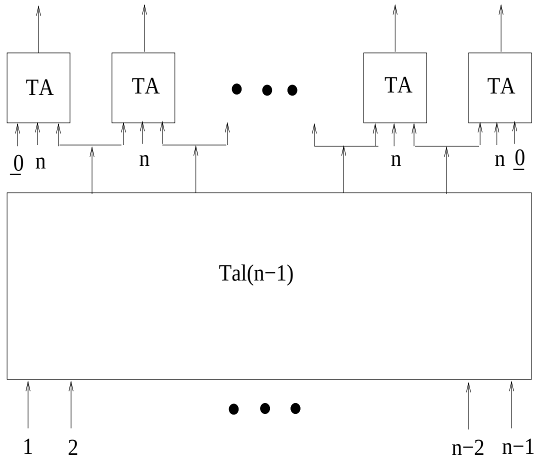


Figure 1: The Mead and Conway tally circuit, Tal(n), showing the recursive call of Tal(n-1).

CHDLs (see [Chu et al., 1992, Borrione et al., 1992] for a fascinating retrospective from 1992). Even in industry, there was a great willingness to experiment. A team at Plessey, Caswell (in the UK), investigated the use of μFP in the design of regular architectures, as part of a large Alvey Project (ARCH013). Their first experiences were encouraging, as shown by the following quotes from their paper about a case study, the design of a video picture motion estimator [Bhandal et al., 1990]:

> Using muFP, the array processing element was described in just one line of code and the complete array required four lines of muFP description. muFP enabled the effects of adding or moving data latches within the array to be assessed quickly. Since the results were in symbolic form it was clear where and when data within the results was input into the array making it simple to examine the data-flow within the array and change it as desired. This was found to be a very useful way to learn about the data dependencies within the array.
>
> [...]
>
> From the experience gained on this design, the most important consideration when designing array processors is to ensure that the processor input/output requirements can be met easily and without sacrificing array performance. The most difficult part of the design task is not the design of the computation units but the design of the data paths and associated storage devices. It is essential to have the right design tools

> to aid and improve the design process. Early use of tools to explore the flow of data within and around the array and to understand the data requirements of the array is important. muFP has been shown to be useful for this purpose.

It is interesting to note that μFP was used in the (important) design exploration stage and that the use of symbolic and multi-level simulation was much appreciated. Later, Ella [Morrison et al., 1985] and Plessey's own cell-based design system were used for the implementation. It was Geraint Jones and Wayne Luk who worked closely with the Plessey team, and our views from that time on the tools needed for regular array design are presented in reference [Luk et al., 1990]. Unfortunately, at just the wrong moment, another of the partners in the project bought the relevant part of Plessey and closed down the design team – a major setback for the research. However, my colleagues and I remained interested in the design of regular architectures, having been much stimulated by contact with real designers with special competence in this area. We moved on to experiment with a relational language that again captured common connection patterns as combinators or higher order functions [Luk and Jones, 1988, Sheeran, 1988, Jones and Sheeran, 1990]. My work at this time seemed to be of considerable interest to academics, but it generated no interest whatsoever from industry. Eventually, my response to this was to work with Satnam Singh, initially, and later Koen Claessen, on the development of a more practical design and verification system for FPGAs, Lava.

1.2 Lava: embedding a simple hardware description language in Haskell

In Lava, in which a μFP- or Lustre-like language is embedded in the functional programming language Haskell [Bjesse et al., 1998, Claessen et al., 2001], the definition of the tally is

```
tal :: [Signal Bool] -> [Signal Bool]
tal []     = [high]
tal (a:as) = bs
 where
   bs = mux(a,(low:cs,cs++[low]))
   cs = tal as
```

It takes a list of bits as input, and produces a list of bits as output. Now, there is only one call to the smaller tally circuit. Also, we now use pattern matching to give a standard-looking recursive definition over lists. The function `mux` is the built-in polymorphic multiplexer, which in this case selects between two input lists. However, all is not quite as it seems. This is not a circuit description, but a circuit *generator*. What actually happens when we want to (say) generate a net-list is that we run the circuit description with symbolic inputs (of fixed size), to produce a symbolic representation of the circuit (again of fixed size). This representation can then be processed in various ways to produce formats for input to other design and verification tools. The important point is that we have a two stage process, not a single stage as you might expect. So, in Lava, you write a Haskell program to generate a net-list, rather than writing a description of the net-list itself. This may sound like a small change of emphasis, but in fact having a full functional language available for writing generators increases expressiveness and ease of use in ways that we had not predicted when we started work on Lava.

There are pros and cons to having a data-structure that represents the circuit. One advantage is that the application of circuit transformations becomes possible. VHDL descriptions, which represent the required *behaviour* cannot easily be manipulated in the same way. Also, we perform formal reasoning only about the simple netlist-like circuit representation (at least at the moment). The semantics of such a netlist is well-understood, so it is very clear what should be proved in verification, and we will consider this in the next section. Separating the semantics of the final circuit from that of the programming language used to write the generator (which includes lazy evaluation) is an important simplification.

The rather structural approach that we advocate is, however, most appropriate in cases where we want to have full control over the properties (both functional and non-functional) of the final result. This is not always the case. Sometimes, a synthesis tool is just what is needed, and the designer is then relieved of having to worry about unnecessary detail. However, even in a synthesis tool, it is necessary to be able to supply library components that are suitable for different contexts. The production of such components is an area in which we think our approach can be applied, even in a standard design flow.

1.3 Verification in Lava

Our approach to the problem of verifying circuit correctness has been to borrow from the synchronous programming community the notion of synchronous observer [Halbwachs et al., 1993]. The observer operates on both the inputs and outputs of the circuit being observed, and produces a single Boolean output that should never become false. Both the circuit and the observer can be sequential, and the production of a logical representation of the combination of the circuit and the observer proceeds in just the same way as for an ordinary circuit. We can, for instance, produce propositional logic for input to SAT-solvers, and we have developed associated formal verification methods [Sheeran et al., 2000, Bjesse et al., 1998]. The following very small example illustrates the idea. We have designed a new full-adder circuit, and would like to check that its behaviour is the same as the built in full-adder. The observer produces a single output called `ok` and we use a SAT-solver (which is called by the function `satzoo`) to prove that it can never be false. Behind the scenes, using symbolic evalutation, a formula in DIMACS format is generated, but the user need not bother with the details of the format. We have similar links to other verification tools, including SMV [McMillan, 1999] and VIS [Brayton, R.K. et al., 1996].

```
fa::(Signal Bool,(Signal Bool,Signal Bool))->(Signal Bool,Signal Bool)
fa (cin, (a,b)) = (sum, cout)
  where
    part_sum = xor2 (a, b)
    sum      = xor2 (part_sum, cin)
    cout     = mux (part_sum, (a, cin))

checkFullAdd :: (Signal Bool,(Signal Bool,Signal Bool)) -> Signal Bool
checkFullAdd ins = ok
  where
    out1 = fa ins
    out2 = fullAdd ins    -- Lava built-in fullAdder
    ok    = out1 <==> out2

Main> satzoo checkFullAdd
Satzoo: ... (t=0.1) Valid.
```

Note that this approach only allows the verification of safety properties. A similar approach to verification can be used in standard languages, and indeed there is growing interest in standard *property specification languages*, which can express a greater variety of properties [Fitzpatrick, 2003, Accelera, 2004]. For a brief description of a course on Hardware Description and Verification that incorporates both PSL and Lava, see [Axelsson et al., 2005a].

