Wired: Wire-Aware Circuit Design

Emil Axelsson, Koen Claessen, and Mary Sheeran

Chalmers University of Technology
{emax, koen, ms}@cs.chalmers.se

Abstract. Routing wires are dominant performance stoppers in deep sub-micron technologies, and there is an urgent need to take them into account already at higher levels of abstraction. However, the normal design flow gives the designer only limited control over the details of the lower levels, risking the quality of the final result. We propose a language, called Wired, which lets the designer express circuit function together with layout, in order to get more precise control over the result. The complexity of larger designs is managed by using parameterised connection patterns. The resulting circuit descriptions are compact, and yet capture detailed layout, including the size and positions of wires. We are able to analyse non-functional properties of these descriptions, by "running" them using non-standard versions of the wire and gate primitives. The language is relational, which means that we can build forwards, backwards and bi-directional analyses. Here, we show the description and analysis of various parallel prefix circuits, including a novel structure with small depth and low fanout.

1 Introduction

In deep sub-micron processes, the effects of wires dominate circuit behaviour and performance. We are investigating an approach to circuit generation in which wires are treated as first class citizens, just as components are. To successfully design high-performance circuits, we must reach convergence not only on functionality, but also *simultaneously* on other properties such as timing, area, power consumption and manufacturability. This demands that we mix what have earlier been separate concerns, and that we find ways to allow non-functional properties to influence design earlier in the flow. We must broaden our notion of correctness to include not only functionality but also performance in a broad sense. For example, we might like to do high-level floor-planning that takes account of the effects of the wires joining the top-level blocks, or to quickly explore the detailed timing behaviour of a number of proposed architectures for a given arithmetic block, without having to resort to full-custom layout. The Wired system is designed to solve both of these problems, though our initial work has concentrated on the latter: easing the design and analysis of data-paths.

Ever since the eighties, there has been much work on *module generation*. For example, Becker et al explored the specification and generation of circuits based on a calculus of nets [1]. As in μFP [8], the design notation took into account

D. Borrione and W. Paul (Eds.): CHARME 2005, LNCS 3725, pp. 5–19, 2005.

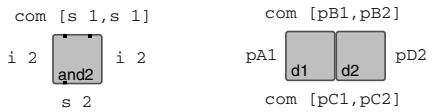
geometric and topological information. However, the designer viewed wires as "simple lines", and did not consider their exact position (although the associated synthesis tool produced real layout using sophisticated algorithms). The Wired user works at a lower level of abstraction and is in full control of the layout, including the exact positions of wires. Our own work with Singh at Xilinx on the use of Lava to give the designer fine control over the resources on the FPGA indicated that for regular circuits such as data-paths, mixing structure and behaviour in a single description gives good results [4]. Wired takes these ideas a step further. It is primarily aimed at giving the designer full control in standard-cell design. In both Lava and μFP, circuit behaviour is described as a *function* from input to output, and combinators capture common connection patterns. This use of functions can lead to a proliferation of connection patterns [8], whereas a relational approach, such as that in Ruby [6], abstracts from direction of data-flow and so avoids this problem. In Wired, the connection patterns provide a simple tiling, and the resulting behaviour is constructed by composing the relations corresponding to the tiles or sub-circuits. Thus, ideas from Ruby are reappearing. A major difference, though, is that Wired is embedded in Haskell, a powerful functional programming language, which eases the writing of circuit generators. As we shall see when we consider RC-delay estimation, the relational approach lends itself to circuit analysis by non-standard interpretation.

Typically, we start in Lava, and step down to the Wired level when it becomes necessary to consider effects that are captured only at that level. We have found that programming idioms used in Lava (that is the net-list generator level) translate surprisingly well into the lower Wired level. You might say that we aim to make circuit description and design exploration at the detailed wire-aware level as easy as it was at the higher net-list generator level – without losing the link to functional verification. In a standard flow, an application might be in the generation of modules that adapt to their context (for example to the delay profile of the inputs). The ideas are also compatible with recent work at Intel on the IDV system (Integrating Design and Verification [12]), which gives the designer full control in a setting based on refinement and a functional language. Our aim is to develop a practical approach to power-aware design in such a setting.

