Generating fast multipliers using clever circuits

Mary Sheeran

Chalmers University of Technology
ms@cs.chalmers.se

Abstract. New insights into the general structure of partial product reduction trees are combined with the notion of clever circuits to give a novel method of writing simple but flexible and highly parameterised data-path generators.

1 Introduction

In this work, our original intention was to describe and analyse a number of different multipliers, and to account for the effects of choices about layout and wire length. We concentrated on the central component of most multipliers, the reduction array that converts the partial products into a pair of binary numbers to be added by a final fast adder. To our surprise, we found a very general way to describe a large class of reduction arrays. They all have the form of the triangular array of cells shown in Fig. 3, and varying just two small wiring patterns inside the cells allows us to cover a range of slow and fast multipliers. This insight into the structure of reduction trees in general led us to the idea of building an adaptive reduction array, in which those two small wiring cells are (repeatedly) instantiated to appropriate wiring patterns *during circuit generation*, based on information about delay on the inputs gained by the use of models of the half and full adder cells and of the wires connecting them. This is a neat application of the idea of *clever circuits* [12]. The resulting reduction tree seems to have rather good performance, at least according to our abstract analyses. Much work will need to be done to confirm that the adaptive array is indeed superior to standard arrays. We will need to experiment with different approaches to the layout of the array. Further, we were able to use clever wiring cells also to take account of restrictions in the availability of tracks for cross-cell wiring.

We became increasingly interested in methods of writing simple but powerful circuit *generators* that look exactly like structured circuit *descriptions* but that produce circuits that are adapted to a given context and so are not as regular as might first appear. We believe that the move to deep sub-micron necessitates new design methods in which lower level details are exposed early in the design, while, at the same time, there is a move to design at higher levels of abstraction and with a greater degree of reuse. Thus, methods of getting low level information up through levels of abstraction will become increasingly important. The method of writing generators for adaptive circuits presented here is an approach to this problem. It is presented via a running example, including the actual code of the generators. Readers who are not familiar with Haskell or Lava are referred to the Lava tutorial [4]

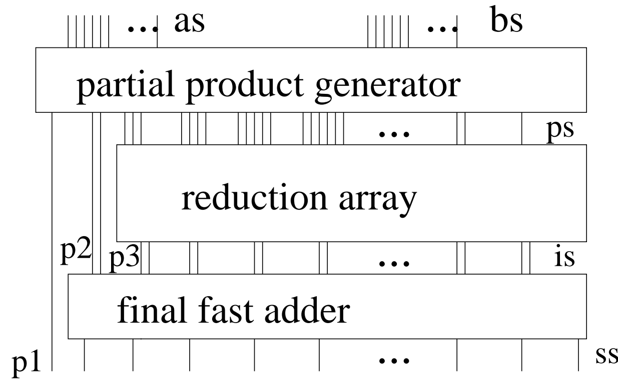

Fig. 1. The structure of a multiplier, as defined in `multBin`

2 A general multiplier

A binary multiplier consists of a partial product generator, a reduction array that reduces the partial products to two bits for each bit-weight, and a final adder that produces the binary result (see Fig. 1). Thus, transcribing from the picture, the top level of a Lava description of a multiplier is

```
multBin comps (as,bs) = p1:ss
  where
      ([p1]:[p2,p3]:ps) = prods_by_weight (as,bs)
      is                = redArray comps  ps
      ss                = binaryAdder     ([p2,p3]:is)
```

Binary numbers are represented by lists, least significant bit first. (In Haskell, `[]` is the empty list, `[a,b,c]` is a list of length 3, and `(a:as)` is a list whose first element is `a` and the remainder of which is the list `as`.) The three equations correspond to the three components of the multiplier, and indicate what their inputs and outputs are. For example, the partial product generator has as inputs the two binary numbers to be multiplied, least significant bit first, and produces a list of lists of bits. The first of these is of weight one and is the singleton list `[p1]`. The second is of weight two and contains two bits: `[p2,p3]`. The remainder, `ps`, is a list of lists of increasing weight, and is input to the reduction tree. This description works only for 3 by 3 bit multiplication and above. The `comps` parameter to both the multiplier contains a tuple of the building blocks, such as full- and half-adders, used to construct the multiplier. We postpone the decision as to what exactly it should contain.