The approach to formal verification currently used in Lava flattens the entire circuit before invoking the verification tools. It would be interesting to explore more hierarchical verification methods, which would entail a more complicated (but still tangible) circuit representation.

1.4 Connection patterns in Lava

In describing larger circuits, we make use of higher order functions that encode commom *connection patterns*. For example, here are the definitions of two of the connection patterns that are built in to Lava:

```
compose []           = id
compose (circ:circs) = circ ->- compose circs

composeN n circ = compose (replicate n circ)
```

The function `compose` composes a list of circuits in series using the series composition pattern `->-`, while `composeN` composes `n` copies of the given circuit. In the tally circuit, above, it would actually have made sense to first define a connection pattern, and then use it with the mux as parameter, rather than hard-wiring that component into the circuit.

Another typical connection pattern is `row`, which makes a linear array of two-input, two-output components. Using it, we can, for instance, make a ripple-carry binary adder from full-adders:

```
binAdder fadd as = ss ++ [c]
  where (ss,c) = row fadd (zero, as)
```

The input `as` is a list of pairs of bits from the two (least significant bit first) binary numbers being added. The carry-out from the rightmost full-adder is appended to the end of the list of sum-bit outputs to give a result whose length is one longer than that of its inputs. Note that this is a generic definition. When we want to analyse it, for example for verification, we must instantiate it to a fixed size by giving it appropriate symbolic inputs. For a case study of the use of a connection pattern oriented approach to the design and analysis of a sorter core, see reference [Claessen et al., 2001].

1.5 Non-standard interpretation for circuit analysis in Lava

We include `fadd` as a parameter in the definition of the binary adder, `binAdder`, not only to allow the use of various full-adder implementations, but also to allow the inclusion of non-standard components that assist in circuit analysis. For instance, we can define a full-adder that operates not on bits but on delays:

```
fAddI (a1s, a2s, a3s, a1c, a2c, a3c) (a1,(a2,a3)) = (s,cout)
  where
    s      =  max (a1s+a1) (max (a2s+a2) (a3s+a3))
    cout   =  max (a1c+a1) (max (a2c+a2) (a3c+a3))

fI :: (Signal Int,(Signal Int, Signal Int)) -> (Signal Int, Signal Int)
fI as = fAddI (20,20,10,10,10,10) as

Main> simulate (binAdder fI) (replicate 10 (0,0))
[20,30,40,50,60,70,80,90,100,110,100]
```

The function `fAddI` has six additional parameters which model the delay for each of the six possible paths from input to output. The component can then be used to calculate worst-case delays. This is useful in design exploration, when different circuits that use this component can be compared. This is a direct application of the idea of Non-Standard Interpretation (NSI) [Jones and Nielsen, 1994] – again an example of a direct and fruitful borrowing of a good idea from programming language research. There was some promising early work on using NSI in circuit analysis [Singh, 1992, Luk, 1990] but I feel that the idea is still ripe for exploitation. Significant advances have been made in functional languages in recent years in the development of more expressive type systems, and of new approaches to metaprogramming, reflection and resource-awareness. I cannot attempt a survey here, but the interested reader is referred to the content and citations in [Kiselyov et al., 2004, Sheard and Peyton Jones, 2002, Grundy et al., 2003, Mycroft and Sharp, 2001]. These developments will allow more type-safe embeddings or direct implementations of hardware description languages, and will allow us to do more with NSI. Later in this paper, I show that NSI can be used not only after but also *during* circuit generation.

1.6 What more do we need?

The ideas presented in the previous sections are appealing, but still they have been largely ignored by those doing circuit design in practice. My conjecture is that this is at least partly because the range of circuits that can easily be generated and analysed is too narrow. And perhaps our work has been too concentrated on the functional programming technology, and not enough on the problems faced by designers. In my recent work, I have aimed to address these issues.

The problem that I have chosen to address is the need to take account of non-functional circuit properties (such as power consumption) early in the design. The need for new ideas in this area was eloquently stated by IBM's Wolfgang Roesner in an invited talk at CHARME 2003 [Roesner, 2003]. I cannot, of course, address all of the issues raised in that talk, but I have become interested in ways to allow non-functional properties to cross abstraction boundaries. And I have started at the bottom! For instance, how might we generate a circuit whose topology is adapted to the delay on its input signals? How might we make generators that produce circuits that adapt to the context in which they are placed? Finding answers to these questions results in a broadening of the notion of *generic circuit*. Attacking these problems has, as a nice side-effect, greatly increased the range of circuits that can easily be described in a functional style.

The following sections illustrate, by means of examples, my first steps towards a design style that takes account of non-functional properties.

2 Parallel prefix circuits

A modern microprocessor contains many *parallel prefix circuits* and it was this that led us to study ways to design and analyse this class of circuits. The best known use of parallel prefix circuits is in the computation of the carries in fast binary addition circuits; another common application is in priority encoders. There are a variety of well known parallel prefix networks, including [Sklansky, 1960] and [Brent and Kung, 1982]. And there are also many papers in the field of circuit design that try to systematically figure out which topology is best for practical circuits [Knowles, 1999, Chan et al., 1992, Zlatanovici and Nikolic, 2003]. Those more familiar with functional programming than with hardware design might know the prefix function as *scan* – see, for example, O'Donnell's work on proving the correctness of a parallel scan algorithm [O'Donnell, 1994]. More

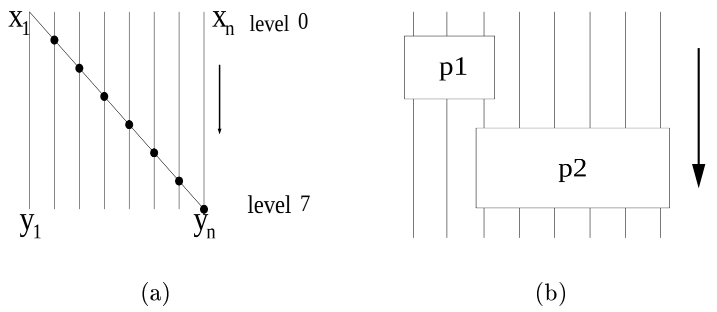
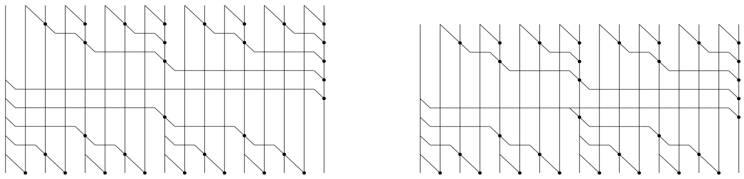


Figure 2: (a) The serial prefix structure (b) Construction of a parallel prefix circuit by composition of two smaller such circuits, p1 and p2.

recently, Hinze has studied the algebra of scans in a delightful paper that is also a great tutorial on parallel prefix networks [Hinze, 2004].