2 The Wired System

2.1 The Core Language

Wired is built around *combinators* with both functional and geometrical interpretations. Every description has a *surface* and a *surface relation*. A surface is a structure of contact segments, where each contact may or may not carry a signal. This structure specifies the interface of the description and keeps track of different signal properties. When flattened, it can represent the description's geometrical appearance. Figure 1(a) shows an example of a simple two-input and-gate. This is a 2-dimensional circuit, so the surface consists of four *ports*. The left- and right-hand ports are i-contacts (contacts without signals) of size 2. The inputs, on top, are two size 1 s-contacts (contacts with signals). The output,



(a) **(b)**

Fig. 1. (a) Two-input and-gate (b) Beside composition, `d1*||*d2`

on the bottom, is a size 2 output signal. This gate has a clear signal flow from input to output, which is not always the case for Wired circuits.

The surface relation relates parts of the surface to each other. It can capture both structural and functional constraints. A structural relation can, for example, specify that the number of inputs is equal to the number of outputs, and a functional relation could specify that two signals are electrically connected.

Wired is embedded in the functional programming language Haskell. The data type that is used to represent descriptions internally is defined as:

```
data Description =   Primitive Surface Relation
                   | Combined  Combinator Description Description
                   | Generic   Surface (Surface -> Maybe Description)
```

A description is either a *primitive* defined by a surface and a relation, or a *combination* of two sub-descriptions. We will look more at *generic* descriptions in section 2.2. The combinator describes how the two sub-surfaces are combined into one, and indicates which surface parts are connected where the two blocks meet. This implicitly defines a new surface and relation for the combined description. Figure 2 illustrates a combination of two (not necessarily primitive) 2-dimensional circuits with relations R_1 and R_2.

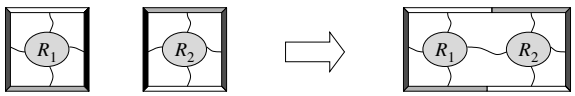


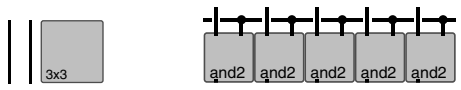
Fig. 2. Combination of sub-descriptions

The combinator `*||*` (*"beside"*) places the first block to the left of the second, while `*=*` (*"below"*) places the first block below the second. Figure 1(b) illustrates `d1*||*d2`. Note how the resulting top and bottom ports are constructed. The top ports of the sub-circuits are named `pB1` and `pB2`, and the resulting top port becomes the pair `com [pB1,pB2]`. The same holds for the side ports when using `*=*`. We will also use variations of these, which have the same geometrical meaning, but combine the surfaces differently. `*||~` does the "cons" operation; if `d1` has port `pB1` and `d2` has port `com [pB21,pB22, ...]`, then the resulting port becomes `com [pB1,pB21,pB22, ...]`. `~||*` does "cons" at the end of the list, and `~||~` and `-||-` are two variations of the "append" operation.

The surface structure may be partially unknown. For example, a wire (wires are normal descriptions, just like anything else) may not have a fixed length, but can be instantiated to whatever length it needs to have. Such instantiation is done – automatically by the system – by looking at the surrounding context of each sub-description. The surrounding surfaces form a so-called *context surface*, and we require, for all sub-descriptions, that the surface and the context surface are structurally equal. This means that if, for example, a stretchy wire is placed next to a block with known geometry, the wire will automatically be instantiated to the appropriate length. The wire also has a relation that states that its two sides have the same length. So, if we place yet another wire next to the first one, size information will be propagated from the block, through the first one and over to the new wire. In Wired, this circuit is written:

```
example1 = wireY *||* wireY *||* block3x3
```

`wireY` is a thin vertical wire with unknown length, and `block3x3` is a pre-defined block of size 3×3 units. Instantiating this description and asking for a picture (through an interactive menu system) gives the layout in Figure 3(a).



(a) **(b)**

Fig. 3. (a) Layout after instantiation of `example1` (b) Layout of 5-bit bit-multiplier

In Lava, circuits are constructed by just running their Haskell descriptions, so most of the instantiation procedure comes for free, from Haskell. Since Wired is relational, we cannot use the same trick here. Instead we have a separate *instantiation engine*, which is implemented in Haskell. This engine works by fix-point iteration – it traverses the description iteratively, propagating surface information from contexts to sub-descriptions and instantiating unknown primitive surfaces, until no more information can be gained.