In this paper, we concentrate entirely on the design of the reduction array. It is implemented as a linear array (a `row`) of `compress` cells, each of which reduces the partial products at its bit-weight to two.

```
redArray comps ps = is
  where (is,[]) = row (compress comps) ([],ps)
```

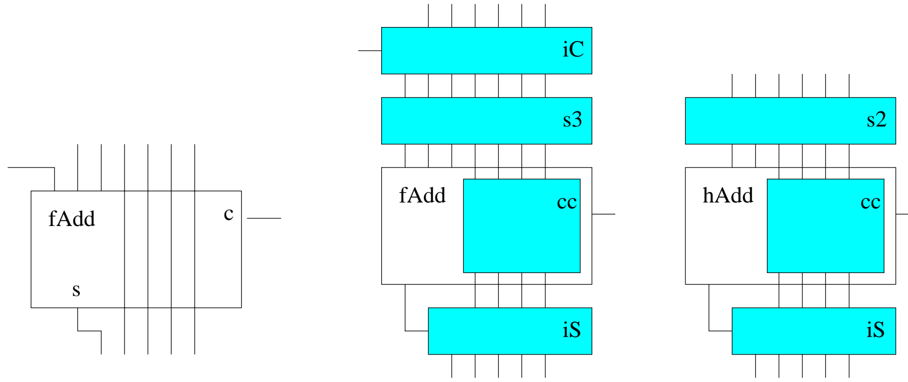

Fig. 2. (a) A specific fcell (b) A general fcell showing building blocks (c) hcell

Carries flow from left to right through the array, with an empty list of carries entering on the left and exiting on the right. Here, again, the `comps` parameter will later contain a tuple of building blocks. That we use a linear array means that we are considering the so-called column-compression multipliers. However, as we shall see, this class of multipliers is large, encompassing a great variety of well-known structures.

It remains to design the `compress` cell. It takes a pair of inputs, consisting of a list of carries on the left, and a list of partial products at the top. It should produce two bits at the bottom, and on the right a list of carries to be passed to the next column, which has weight one higher. All of the input bits to `compress` have the same weight. It must produce two bits of the same weight, and any necessary carries.

If we had a building block that reduced the number of product bits by one, with a carry-in and a carry-out, we would be well on the way. An example of such a `fcell` (in this case with six inputs at the top and five outputs at the bottom) is shown in Fig. 2(a). It is a kind of generalised full-adder, made from a full-adder and some wiring. Fig. 2(a) shows a specific instance, but we can make a general `fcell` by parameterising the component not only on the full adder but also on the wiring, as shown in Fig. 2(b).

The wiring component `iC`, for *insert Carry*, determines how the carry input, which comes from the left, is placed among the $n+1$ outputs of the `iC` block. If it is placed in the leftmost position, then that part of the cell looks like the particular cell shown in Figure 2(a). But there are many other choices, as we shall see later. Similarly, there are many choices for how the *insert Sum* component, `iS`, can look. In experimenting with wire-aware design, we have found that it makes sense to include *all* of the wiring as explicit components. One should think of tiling the plane, not only when building regular arrays of cells, but even inside the cells. So, we add two further components. `s3` divides $n+3$ wires into 3 that are passed to the full-adder and n that cross it. `cc` is the wiring by means of which

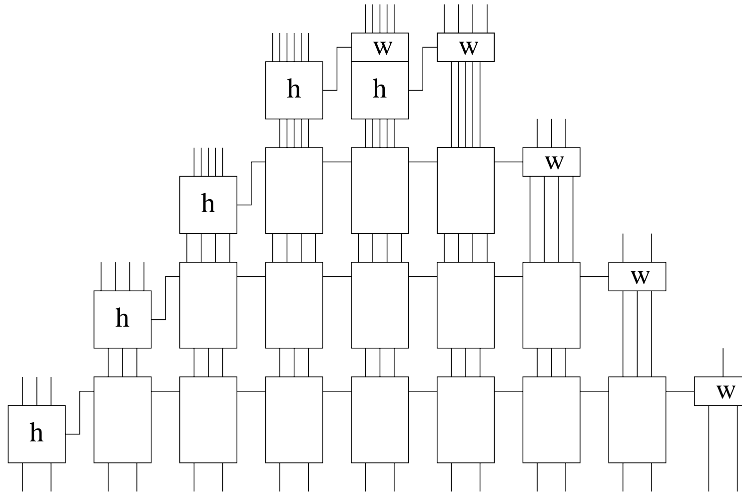

Fig. 3. The reduction array for 6 by 6 bit multiplication

those values cross the cell. `fcell` is defined using layout-oriented combinators, following the pattern shown in Figure 2(b).