Given n inputs, $x_1, x_2, \ldots, x_n$, the problem is to design a circuit that takes these inputs and produces the n outputs $y_1 = x_1$, $y_2 = x_1 \circ x_2$, $y_3 = x_1 \circ x_2 \circ x_3$, $\ldots$ $y_n = x_1 \circ \ldots \circ x_n$, where $\circ$ is an arbitrary associative (but not necessarily commutative) binary operator. One possible solution is the *serial prefix circuit* shown schematically in Figure 2(a). Input nodes are on the top of the circuit, with the least significant input (x_1) being on the left. Data flows from top to bottom, and we also count the stages or levels of the circuit in this direction, starting with level zero on the top. An operation node, represented by a small circle, performs the $\circ$ operations on its two inputs. One of the inputs comes along the diagonal line above and to the left of the node, and the other along the vertical line feeding the node from the top. A node always produces an output to the bottom along the vertical line. It may also produce an output along a diagonal line below and to the right of the node. Here, at level zero, there is a diagonal line leaving a vertical wire in the absence of a node. This is a fork. The diagram shows a ripple-carry structure. The circuit shown contains 7 nodes, and so is said to be of *size* 7. Its lowest level in the picture is level 7, so the circuit has *depth* 7. The fan-out of a node is its out-degree. In this example, all but the rightmost node have fan-out 2, so the whole circuit is said to have fan-out 2. Examining the serial prefix structure in Figure 2(a), we see that at each non-zero level only one of the vertical lines contains a node. We are interested in exploring ways to design and analyse *parallel* prefix circuits, in which there can be more than one node at a given level, so that there is some parallelism in the resulting computation.

For two inputs, there is only one reasonable way to construct a prefix circuit,



(a) (b)

Figure 3: (a) A parallel prefix construction using a forwards and a backwards tree (b) The same construction with the lower tree slid back by one level, which reduces the depth by one.

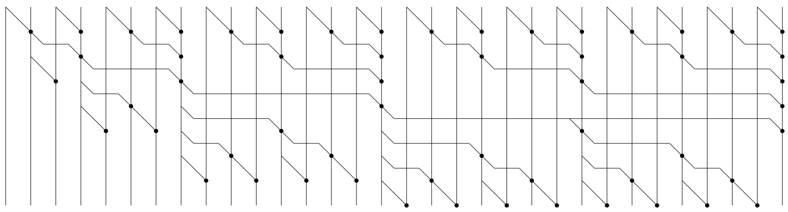


Figure 4: The Brent-Kung construction for 32 inputs.

using one copy of the operator. Parallel prefix circuits can also be formed by composing two smaller such circuits, as shown in Figure 2(b). Repeated application of this pattern (and the base case) produces the ripple-carry circuit that we have already seen.

For $2^n + 1$ inputs, one can use so-called forwards and backwards trees, as shown in Figure 3(a). We call a parallel prefix circuit of this form a *slice*. At the expense of a little extra fan-out at a single level in the middle of the circuit, one can slide the (lower) backwards tree up one level, as shown in Figure 3(b). Composing increasing sized slices gives the well-known Brent-Kung construction [Brent and Kung, 1982] shown in Figure 4.

A shallower n-input parallel prefix circuit can be obtained by using the recursive Sklansky construction, which combines the results of two separate $n/2$-input parallel prefix circuits [Sklansky, 1960], as shown in Figure 5. It is written in Lava as

```
sklansky :: ([a] -> [a]) -> [a] -> [a]
sklansky op [a]    = [a]
sklansky op as     = (sklansky op >|> join op (sklansky op)) as
   where join op p (a:as) = withEach op (a: p as)
```

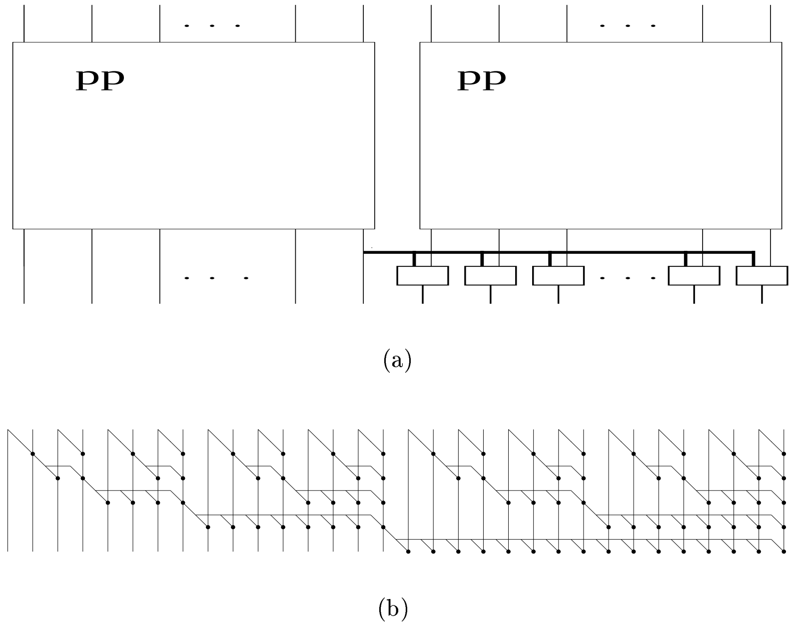


Figure 5: (a) The recursive Sklansky construction (b) Sklansky for 32 inputs, with fan-out 17.

Here, in the second equation, the Sklansky circuit is constructed by composing two smaller parallel prefix circuits. The connection pattern called `>|>` composes two parallel prefix circuits as shown in Figure 2(b), but in addition makes sure that the two sub-circuits each operate on half of the inputs.