2.2 Generic Descriptions and Connection Patterns

In `example1` we saw wires that adapted to the size of their context. This is very useful since the designer doesn't have to give all wire sizes explicitly when writing the code. Sometimes we want to have sub-blocks whose entire content adapts to the context. For this we use the final constructor in the definition of descriptions (section 2.1) – `Generic`. A generic description is defined by a surface and an *instantiation function*. As the type (`Surface -> Maybe Description`) indicates, this function reads its current surface and may choose, depending on that information, to instantiate to a completely new description. Since this is a normal function on the Haskell level, it is possible to make clever choices here.

For example, in the context of non-functional analysis (section 3), we can choose between two different implementations depending on some estimated value.

The most common use for generic descriptions is in defining *connection patterns* which capture commonly used regular structures. The simplest connection pattern is the *row*, which places a number of copies of the same component next to each other. We can use it to define a bit multiplier circuit, for example:

```
bitMult = row and_bitM
  where and_bitM = and2 *=* (cro *||* crT0)
```

The primitives used are: `and2`, an and-gate with the surface from figure 1(a), `cro`, two wires crossing without connection and `crT0`, a T-shaped wire connection. Figure 3(b) shows the layout of this circuit instantiated for 5 bits.

We define `row` as follows:

```
row d = generic "row" xpSurf (row_inst d)

row_inst d surf = do len <- lengthX surf
                     case len of N 0 -> newInst thinEmptyY
                                 N _ -> newInst (d *||~ row d)
                                 _   -> noInst
```

The pattern is parameterised by a description `d`, and has unknown initial surface (`xpSurf`) and instantiation function `row_inst` (also parameterised by `d`). The instantiation function looks at the current surface and does a case analysis on its horizontal length. If the length is known to be 0 (the constructor `N` means known), the row becomes a thin empty block. This is the base-case in the recursive definition of `row`. For lengths greater than 0, we place one copy of `d` beside another row, using the `*||~` combinator. If the length of the context has not yet been resolved (the last case), we do not instantiate.

A simpler alternative to the above definition of `row` is

```
rowN 0 _ = thinEmptyY
rowN n d = d *||~ rowN (n-1) d
```

This definition takes an extra length parameter `n`, and does the whole unrolling on the Haskell level instead, before instantiation. This is both simpler and runs faster, but has the down-side that the length has to be known in advance. In the normal `row`, instantiation automatically resolves this length.

Generic descriptions or partially unknown surfaces are only present during circuit instantiation. After instantiation, when we want to view the layout or extract a net-list, we require all surfaces to be complete, and that all generic parts have been instantiated away.

2.3 Signal Interpretation

Surfaces are structures of contact segments. A contact is a piece of area that may or may not carry a signal. However, it is possible to have more information here. The signal can, for example, be decorated with information about whether it is

an input or an output, and about its estimated delay. This allows an analysis that checks that two outputs are never connected, and a way to compute the circuit's delay from input to output. We want to separate the description from the interpretation that we will later give to it, so that the same description can be used with different interpretations. This is done by abstracting the signal information on the Haskell type level. That is, we parameterise the `Description` type by the signal type `s`. This type variable can be kept abstract as long as we want, but before we instantiate the description, `s` must be given a particular type.

At the moment, possible signal types are:

`NoInfo`	No information, just a structural placeholder
`Direction`	Signal direction (in/out)
`UnitTime`	Delay estimation under unit delay model
`Resistance`	Output driving resistance
`Capacitance`	Input load capacitance
`Time`	Accurate RC-delay estimation

The operator `:+:` combines signal types. Such combinations are needed since it makes no sense to talk about delays if there is no notion of direction, for example. To increase readability, we define some useful type macros. For example,

```
type Desc_RCDelay = Description (Direction :+: Resistance :+: Capacitance :+: Time)
```