A column of `fcell`s is exactly what we want for `compress` in the special case where the number of carries is exactly two less than the number of partial product bits. Then, the `fcell`s, reduce the partial products to two, one bit at a time. If this difference in length is greater than 2, we can rescue the situation by topping a recursive call of `compress` with a generalised half-adder cell that does not have a carry-in, using the combinator `|-`. The half-adder cell is called `hcell`, and is illustrated in Fig. 2(c). The `hcell` reduces the length difference by one, and can be thought of as handing the remaining problem to the smaller recursive call below it (see also the columns on the left in Fig. 3). It uses the same `iS`, `iC` and `cc` cells as the full-adder, and needs an `s2` cell that passes two of its inputs to the half-adder. (For a more precise analysis, it would be better to have different building blocks for the half- and full-adder, but we choose to reuse building blocks for simplicity.)

On the other hand, if the length difference is less than two, removing the topmost carry input and placing it among the partial products, using the *insert Carry* wiring that we have already seen adds two to the length difference. The `wcell` selects this piece of wiring from the tuple of building blocks. The recursive call that is placed below `wcell` again takes care of solving the remaining problem.

```
wcell (hAdd,fAdd,iS,iC,cc,s2,s3) = iC

compress comps (as,bs)
   | (diff >  2)  = (compress comps  |-  hcell comps)   (as,bs)
   | (diff == 2)  = column (fcell comps)                (as,bs)
   | (diff <  2)  = (compress comps  -|  wcell comps)   (as,bs)
  where diff = length bs - length as
```

Fig. 3 shows a row of compress components, applied to partial products of the shape produced by the partial product generator. The instances of `compress` on the left have a length difference of three (between the number of partial product inputs and the number of carry inputs), and so consist of a column of `fcells` below a `hcell`. (Unmarked cells in the diagram are `fcells`; those marked h are `hcells`, and those marked w are `wcells`.) Then, there is one instance where the difference is one, so it is `wcell` above a recursive call in which the difference is 3, that is is a column of `fcells` topped by `hcell`. Finally, there are several instances of `compress` where the difference is zero, and these are columns of `fcell` topped by `wcell`. This triangular shape contains the minimum hardware (in terms of half- and full-adder cells) needed to perform the reduction of partial products. To multiply two n-bit binary numbers, one needs $(n-1)(n-2)$ half- or full-adder cells in the reduction array, $n-1$ of them half-adders.

A row of compress cells can adapt itself to the shape of its inputs, as the definition of `compress` causes the right number of full or half-adders to be placed in each column. So such a row can function as a multi-operand adder, or function correctly with different encodings of the inputs. For the particular case of the reduction tree for a standard multiplier, there is no need to define a special triangular connection pattern to achieve the desired triangular shape. Such a specialised connection pattern would have lacked the flexibility of this approach.

A reduction array defined by the function `redArray` is also very general in another sense. By varying `iS` and `iC` wiring cells in the tuple of building blocks called `comps`, a great variety of different multipliers can be constructed, ranging from simple slow arrays to fast logarithmic Dadda-like trees. This surprisingly simple but general description of reduction arrays is, as far as we know, new. We had not expected to find such regularity even among the so-called irregular multipliers.

3 Making specific reduction arrays

A reduction array built using the function `redArray` is characterised by the tuple of building blocks: `(hAdd,fAdd,iS,iC,cc,s2,s3)` that is the `comps` parameter (see Figs. 2 and 3). For now, we fix `cc` to be the identity function, and `s2` and `s3` to be the functions `sep2 = splitAt 2` and `sep3 = splitAt 3` that split the list without performing any permutation. What we will vary between multipliers are the `iS` and `iC` wiring cells. Their role is to insert a single bit into a list of bits, to give a new list whose length is one longer than that of the original input list. The full-adder or half-adder that consumes some of the resulting list always works from the beginning, that is from the left in Fig. 2. So by choosing where to place the single bit, we also choose where in the array it will be processed.