`join op p` takes a parallel prefix circuit `p` with n inputs, and makes a parallel prefix circuit with $n+1$ inputs, by combining the first input with each of the outputs of `p`, see Figure 6. The first of the two smaller circuits composed using `>|>` is simply a recursive call of the Sklansky construction, while the second is `join op` applied to such a sub-circuit. The depth of this construction is $\lceil \log_2 n \rceil$ and this is the shallowest possible. The down-side of the Sklansky construction is that it has high fan-out. The Kogge-Stone construction, which has fan-out of only 2, also achieves minimum logical depth, but at the expense of a large number of operators and long wires [Kogge and Stone, 1973].

2.1 Generating depth-size optimal parallel prefix circuits

An n-input prefix circuit of depth d and size s is said to be *depth-size optimal* if $d + s = 2n - 2$. Of those parallel prefix constructions discussed above, only the

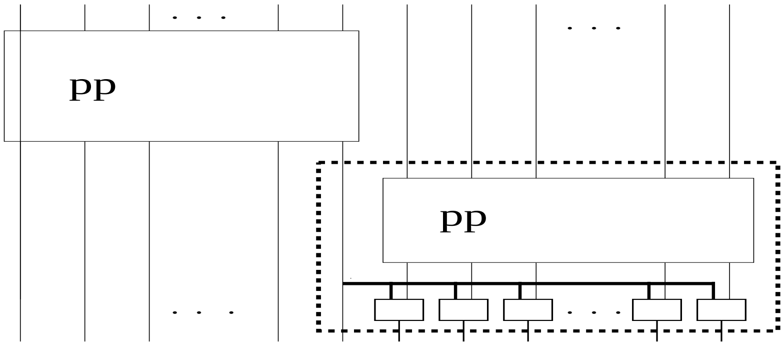


Figure 6: The decomposition of the Sklansky construction corresponding to the Lava code.

serial prefix is depth-size optimal. We have investigated the generation of depth-size optimal circuits as a means to find a good compromise between number of operators, depth, wire-length and fan-out. This entails a study of the algebra of the prefix (or scan) combinator, in true functional programming tradition.

Consider the slice shown in Figure 3(a). The first input line is forked for the first time at level 4, and we say that it has *firstFork* 4. Its last output is calculated at level 5, even though the circuit has depth 9. We call the level at which the last output is computed *lastd*. We call the difference between *lastd* and *firstFork* the *waist* of the circuit. A parallel prefix circuit with waist w is *waist-size optimal* if $w + s = 2n - 2$. (This property is called size optimality by its originators, who also present it a little differently [Lin et al., 2003]). Both of the 17-input slices shown in Figure 3 have waist one and size 31; so both are waist-size optimal. The composition (as in Figure 2(b)) of two waist-size optimal circuits gives a new waist-size optimal circuit whose waist is the sum of the waists of its components. We note also that a waist-size optimal circuit whose waist is the same as its depth is, in addition, depth-size optimal. This gives a clue about how to build a depth-size optimal circuit: we should compose a series of waist-size optimal circuits in such a way that the resulting circuit's waist is the same as its depth. So, to build a depth n circuit, we must compose n waist-one slices whose waists span from level i to level $i+1$, for i ranging from 0 to $n-1$. Each slice should have as many inputs as possible, and the depths of the forwards and backwards trees must be such that the total circuit depth does not exceed n.

If we restrict fan-out to 2, as in the slice shown in Figure 3(a), then the result is like an extended version of Brent-Kung. It is made of 9 composed slices, with the size of the slices increasing and then decreasing, see Figure 7. This circuit

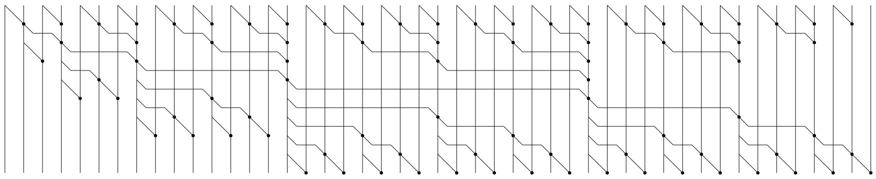


Figure 7: Composing waist-one slices to make a depth-size optimal parallel prefix circuit with fan-out 2. It has 47 inputs, depth 9 and 83 operators.

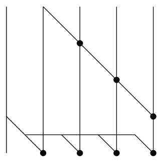


Figure 8: A waist-size optimal circuit (of waist one) that combines serial prefix with matching fan-out.

has a pleasing symmetry, and the reader might like to consider the effect of rotating it through 180 degrees and then sliding all the dots to the other end of their diagonal lines. (If that was too easy, consider the effect of performing this operation on *any* parallel prefix circuit.)

Things become more interesting, though, when we allow fan-out to increase slightly. Allowing the arbitrary use of fan-out gives circuits that contain too many operators to be waist-size optimal. However, using a serial prefix construction in the forward tree, matched by fan-out in the backwards tree, as shown in Figure 8, gives a waist-size optimal circuit. Moreover, this construction can be used at any level in the trees, meaning that the serial prefix construction can here combine the results of 4 sub-trees, while there will be fan-out to 4 matching sub-trees in the lower tree. Armed with this knowledge, we can write a function that constructs a slice with a forward tree of depth m and matching backwards tree of depth n, for a given fan-out, f. The fact that we can use fan-out greater than 2 in the backwards tree allows us to make larger trees for a given depth. The slice should take in as many inputs as possible for the required tree depths.

```
slice m n f op d [a] = [a]
slice m n f op d (a:as) = cs ++ [bsl']
  where
    bs  = ftree m n f op d as
    bsm = most bs
    bsl = last bs
    [a',bsl'] = op [a,bsl]
    cs  = btree m n f op d (a':bsm)
```

Depth	Slices(4)	WE4	Z4	H4	M	LYD
8	47-72	47-64	45-67	42-57	45-62	55-77
9	73-114	65-102	68-98	58-87	63-87	78-95
10	115-179	103-138	99-145	88-119	88-122	96-135
12	282-440	215-286	209-303	180-243	166-233	170-242
14	692-1082	439-582	431-621	364-491	311-443	309-446

Table 1: The number of inputs that can be processed by some depth-size optimal circuits for a specific depth

The recursive definitions of the forwards and backwards trees are slightly tricky, but well suited to the functional style of description.

Now, we can make a depth-size optimal parallel prefix circuit with depth `m` and fan-out `f` by composing `slice i (m-i) f`, for `i` ranging from `0` to `m-1`. For depth 8 and fan-out 4, the result is shown in Figure 9. For want of a better name, we call the new construction $Slices(f)$, where f is the fan-out. Here, we have shown how to construct the circuit with the largest possible number of inputs for a given depth and fan-out. To make circuits of the same depth, but with fewer inputs, one can remove some lines and their associated operators. This can be done in a way that preserves depth-size optimality.