3 Non-functional Analysis

3.1 Direction and Unit-Delay

Wired is a relational language, and is thereby not bound to any specific direction of signal flow. Still, most circuits that we describe are functional, so we need to be able to check that a description has a consistent flow from input to output. Here we use the signal interpretation with direction. While it is usually known for gates which signals are inputs and outputs, wire junctions normally have undefined directions initially. However, we know that if one signal in a junction is an input (seen from inside the junction), all the others must be outputs, in order to avoid the risk of short-circuit. This constraint propagation can be baked into the circuit relation, and this is enough to help the instantiation engine resolve all directions (or report error). Figure 4 shows an example of a gate cell and a wire junction. Signal $s_{j,1}$ of the junction is indirectly connected to gate output $s_{g,5}$. If we assume that directions are propagated correctly through the intermediate wires, the context will eventually constrain $s_{j,1}$ to be an input, and by direction propagation, $s_{j,2}$ and $s_{j,3}$ will be constrained to outputs.

The simplest model for circuit delay estimation is the *unit-delay* model, in which each stage just adds a constant unit to the input delay – independent of electrical properties, such as signal drive strength and load. This gives a rather rough estimate of circuit delay.

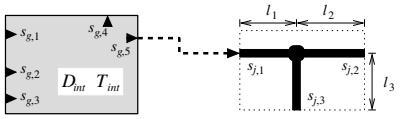


Fig. 4. Gate and wire junction

As with directions, unit-delay can be resolved by the instantiation engine, provided that delays are propagated correctly through gates and wires. The gate in the above example has intrinsic unit-delay D_{int} (and an accurate time delay T_{int}, which will be used in the next section). D_k refers to the unit-delay of signal s_k. As instantiation proceeds, delay estimates of the input signals will become available. Delay propagation can then set the constraints

$$D_{g,4} = D_{g,5} = max[D_{g,1}, D_{g,2}, D_{g,3}] + D_{int}$$

The model can easily be extended so that different input-output paths have different delays.

For the wire junction, we want to capture the fact that longer wires have more delay. This dependency is hidden in the function *conv*, which converts distance to unit-delay. By choosing different definitions for *conv*, we can adjust the importance of wire delays compared to gate delays. Once the delay of $s_{j,1}$ becomes available, the following propagation can be performed:

$$D_{j,k} = D_{j,1} + conv(l_1 + l_k) \quad \text{for } k \in [2, 3]$$

These two propagation methods work for all wires and gates, independent of number of signals and logical function, and they are part of the relations of all wire and gate primitives. Since information is only propagated from inputs to outputs, this is a *forwards analysis*. In the next section, we will use a combination of forwards and backwards analysis.

3.2 RC-Delay

For a more accurate timing analysis, we use the model in Figure 5. A gate output is a voltage source with intrinsic delay T_{int} and output resistance R_o. A wire is a distributed RC-stage with r and c as resistance and capacitance per length unit respectively. Gate input is a single capacitance C_g.

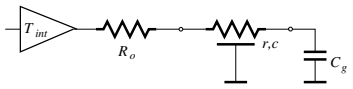


Fig. 5. Circuit stage from output to input

A signal change on an output gives rise to a signal *slope* on connected inputs. This slope is characterised by a *time constant*, τ. For output stages with equal

rise and fall times (which is normally the case), it is standard to define wire delay as the time from an output change until the input slope reaches 50% of its final value. For a simple RC-stage, see Figure 6(a), the time constant is given by $\tau = RC$. The delay from the left terminal to the capacitor is then approximately equal to 0.69τ. For a distributed RC-stage (Figure 6(b)) with total resistance $R = rL$ and capacitance $C = cL$, it can be shown that $\tau_{dist} \approx RC/2$.

Figure 6(c) shows a fanout composition of n RC-stages. Based on Elmore's formula [7], the delay from the left terminal to capacitor C_i can be approximated by a simple RC-stage with the time constant

$$\tau_{1,i} = R_1 \cdot \left[\sum_{l \in [1..n] \setminus i} C_l \right] + (R_1 + R_i)C_i = \tau_1 + R_1 \cdot \left[\sum_{l \in [2..n]} C_l \right] + \tau_i \qquad (1)$$

This formula also holds for distributed stages – R_i and C_i are then the total resistance and capacitance of stage i – or for combinations of simple and distributed stages. Note that the local time constants τ_1 and τ_i are computed separately and added, much as unit-delays of different stages were added. What is different here is the extra fanout term, where R_1 is multiplied by the whole load capacitance. It is generally the case that the stages on the right-hand side are themselves compound; the RC-stage is merely an approximation of their timing behaviour. Therefore, load capacitance needs to be propagated backwards from the load, through the stages and to the junction we are currently considering. So, for RC-delay analysis, we need a combination of forwards and backwards analysis. This is, however, a simple matter in a relational system like Wired.