3.1 A simple reduction array

If we place the single bit at the beginning of the list for both the sum and the carry case, the choice shown in Fig. 2(a), we make a reduction array that

consumes carries and sums as soon as possible. A carry-out is consumed by the next rightmost cell, and a sum output is consumed by the next cell down in the column. This is the reduction array that forms the basis of the standard linear array multiplier. It has only nearest neighbour connections between the full and half adder cells [6].

3.2 The Regular Reduction Tree multiplier

Postponing sums, by choosing `iS` to be `toEnd`, the function that places a value at the end of a list, but consuming carries immediately as before gives the Regular Reduction Tree multiplier proposed by Eriksson et al. [7].

3.3 A Dadda multiplier

A better choice, though, is to postpone both sums and carries, using the `toEnd` wiring function for both `iS` and `iC`. This gives a Dadda-like multiplier with both good performance and pleasing regularity. It is very similar to the modified Dadda multiplier proposed by Eriksson [6], but adds a simple strategy for layout.

3.4 A variety of arrays

If carries are consumed not in the current cell but in the cell below, and if sums are consumed as soon as possible, we get a variant on the carry-save array, called CCSA [6]. Similarly, other choices of the wiring cells give different arrays. We could easily describe yet more reduction arrays if we divided the triangular shape into two, roughly down the middle, and used different versions of the wiring cells in each half. Instead, we turn our attention to the problem of estimating gate and wire delays in data-paths like the reduction arrays.

4 Calculating gate delays

The description of the array is parameterised on the tuple of building blocks. This allows us to calculate gate delays simply by simulating a circuit made from versions of those cells that, instead of operating on bits, operate on integers representing delays. The non-standard cells model the delay behaviour of the real cells. For example, the delay-model of the half-adder is

```
halfAddI (as, bs, ac, bc) [a1,a2] = [s,cout]
   where
    s    = max (as+a1) (bs+a2)
    cout = max (ac+a1) (bc+a2)
```

It has four parameters representing the delay between each input and the sum and each input and the carry. The delay version of the full-adder is similar. Here, `max` and `+` are overloaded functions that work on both Haskell integers and the integers that flow in circuits; as a result, `halfAddI` is also overloaded.

A standard approach to the analysis of partial product reduction trees is to count delays in terms of *xor* gate delay equivalents [11]. For the half-adder, this would give cross-cell delays of (1,1,0.5,0.5), for instance. Since we will simulate circuits carrying delay values, we are restricted to integer delays, and so must multiply all of these numbers by 10. Thus, we define half- and full-adder delay estimation circuits as

```
hI as = halfAddI (10,10,5,5) as
fI as = fullAddI (20,20,10,10,10,10) as

Main> fI [0,5,5]
[25,15]
```

hI and fI are again overloaded to work both at the Haskell level and the circuit level. This overloading will prove useful later, when these delay-modelling cells will be used first at the circuit level, that is during simulation, and then at the Haskell level, that is during generation. This is a standard and simple delay model, giving worst-case delays without taking account of false paths. For the moment, it is sufficient for our purposes, though in future work we will want to look at more complex delay estimation.

To make a delay estimation version of a reduction array, we simply replace the half- and full-adder in its component tuple by hI and fI. We leave the wiring cells alone; they are polymorphic and so can operate on either bits or integers as required. To find the gate delay leading to each output, we feed zeros into the resulting circuit, and simulate. The function ppzs n produces a list of lists of zeros of the shape expected by the array.

```
dDadG n = simulate (redArray (hI,fI,toEnd,toEnd,id,sep2,sep3)) (ppzs n)

Main> dDadG 16
[[0,10],[5,20],[20,30],[30,40],[40,50],[50,50],[50,60],[60,70],[70,70],
[70,70],[70,80],[70,80],[80,90],[90,90],[90,90],[90,90],[90,90],[90,90],
[80,90],[80,80],[70,80],[70,80],[70,70],[60,70],[60,60],[50,60],[50,50],
[40,20],[0,20]]
```

The Dadda array has a maximum delay of 9 xor-gate delays at size 16 by 16. By comparison, the linear array (from section 3.1) of the same size has a maximum delay of 41 xor-gate delays. The next step is to take account of wire delays.