The previous best known depth-size optimal construction for fan-out 4 and depth 8 was $Z4$, which could take 67 inputs [Lin and Hsiao, 2004]. Table 1 (which is drawn from [Lin and Hsiao, 2004], with the addition of figures for our new circuit) compares the new construction with the best previously known depth-size optimal circuits. *Z4*, *WE4* and *H4* have fan-out 4, while *M* and *LYD* have unrestricted fan-out per level. Note that we have improved considerably on all known depth-size optimal constructions, even those with unrestricted fan-out.

I would also assert that the Slices construction is very much simpler than its depth-size optimal competitors! A word of caution, though: these are interesting theoretical results, but to check whether or not they are useful in practice, we must *build* circuits using them, and measure their performance, including power consumption. To this end, three groups of VLSI design students, supervised by Prof. Per Larsson-Edefors at Chalmers, are designing fast adders based on the Slices(4) construction (which is used for computing the carries). The students will implement reference adders using the construction given in [Han and Carslon, 1987], which is a hybrid of Brent-Kung and Kogge-Stone. One of the groups will have their project implemented on silicon, and measurements will be conducted in the autumn of 2005. We await the results with interest.

It is also interesting to play with varying the fan-out. For depth 8, fan-out

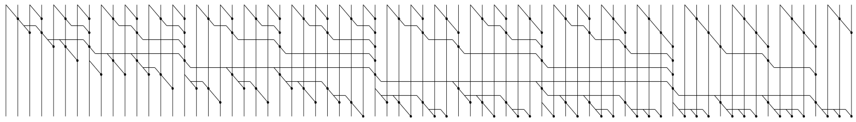


Figure 9: A new arrangement of composed slices, with 72 inputs, depth 8 and fan-out 4. We call this construction Slices(4).

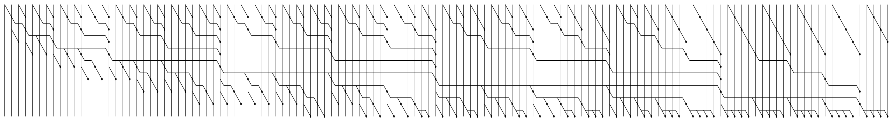


Figure 10: A depth 9, fan-out 5, 128 input parallel prefix circuit.

of 5 gets us to 80 inputs, while fan-out of 9 (the maximum the construction can use) reaches 88 inputs. For depth 9, increasing fan-out to 5 gets us to 128 inputs, which might well be interesting in practice. The resulting circuit is shown in Figure 10. It has (as expected) 245 operators. By comparison, the Sklansky construction, for 128 inputs, has 448 operators (with depth 7 and fan-out 65).

Incidentally, the dot-diagrams of slices and parallel prefix circuits shown in this paper have been automatically produced from Lava descriptions. A non-standard interpretation is first used to gather information about the circuit network and a small Haskell program then produces Xfig input. This is typical of the kind of lightweight analysis that is easy to do in a functional language.

3 Clever circuits

The parallel prefix constructions that we have just considered have a degree of regularity, although that regularity is not always immediately obvious when one considers the diagrams. In recent years, I have come to realise that it is important to be able to describe and analyse not only regular circuits but also circuits that are *almost regular*. The idea was born in a discussion with a designer of high performance arithmetic circuits. He explained how what starts off as a regular adder circuit becomes less and less regular as one goes down through the levels of abstraction, taking account of more and more non-functional properties that depend on the exact context in which each cell is placed. But still, the final circuit has some relationship to its more regular ancestor, even if we don't currently have good intellectual tools for expressing that relationship. I became fascinated by the general question of how to generate circuits that adapt to their contexts, losing some but not all of their regularity in the process. This led to a new programming idiom, which I call *clever circuits*.

We have seen in earlier examples how one uses pattern matching in recursive functions over lists to define parameterised circuits (see the definition of `sklansky` for instance). Another idiom used Haskell integers to control the unrolling that is used to generate circuit net-lists (as in the definition of `composeN`). In *clever circuits*, Haskell-level *shadow values* are paired with circuit-level inputs and outputs, and used to control circuit generation. The shadow-values can contain whatever information we choose. In a first application, I encoded, on the shadow wires, the degree of sortedness of the values on the circuit wires, and used this information to generate median circuits from descriptions of sorters [Sheeran, 2003]. The shadow values controlled the presence or absence of comparator components. In a development of that idea, I used shadow values to make choices about circuit *wiring* during the generation of multiplier reduction trees [Sheeran, 2004].

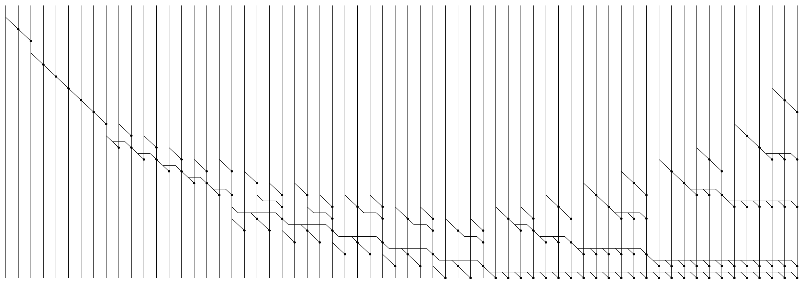
4 Combining clever circuits and NSI to get adaptive circuits

Here, I shall use clever circuits to determine circuit *topology* based on information about non-functional properties (in this case information about the delay on circuit inputs). For example, Figure 11 shows a 64-input parallel prefix circuit that is well adapted to its input delay profile, which is typical of that from a reduction tree. The example is taken from work at Synopsys on optimised synthesis of sum-of-products circuits [Zimmermann and Tran, 2003]. This example uses a series of narrow slices on the left, and a Sklansky-like construction on the right.

To facilitate such descriptions, it makes sense to build combinators that work on *sets* of circuits (or in fact of circuit generators). For example, the function `best` chooses among a set of possible circuits according to an ordering on the output shadow values.

```
best comp [f] i    = f i
best comp (a:as) i = if (comp fs gs) then a i else best comp as i
          where (_,fs) = unzip (a i)
                (_,gs) = unzip (best comp as i)
```

Then, one way to build a composite parallel prefix circuit is to define a set of solutions for the left hand side of the circuit, and then choose the one that has smallest *lastd*. Similarly, one can minimise the difference between depth and *firstFork* from a set of possible solutions for the right hand side. Finally, one can choose where to place the division between these two parts to minimise the overall measure, which is the `comp` parameter.



64 lines, 23 stages, 26 max fanout, horizontal wirelength is 79, 137 operators.