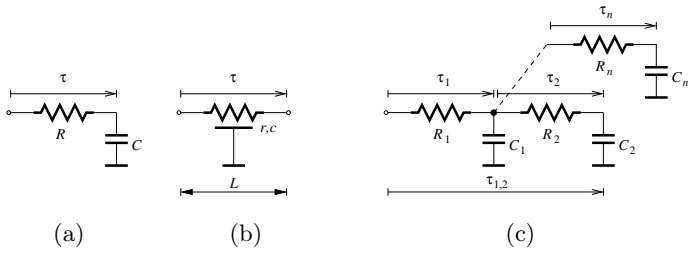


Fig. 6. (a) RC- and (b) distributed RC-stage (c) Composition of RC-stages

We describe gate and wire propagation from the example in Figure 4. Gates always have known output resistances and input capacitances, so no such propagation is needed. Therefore RC-delay propagation through gates behaves just like for unit-delay. Propagation through wire junctions is more tricky. We use R_k, C_k and τ_k to refer to the resistance, capacitance and RC-delay of the signal s_k. We also define R'_k, C'_k and τ'_k as the local resistance, capacitance and time constant of the piece of wire going from s_k to the junction. R'_k and C'_k can be computed directly from the corresponding length l_k, and $\tau'_k = R'_k C'_k/2$. The drive resistance and time constant of $s_{j,1}$, and the load capacitance of $s_{j,2}$ and $s_{j,3}$ will be resolved from the context.

The total load capacitance at the junction is given by $C_{junc} = C_{j,2} + C_{j,3} + C'_{j,2} + C'_{j,3}$. Backwards capacitance propagation can then be done directly by the constraint

$$C_{j,1} = C_{junc} + C'_1$$

From (1) we get the time constant at the junction as

$$\tau_{junc} = \tau_{j,1} + \frac{R_{j,1}C'_{j,1}}{2} + \tau'_{j,1} + (R_{j,1} + R'_{j,1}) \cdot C_{junc}$$

Finally, forwards resistance and RC-delay propagation is done as

$$R_{j,k} = R'_{j,k} \quad \text{and} \quad \tau_{j,k} = \tau_{junc} + \tau'_{j,k} \quad \text{for } k \in [2,3]$$

Now, to perform an RC-analysis of a circuit, say `bitMult` from section 2.2, we first select the appropriate signal interpretation:

```
bitMultRC = bitMult :: Desc_RCDelay
```

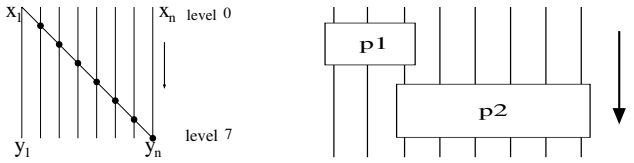
This description is then instantiated in a context surface that specifies resistance, capacitance and delay on the inputs or outputs of the circuit.

4 Parallel Prefix Circuits

A modern microprocessor contains many *parallel prefix 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 [11] and Brent-Kung [2]. There are also many papers in the field of circuit design that try to systematically figure out which topology is best for practical circuits. We have been inspired by previous work on investigating the effect of wire delay on the performance of parallel prefix circuits [5]. Our aim has been to perform a similar analysis, not by writing a specialised simulator, but by describing the circuits in Wired and using the instantiation engine to run RC-delay estimations.

4.1 The Parallel Prefix Problem

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 7(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 from the top. A node always produces an output to the bottom



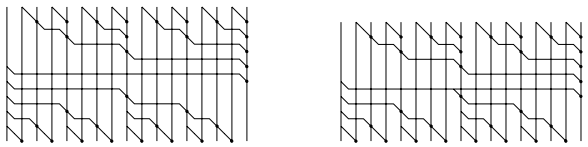
(a) Serial prefix **(b)** Composition

Fig. 7.