5 Taking account of wire delays

The wires whose delays are of interest to us are those that cross the cells, that is they correspond to the cc parameter in the tuple of functional and wiring components (see Fig. 2). So, in our delay calculation, we can just replace this parameter, which was the identity function before, with a function that adds a fixed value d to the incoming delay. The gate delay models are hI and fI, as before. However, this is not quite right. When we check the various delay profiles using this approach, we find that *all* of the array topologies incur long wire delays. Why is this? It is because our triangular shaped array takes all of

the input partial products at the top, and allows them to flow down to the cell that processes them. So, for example, in the linear array, there is a long wire carrying a partial product bit right from the top of the array to the bottom. But the delay on these partial product carrying wires is not of interest to us in the current analysis (though we would want to count these delays if we implemented the array in exactly this form). The standard approach to comparing reduction arrays in the literature is to ignore the problem of getting each partial product bit to the cell that processes it. In real arrays, the partial product production is typically mixed with the reduction array, and the problem of distributing the multiplier inputs to the right positions is handled separately. We would like, at a later date, to tackle the analysis of such a mixed partial product production and reduction array, including all wire delays. Here, though, we will analyse just the reduction array, and will count delays only in the inter-cell wires, that is the sum and carry wires that run across cells from top to bottom.

To achieve this, we tag each wire that might contribute delays in the simulation with a Boolean, indicating whether it is a partial product or is an output from a full- or half-adder. The partial product inputs are tagged with `True`. Those wires do not contribute to the delay, while wires tagged with False do.

```
wireIB d (m,b) = if b then (m,b) else (m+d,b)

cross d = map (wireIB d)
```

So now, in the tuple used in delay simulation, we can replace the `cc` wiring cell by `cross d`. As a result, the half- and full-adder cells must also change, to be able to accommodate the new Boolean tags. For instance, to make the hybrid full-adder circuit, `fIB`, we combine `fI` with an abstract half-adder, `fB`, that takes Booleans as input and outputs `[False, False]`, indicating that the corresponding wires do not carry partial product bits. The two full-adder variants operate completely independently, one working on delay values, and the other on Boolean tags.

```
fIB as = (fI // fB) as

Main> fIB [(0,True),(5,True),(5,True)]
[(25,False),(15,False)]
```

The hybrid half-adder, `hIB`, is constructed similarly. The `sep2` and `sep3` wiring cells are polymorphic, and so do not need to change, even though the types of values that flow along their wires are now different. To study the gate and wire delay behaviour of a particular array, we construct a component tuple with `hIB` in the half-adder position, `fIB` in the full-adder position, and `cross d` as the cross-cell wiring. The necessary Boolean tags are inserted and removed by the functions `markTrue` and `unmark`, while `getmax` returns the largest delay. For larger sizes, the effects of wire delay become significant. For 53 bit multiplication, the maximum delays for the Dadda reduction array range from 15 xor-gates for zero cross-cell wire delay to 32 for a wire delay of 4.

```
maxDel f n = simulate (markTrue ->- f ->- unmark ->- getmax) (ppzs n)

mDad d = maxDel (redArray(hIB, fIB, toEnd, toEnd, cross d, sep2, sep3))
```

```
Main> [(i,mDad i 53) | i <- [0..4]]
[(0,150),(1,189),(2,230),(3,275),(4,320)]
```

It would be interesting to develop further analyses to help in understanding the different delay behaviours of the multipliers, as well as to repeat these calculations with different settings of the cell and wire delays, for various sizes, and perhaps in a more symbolic way. Here, we continue our series of multiplier descriptions by considering a multiplier in which the wiring depends, in a systematic way, on the estimated delay in the cells and the wires.

6 A cleverer multiplier

In simpler circuit descriptions, it is usual to use a Haskell integer variable to control the size of the generated circuit. *Clever circuits* is a more sophisticated programming idiom in which Haskell values that are more closely tied to the circuit structure are used to guide circuit generation [12]. In the previous section, we used this idea to build a circuit to perform delay calculations. The Boolean tags were what we call *shadow values* – Haskell-level values that were used to guide the generation of the delay-estimation circuit. The shadow values could be seen as controlling the production of a net-list containing wire-modelling components that add `d` to their inputs as well as `fI` and `hI` components. This ensured that only certain wires of the original circuit contributed to the delay estimation.