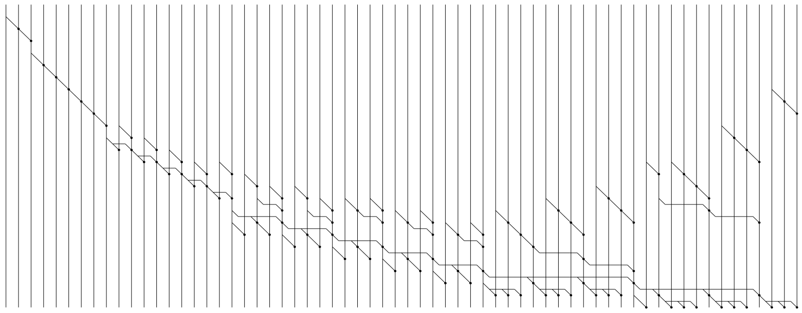
Figure 11: A parallel prefix circuit adapted to a typical delay profile on the output of a reduction tree.

```
adapt m n f comp op d as
  = best comp [composeK k (left k) (right k) | k <- [t-m..t+m]] as
 where t =  div (length as) 2
           left tr  = bestf lastdP (lpart f m n op d ....
           right tl = bestf dminfirstfoP (rpart f m n op d ....
```

Figure 11 shows what happens when the set corresponding to `rpart` contains only the Sklansky construction. We can, as we did in the non-adaptive case, get a low-fan-out solution by using slices. `rpart` should then express the set of possible combinations of slices, and the context, that is the delay on the input shadow values, determines which is actually chosen. Figure 12 shows the result for our typical delay profile, while in Figure 13 different choices have been made, in response to a shallower profile.

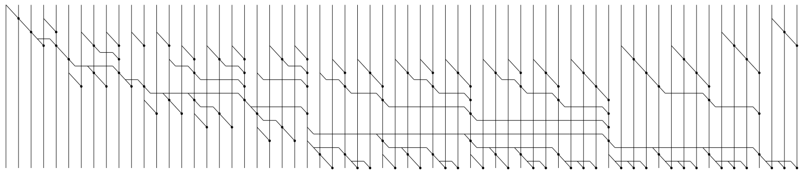
It would be easy to use a finer degree of modelling in the calculation of the shadow values that control the generation. Here, the modelling was crude, and counted only the number of operator delays. It is just a question of including the required modelling functions in the non-standard versions of the operators. Similarly, it would be possible to include the modelling of wires, as we did in multiplier reduction tree generation [Sheeran, 2004], or to have more than one kind of dot operator, which would be useful in the generation of adders.

Note that the user is now expressing *sets* of possible circuits and how to choose between them. This is just a first attempt at defining suitable combinators, and it is surprisingly effective. However, we feel that we can do better. Koen Claessen has pointed out a possible link to pretty-printing combinators. It should be possible to adapt Hughes' ideas about the development of libraries of pretty-printing combinators [Hughes, 1995] to the problems of generating circuits that meet constraints on their non-functional properties. This is yet another opportunity for cross-fertilisation between functional programming and hardware design. Thinking along these lines will most likely lead to a much broader notion



64 lines, 25 stages, 4 max fanout, horizontal wirelength is 69, 105 operators.

Figure 12: A low-fan-out adaptive parallel prefix construction.



64 lines, 12 stages, 4 max fanout, horizontal wirelength is 120, 115 operators.

Figure 13: A parallel prefix circuit adapted to a shallower delay profile.

of shadow value. Sometimes, it will make sense to pair the shadow values with wires, as we have done here, but in other cases the shadow values will be linked to the circuit in more general ways. Many interesting questions remain to be investigated.

5 Discussion and Conclusion

In section 2.1, I demonstrated the power of the connection pattern approach to the writing of circuit generators. A study of the algebra of parallel prefix suggested a new design, and the functional language made it easy to realise and analyse that design. The compactness of the descriptions, and the immediate feedback from the generated pictures, enabled a great deal of design exploration along the way to the new design. I am convinced that this exploration would have been very difficult, if not impossible, in a more standard hardware description language such as VHDL.

Next, I showed how the combination of non-standard interpretation and clever circuits gives a method of writing generators for circuits that adapt to their contexts. Although I, again, demonstrated the approach using the example of parallel prefix, the method is a general one. The better non-standard models we can create, the better circuits we will generate. This seems to be a promising approach to the problem of creating families of circuits with different non-functional properties to act as library components implementing a given function.

So far, I have been considering a rather low level of abstraction in circuit design. And what my colleagues and I are doing is going even lower! We are developing *Wired*, a system for circuit design that takes account of the effects of wires by being precise about their sizes and locations [Axelsson et al., 2005b]. The descriptions are essentially relational, which means that ideas from my earlier work with Geraint Jones on Ruby are reappearing [Jones and Sheeran, 1990]. The use of relations allows not only the kind of unidirectional or functional non-standard interpretation that has been shown here, but also bidirectional analyses. An example of such an analysis is RC-delay estimation, in which resistance values flow in one direction, and capacitance values in the other. Our aim is to make this lower level of design and analysis quicker and easier, and our first results are promising. Here, again, we feel that we will be able to incorporate ideas that have been proposed for use in pretty-printing [Hughes, 1995]. We are particularly interested in ways to generate circuits with low switching activity, and we hope to develop new methods of generating low-power circuits.

While developments in low level circuit design are important, we will not solve the problems facing designers unless we find new design and verification methods that work earlier in the design process, at higher levels of abstraction. An important question is "Can the methods described in this paper be used at higher levels of abstraction?". My tentative answer is that these are general programming idioms, and so may have application at other levels of abstraction. Even at the higer levels, we are still faced with the pressing problem of how to get non-functional information across abstraction levels in ways that assist the designer, rather than overwhelming her. The use of shadow values is one idea, but I am convinced that there are many more waiting to be discovered. Programming language researchers have much to contribute here.

Another big question is "Can we make refinement work?". Early work in refinment for hardware design resulted in some nice examples, but had little impact [Jones and Sheeran, 1990, O'Donnell and Ruenger, 2004]. Now, Carl Seger and his colleagues at Intel have boldly built the IDV system – Integrating Design with Verification [Spirakis, 2004]. It allows the refinement of a (relatively) high level model all the way to implementation on the silicon, with every transformation step being guaranteed to preserve the original specification. The message is that the current design flow is broken and that we must replace it by a new flow that puts the designer in charge of guaranteeing both correctness and non-functional properties such as timing and power consumption, and gives him the necessary analysis tools, and well-designed ways of using them. The latest version of IDV is built around the reflective functional language ReFLect [Grundy et al., 2003].