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 *serial prefix* 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 fanout of a node is its out-degree. In this example, all but the rightmost node have fanout 2, so the whole circuit is said to have fanout 2. Examining Figure 7(a), we see that at each non-zero level only one of the vertical lines contains a node. We aim to design *parallel* prefix circuits, in which there can be more than one node per level.

For two inputs, there is only one reasonable way to construct a prefix circuit, using one copy of the operator. Parallel prefix circuits can also be formed by composing two smaller such circuits, as shown in Figure 7(b). Repeated application of this pattern (and the base case) produces the serial prefix circuit.

For $2^n + 1$ inputs, one can use so-called forwards and backwards trees, as shown in Figure 8(a). We call a parallel prefix circuit of this form a *slice*. At the expense of a little extra fanout at a single level in the middle of the circuit, one can slide the (lower) backwards tree up one level, as shown in Figure 8(b). Composing increasing sized slices gives the well-known Brent-Kung construction [2] shown in Figure 9(a). A shallower n-input parallel prefix circuit can be ob-

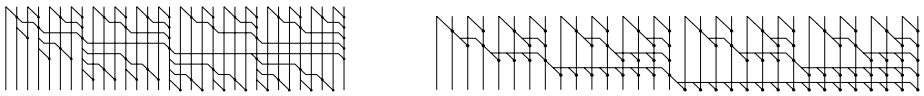


(a) **(b)**

Fig. 8. (a) Parallel prefix construction using a forwards and a backwards tree (b) The same construction with the lower tree slid back by one level

tained by using the recursive Sklansky construction, which combines the results of two separate $n/2$-input parallel prefix circuits [11], as shown in Figure 9(b).

We have studied new ways to combine slices like those used to build Brent-Kung. By allowing the constrained use of fanout greater than 2, we have found a way to make slices in which the forward and backwards trees have different



(a) **(b)**

Fig. 9. (a) Brent Kung for 32 inputs (b) Sklansky for 32 inputs, with fanout 17

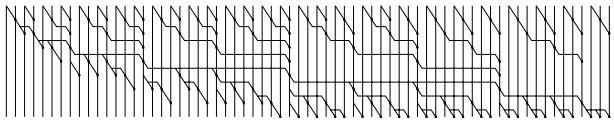


Fig. 10. A new arrangement of composed slices, with 67 inputs, depth 8 and fanout 4

depths, and this leads to parallel prefix structures that combine small depth with low fanout. An example of such a structure is shown in Figure 10.

4.2 Wired Descriptions

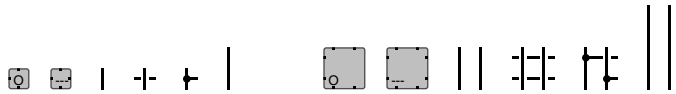
All of our parallel prefix circuits can be built from the primitives in Figure 11(a). `d` is the operator with inputs on the left and top ports, and output on the bottom. Its companion `d2` additionally feeds the left input over to the output on the right-hand side. Although `d2` is a primitive, it behaves as if there were a horizontal wire crossing on top of it. `w1`, `w2` and `w3` are unit-size wire cells, and `w4` is a wire with adaptive length. Instead of hard-coding these into the descriptions, we will supply them as parameters, which allows us to use the same pattern with a different set of parameter blocks.

Just like for the row in section 2.2, we can choose between unrolling the structure during instantiation, or in advance on the Haskell level. Here we choose the latter, since it gives us more readable descriptions.