For an academic who has envisaged but never built such a system, IDV is both exciting and stimulating. It raises many questions. As it starts with a relatively low level specification, what should be built on top of it?. How can we assist micro-architects in making early decisions based on estimates of non-functional properties? Can the kind of microarchitectural algebra that was studied in Hawk [Matthews and Launchbury, 1999] be made to work on real designs? Can we make use of Vuillemin's elegant work linking circuits and numbers [Vuillemin, 1994] in the verification of arithmetic circuits? Can we cope with Globally Asynchronous Locally Synchronous systems, in order to get a handle on buses, networks on chip, caches and all the problems that arise above the hardware level? Can we find good ways to do data-path generation from Electronic System Level Design languages like SystemC [OSCI, 2005]? What new verification technology is needed if we are to raise the level of abstraction at which design decisions are made? The current version of IDV works on fixed size instantiations of circuits. The million dollar question is "Can we design appropriate *generic* design and verification methods?". Hunt's work on design and verification using the DUAL-EVAL language [Brock and Hunt Jr., 1997] and his recent work that builds upon it are good starting points. But much remains to be done. How, for instance, can we make better use of hierarchical verification? Insights from functional programming as well as from automated and interactive theorem proving, will doubtless play a role in developing practical usable methods and tools. Now, we need to reason about the generators themselves, rather than about the fixed-size tangible representations of the generated circuits. My hope is that the fact that the generators are structured in the same way as the circuits themselves will aid the construction of the necessary inductive proofs.

Mead and Conway developed simplified design methods aimed at enabling a new kind of "tall thin person" to get all the way from architecture to circuit layout [Mead and Conway, 1980]. The aim was to "deal with complexity through deliberate methodological simplifications" [Conway, 2003]. It seems to me that it is time to return to these ideas. And to succeed, I think we will need another kind of tall thin person, who can span the range from low level hardware design to high level functional programming!

Acknowledgements

Thanks to Emil Axelsson and Satnam Singh for their stimulating comments on an earlier draft of this paper. This work is funded, in part, by the Swedish

Research Council, Vetenskapsrådet. Many thanks to Ricardo Massa and Martin Musicante, who gave me the opportunity to speak at the Brazilian Symposium on Programming Languages, 2005, and to write this invited paper.

References

[Accelera, 2004] Accelera (2004). *PSL v1.1 Language Reference Manual*.

[Axelsson et al., 2005a] Axelsson, E., Björk, M., and Sheeran, M. (2005a). Teaching Hardware Description and Verification. In *International Conference on Microelectronic Systems Education, MSE*. IEEE.

[Axelsson et al., 2005b] Axelsson, E., Claessen, K., and Sheeran, M. (2005b). Wired: wire-aware circuit design. In *Correct Hardware Design and Verification Methods, CHARME, to appear*, Lecture Notes in Computer Science. Springer.

[Backus, 1978] Backus, J. (1978). Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):165–180.

[Bhandal et al., 1990] Bhandal, A. S., Considine, V., and Dixon, G. E. (1990). An array processor for video picture motion estimation. In *Systolic array processors*, pages 369–378. Prentice-Hall, Inc.

[Bjesse et al., 1998] Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. (1998). Lava: Hardware Design in Haskell. In *International Conference on Functional Programming, ICFP*, pages 174–184. ACM.

[Borrione et al., 1992] Borrione, D., Piloty, R., Hill, D., Lieberherr, K., and Moorby, P. (1992). Three Decades of HDLs, part 2: Conlan through Verilog. *Design & Test of Computers*, 9(3):54 – 63.

[Boute, 1984] Boute, R. (1984). Functional languages and their application to the description of digital systems. *Journal A*, 25(1):27–33.

[Brayton, R.K. et al., 1996] Brayton, R.K., Hachtel,G.D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S. -T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R.K., Sarwary, S., Shiple, T.R., Swamy, G., and Villa, G. (1996). VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer.

[Brent and Kung, 1982] Brent, R. and Kung, H. (1982). A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31:260–264.

[Brock and Hunt Jr., 1997] Brock, B. and Hunt Jr., W. (1997). The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor. *Formal Methods in System Design*, 11(1):71–104.

[Chan et al., 1992] Chan, P., Schlag, M., Thomborson, C., and Oklobdzija, V. (1992). Delay Optimization of Carry-Skip Adders and Block Carry-Lookahead Adders using Multi-dimensional Dynamic Programming. *IEEE Transactions on Computers*, 41(8):920–930.

[Chu et al., 1992] Chu, Y., Dietmeyer, D., Duley, J., Hill, F., Barbacci, M., Rose, C., Ordy, G., Johnson, B., and Roberts, M. (1992). Three Decades of HDLs, part 1: CDL through TI-HDL. *Design & Test of Computers*, 9(2):69 – 81.

[Claessen et al., 2001] Claessen, K., Sheeran, M., and Singh, S. (2001). The Design and Verification of a Sorter Core. In *Correct Hardware Design and Verification Methods, CHARME*, volume 2144 of *Lecture Notes in Computer Science*, pages 355–369. Springer.

[Conway, 2003] Conway, L. (2003). Lynn Conway's Retrospective. Part III: Starting Over. http://ai.eecs.umich.edu/people/conway/Retrospective3.html.

[Fitzpatrick, 2003] Fitzpatrick, T. (2003). *SystemVerilog Assertions: A Unified Language for More Efficient Verification*. Synopsys, Inc.

[Grundy et al., 2003] Grundy, J., Melham, T., and O'Leary, J. (2003). A reflective functional language for hardware design and theorem proving. Research Report PRG-RR-03-16, Programming Research Group, Oxford University Computing Laboratory. http://www.comlab.ox.ac.uk/tom.melham/pub/Grundy-2003-RFL.pdf.

[Halbwachs et al., 1993] Halbwachs, N., Lagnier, F., and Raymond, P. (1993). Synchronous observers and the verification of reactive systems. In *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Workshops in Computing, pages 83–96. Springer.

[Han and Carslon, 1987] Han, T. and Carslon, D. (1987). Fast Area-Efficient VLSI Adders. In *Int. Symposium on Computer Arithmetic*. IEEE.

[Hinze, 2004] Hinze, R. (2004). An algebra of scans. In *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 186–210. Springer.

[Hughes, 1995] Hughes, J. (1995). The Design of a Pretty-printing Library. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 53–96. Springer.

[Hunt Jr., 1985] Hunt Jr., W. (1985). *FM8501, A Verified Microprocessor*. PhD thesis, University of Texas at Austin.

[Johnson, 1984a] Johnson, S. (1984a). Applicative programming and digital design. In *Principles of Programming Langauges*, pages 218–227. ACM.

[Johnson, 1984b] Johnson, S. (1984b). *Synthesis of Digital Design from Recursive Equations*. MIT Press.

[Jones and Sheeran, 1990] Jones, G. and Sheeran, M. (1990). Circuit design in Ruby. In Staunstrup, J., editor, *Formal Methods for VLSI Design*, pages 13–70. North-Holland.

[Jones and Nielsen, 1994] Jones, N. and Nielsen, F. (1994). Abstract interpretation: A semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, pages 527–636. OUP.

[Kiselyov et al., 2004] Kiselyov, O., Swadi, K., and Taha, W. (2004). A methodology for generating verified combinatorial circuits. In *Fourth International Conference on Embedded Software (EMSOFT)*, pages 249–258. ACM Press.

[Knowles, 1999] Knowles, S. (1999). A Family of Adders. *Int. Symp. on Computer Arithmetic*, pages 277–284.

[Kogge and Stone, 1973] Kogge, P. and Stone, H. (1973). A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–793.

[Lahti, 1980] Lahti, D. (1980). Applications of a Functional Programming Language to Hardware Synthesis. Master's thesis, UCLA.

[Lin and Hsiao, 2004] Lin, Y.-C. and Hsiao, J.-W. (2004). A new approach to constructing optimal parallel prefix circuits with small depth. *Journal of Parallel and Distributed Computing*, 64(1):97–107.

[Lin et al., 2003] Lin, Y.-C., Hsu, Y.-H., and Liu, C.-K. (2003). Constructing H4, a Fast Depth-Size Optimal Parallel Prefix Circuit. *The Journal of Supercomputing*, 24(3):279–304.

[Luk, 1990] Luk, W. (1990). Analysing parametrised designs by non-standard interpretation. In *International Conference on Application-Specific Array Processors*, pages 133–144. IEEE Computer Society Press.

[Luk and Jones, 1988] Luk, W. and Jones, G. (1988). From specification to parametrised architectures. In Milne, G., editor, *The fusion of hardware design and verification*. North-Holland.

[Luk et al., 1990] Luk, W., Jones, G., and Sheeran, M. (1990). Computer-based tools for regular array design. In *Systolic array processors*, pages 589–598. Prentice-Hall, Inc.

[Matthews and Launchbury, 1999] Matthews, J. and Launchbury, J. (1999). Elementary Microarchitecture Algebra. In *International Conference on Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 288–300. Springer.

[McMillan, 1999] McMillan, K. (1999). *A methodology for hardware verification using compositional model checking*. Cadence.

[Mead and Conway, 1980] Mead, C. and Conway, L. (1980). *Introduction to VLSI Systems*. Addison-Wesley.

[Morrison et al., 1985] Morrison, J., Peeling, N., and Thorp, T. (1985). The Design Rationale of ELLA, a Hardware Design and Description Language. In *IFIP 7th International Symposium on Computer Hardware Description Languages and their Applications*, pages 303–320. North-Holland.

[Mycroft and Sharp, 2001] Mycroft, A. and Sharp, R. (2001). Hardware synthesis using safl and application to processor design, invited talk. In *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME*, volume 2144 of *Lecture Notes in Computer Science*, pages 13–39. Springer.

[O'Donnell, 1994] O'Donnell, J. (1994). A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338.

[O'Donnell and Ruenger, 2004] O'Donnell, J. and Ruenger, G. (2004). Derivation of a Logarithmic Time Carry Lookahead Addition Circuit. *Journal of Functional Programming*, 14(6):697–713.

[O'Donnell, 1988] O'Donnell, J. T. (1988). Hydra: Hardware description in a Functional Language using Recursion Equations and High Order Combining Forms. In Milne, G., editor, *The fusion of hardware design and verification*. North-Holland.

[OSCI, 2005] OSCI (2005). *SystemC 2.1 Language Reference Manual*. Open SystemC Initiative, www.systemc.org.

[Patel et al., 1985] Patel, D., Schlag, M., and Ercegovac, M. (1985). νfp: An environment for the multi-level specification, analysis, and synthesis of hardware algorithms. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 238–255. Springer.

[Roesner, 2003] Roesner, W. (2003). What is beyond the RTL Horizon for Microprocessor and System Design?, invited talk. In *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, page 1. Springer.

[Sheard and Peyton Jones, 2002] Sheard, T. and Peyton Jones, S. (2002). Template metaprogramming for Haskell. In Chakravarty, M. M. T., editor, *ACM SIGPLAN Haskell Workshop*, pages 1–16. ACM Press.

[Sheeran, 1983] Sheeran, M. (1983). μFP, an algebraic VLSI design language. D. Phil. Thesis, Programming Research Group, Oxford University.

[Sheeran, 1984] Sheeran, M. (1984). muFP, A Language for VLSI Design. In *LISP and Functional Programming*, pages 104–112. ACM.

[Sheeran, 1988] Sheeran, M. (1988). Retiming and Slowdown in Ruby. In Milne, G., editor, *The fusion of hardware design and verification*. North-Holland.

[Sheeran, 2003] Sheeran, M. (2003). Finding regularity: describing and analysing circuits that are almost regular. In *Correct Hardware Design and Verification Methods, CHARME*, volume 2860 of *Lecture Notes in Computer Science*, pages 4–18. Springer.

[Sheeran, 2004] Sheeran, M. (2004). Generating fast multipliers using clever circuits. In *Formal Methods in Computer-Aided Design, FMCAD*, volume 3312 of *Lecture Notes in Computer Science*, pages 6–20. Springer.

[Sheeran et al., 2000] Sheeran, M., Singh, S., and Stålmarck, G. (2000). Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design, FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer.

[Singh, 1992] Singh, S. (1992). Circuit Analysis by Non-Standard Interpretation. In

Designing Correct Circuits, volume A-5 of *IFIP Transactions*, pages 119–138. North-Holland.

[Sklansky, 1960] Sklansky, J. (1960). Conditional-sum addition logic. *IRE Trans. Electron. Comput.*, EC-9:226–231.

[Spirakis, 2004] Spirakis, G. (2004). Opportunities and challenges in building silicon products at 65nm and beyond, invited talk. In *Design and Test in Europe (DATE)*, pages 2–3. IEEE.

[Verkest et al., 1988] Verkest, D., Johannes, P., Claessen, L., and De Man, H. (1988). Formal Techniques for Proving Correctness of Parameterized Hardware using Correctness Preserving Transformations. In Milne, G., editor, *The fusion of hardware design and verification*. North-Holland.

[Vuillemin, 1994] Vuillemin, J. (1994). On circuits and numbers. *IEEE Transactions on Computers*, 43:8:868–879.

[Zimmermann and Tran, 2003] Zimmermann, R. and Tran, D. (2003). Optimized synthesis of sum-of-products. In *Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, pages 867–872. IEEE.

[Zlatanovici and Nikolic, 2003] Zlatanovici, R. and Nikolic, B. (2003). Power-Performance Optimal 64-bit Carry-Lookahead Adders. In *29th European Solid-State Circuits Conference, ESSCIRC*, pages 321–324.