As shown in figure Figure 9(b), Sklansky consists of two smaller recursive calls, and something on the bottom to join their results. This leads to the following description in Wired:

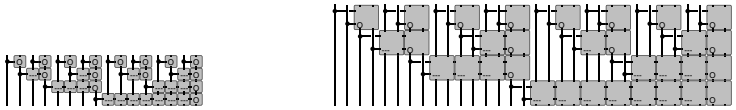
```
sklansky 0   = thinEmptyX1
sklansky dep = join *=~ (sklansky (dep-1) ~||~ sklansky (dep-1))
  where
    join = (row w1 ~||* w3) -||- (row d2 ~||* d)
      where (d,d2,w1,_,w3,_) = params
```

The parameter blocks are taken from the global variable `params`. The local parameter `dep` determines the depth of the circuit. For each recursive call, this parameter is decreased by one. Figure 12 shows this structure instantiated for 16 inputs, both for bits and for pairs of bits. The distinction between the two cases is simply made by choosing parameter blocks from Figure 11(a) or (b).



(a) (b)

Fig. 11. Parameters `d`, `d2`, `w1`, `w2`, `w3` and `w4` for (a) 1-bit, and (b) 2-bit computations



(a) (b)

Fig. 12. 16-bit Sklansky: (a) For single bits (b) For pairs of bits

Brent-Kung consists of a sequence of the slices from Figure 8(b). Each slice contains a forwards and a backwards tree. The forwards tree is:

```
fwdTree 1   = thinEmptyX1
fwdTree dep = join *=~ (fwdTree (dep-1) ~||~ fwdTree (dep-1))
  where
    join = (row w1 ~||* w3) -||- (row w2 ~||* d)
      where (d,d2,w1,w2,w3,_) = params
```

Note the similarity to the definition of `sklansky`. Only a parameter to the second occurrence of `row` has changed.

`backTree` shares the same structure, but with extra control to take care of the slide-back (Figure 8(b)). Brent-Kung is then defined as

```
(d,_,w1,_,w3,w4) = params

bk True  1   = colN 1 $ w3 *||* d
bk _     1   = rowN 2 w4 ~=~ ((w1 *||* w3) *=* (w3 *||* d))
bk first dep = wFill ~=~ bk False (dep-1)
               ~||~ (backTree first True dep ~=~ fwdTree dep)
  where
    wFill = if depth==2 then thinEmptyX
                        else row w4 ~=~ (row w4 ~||* (w3 *=* w3))
```

The recursive case places a smaller Brent-Kung next to a slice consisting of a forwards and backwards tree. The rest of the code is a bit messy due to the fact that slide-back destroys the regularity around the base case.

The new structure in Figure 10 is also based on slices, and can be described in Wired without introducing any new ideas. Its Wired layout is shown in Figure 13.

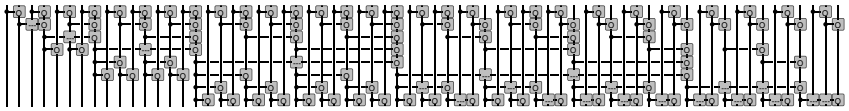


Fig. 13. Wired layout of the new design

4.3 Results

The parameters used in our estimations are (see also Figure 5):

T_{int}	R_o	C_g	r	c
50.1ps	23.4kΩ	0.072fF	0.0229Ω/λ	1.43aF/λ

We analyse for a 100nm process (λ = 50nm, half the technology feature size), following a similar analysis by Huang and Ercegovac [5]. This is not a normal process node, but rather a speculative process derived from NTRS'97 [10]. The gate parameters refer to a min. size gate, but in the analyses, the output resistance is that of a 5× min. size device, while input capacitance is 1.5× min. size. Wiring parameters r and c are obtained by assuming a wire width of 4λ (see formula(4) in [5]). The operator cells are square, with a side length of 160λ.

The delays in nanoseconds resulting from the analsysis for Sklansky, Brent-Kung and the new structure for 64 bits are (starting from the least significant output):

Sklansky	0.010, 0.058, 0.10, 0.11, ... 0.42, 0.42, 0.42, 0.42
Brent-Kung	0.015, 0.066, 0.11, 0.12, ... 0.51, 0.51, 0.55, 0.36
New	0.012, 0.062, 0.11, 0.11, ... 0.40, 0.44, 0.44, 0.40

The result for Sklansky is very closely in line with those in [5], and the results for the new structure are promising.

5 Discussion

We have shown how the Wired system allows the user to describe and analyse circuits at a level of detail that captures not only the size and position of wires and components but also the overall behaviour. The descriptions are parameterised on the building blocks (such as gates, wires and junctions) used in the construction, and these blocks can also model non-functional aspects of circuit behaviour. This permits detailed circuit analyses such as the RC-delay estimation shown here. The instantiation engine that is used to propagate size information through the circuit based on information both from the circuit and its context is also used to perform these analyses. Circuit behaviour, whether functional or non-functional, is captured as a relation that is the composition of the relations for the sub-circuits. Thus, the analyses can be relational in nature (as the RC-delay estimation is), involving the flow of data in more than

one direction, and multiple iterations before a fixed point is reached. Compared to a purely functional approach such as that used in Lava, this gives a strict increase in the range of analyses that can be performed by non-standard interpretation. Once the non-functional behaviour of the lowest level building blocks has been modelled, it becomes easy to explore the properties of a variety of different circuits to implement the same function. The fact that the descriptions are compact and quick to write is important here, and this has been demonstrated in our exploration of both standard and new parallel prefix circuits.

So far, we have only seen descriptions with 2-dimensional surfaces. We call these *hard* circuits, since they have a strict geometrical interpretation. Wired also supports hard *3-dimensional* and *soft* circuits. Hard 3D descriptions are used to make realistic circuits in processes with several metal layers, but if we only need to see and analyse a simplified layout, we prefer to use hard 2D descriptions, as they are much simpler. Soft descriptions do not have the geometrical constraints that hard descriptions have. They are used to fill in parts of the geometry that we don't want to describe exactly. It is possible to convert soft descriptions to hard, and vice versa. These operations only replace the outer surface of the description and keep the internal contents unchanged. Using these ideas about soft and hard circuits, we hope to return to the problem of high-level floor-planning that takes account of the effects of wires between blocks.

Currently, the only available output format is the kind of postscript picture shown here. The next step is to produce layout in a format such as GDSII. We will also, in the style of Lava, perform net-list extraction and produce input for verification tools. This will allow quick sanity checks during design exploration.

Work on Wired will continue to be driven by case studies. Building on our work on the generation of reduction trees [9] and on parallel prefix circuits, we plan to build, using Wired, a fast binary multiplier circuit. This will involve the study of buffer generation and of methods of folding circuits to improve layout.

The resulting circuit descriptions are *highly parameterised*, raising the question of how to provide *generic* verification methods. This is an important question, and we have no easy answers. We feel, however, that Hunt's work on the DUAL-EVAL language [3] and his current work that builds upon it is the best available starting point. We hope that others in the community will be inspired to tackle the problem of how to verify highly parameterised circuit generators.

Acknowledgements

This work receives Intel-custom funding from the SRC. We acknowledge an equipment grant from Intel Corporation. Thanks to Ingo Sander for comments on an earlier draft.

References

1. B. Becker, G. Hotz, R. Kolla, P. Molitor, and H.-G. Osthof. Hierarchical design based on a calculus of nets. In *DAC '87: Proceedings of the 24th ACM/IEEE conference on Design automation*, pages 649–653. ACM Press, 1987.
2. R.P. Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31, 1982.
3. B. Brock and W.A. Hunt Jr. 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), 1997.
4. K. Claessen, M. Sheeran, and S. Singh. The Design and Verification of a Sorter Core. In *CHARME*, volume 2144 of *LNCS*. Springer, 2001.
5. Z. Huang and M. Ercegovac. Effect of Wire Delay on the Design of Prefix Adders in Deep-Submicron Technology. In *Proc. 34th Asilomar Conf.* IEEE, 2000.
6. G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.
7. J.M. Rabaey et al. *Digital Integrated Circuits*. Prentice Hall, 2003.
8. M. Sheeran. *μFP, an algebraic VLSI design language, D.Phil. Thesis*. Oxford University, 1983.
9. M. Sheeran. Generating fast multipliers using clever circuits. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 3312 of *LNCS*. Springer, 2004.
10. SIA. *National Technology Roadmap for Semiconductors*. 1997.
11. J. Sklansky. Conditional-sum addition logic. *IRE Trans. Electron. Comput.*, EC-9, 1960.
12. G. Spirakis. Opportunities and challenges in building silicon products at 65nm and beyond. In *Design and Test in Europe (DATE)*. IEEE, 2004.