Here, we apply the same idea one step earlier, during the initial circuit generation. In this case, though, we use clever circuits to control the generation of *wiring* rather than of components. We have shown how multipliers can be parameterised on the `iS` and `iC` cells, and we made various decisions about their design, always choosing a fixed piece of wiring for each of these cells, for each array. But why not make these cells react to the delays on their inputs during circuit generation, and configure themselves accordingly, resulting in possibly different choices throughout the multiplier? This appealing idea is similar to that used in the TDM multiplier, and in related work [11, 13, 1]. Here, we show how well it fits into our general reduction array description, giving an adaptive array that can depend not only on gate delays (as in the TDM), but also on delays in the wires crossing cells.

The first step is to make a cell that becomes either the identity on two bits, or a crossing (or swap), depending on the values on its shadow inputs. If the predicate `p` holds of `x` and `y`, the swap is performed; otherwise the output is the same as the input.

```
cswap p ((a,x),(b,y)) = if (p x y) then ((b,y),(a,x)) else ((a,x),(b,y))
```

Now, during circuit generation, if `cswap` receives, on its shadow inputs, two values for which `p` holds, then it becomes a crossing, otherwise it is the identity. And once the circuit has been generated, all record of the shadow values that influenced its shape have disappeared.

The clever wiring cells should (like `iC` and `iS`) have a pair of inputs, consisting of a single value and a list. We assume that the list of inputs is in increasing order of delay. Then, we want to insert the single value into the correct position in the list, so that the resulting list is also in increasing order of delay, and so presents the wires with the least delay for use by the next half- or full-adder. We do this using a row of `cswap` components, and then sticking the second output of the row onto the end of the first, using append right (`apr`):

```
cInsert = row (cswap p) ->- apr
  where p (g1,b1) (g2,b2) = g1 > (g2::Int)
```

This is similar to the row of comparators that is the insertion component of insertion sort. The predicate `p` compares the integer delay values. Depending on the delays of the input wires, various wirings can be formed, ranging from the simplest `apl`, in which no swaps are made, to `toEnd`, in which all the possible swaps are made, placing the single input at the end of the list. The important point is that the wiring adapts to its position in the array, based on the delay information that it receives during circuit generation. We replace `iS` and `iC` by `cInsert`.

What remains to be done is to make shadow versions of all of the other components, and to combine them with their concrete counterparts. These shadow versions must work on Haskell-level (integer, Boolean) pairs. We have, however, already constructed `hIB` and `fIB` cells that work on such pairs; we simply reuse them here (exploiting the fact that `hI` can also work on Haskell integers).

```
adapt (hAdd, fAdd, cc) (d,pds)
  =  mmark pds ->-  redArray (hAdd // hIB,
                              fAdd // fIB,
                              cInsert, cInsert,
                              cc // cross d,
                              sep2, sep3)  ->- unmark
```

`adapt` defines the adaptive reduction array. It is just a call of the `redArray` function with a suitable tuple of building blocks. The `mmark pds` function sets up the shadow values correctly, based on the input delay profile `pds`, and `unmark` removes the shadow values on the outputs. Thus, `adapt` is parameterised not only on the half-adder, full-adder and the cross-cell wiring, but also on the delay `d` in that wiring and on the delay profile of the incoming partial products, `pds`. The latter two parameters influence the formation of the circuit, that is they control exactly what wiring pattern each instance of `cInsert` becomes in the final circuit. Each sublist of `pds` is assumed to be in delay sorted order (and this could, if necessary, be achieved by further use of clever wiring).

If, during generation, we choose the cross-cell wire delay to be zero, and the input delay profile to be all zeros, we get a basic TDM-style multiplier [11]. Measuring both gate and wire delays, with a cross-cell delay of 2, for 16 by 16 bit multiplication, the TDM array has a maximum delay of 116 units (where 10 units is one xor-gate delay). If the same wire delay is used during the generation of the array, the maximum delay reduces to 100. We call the resulting array wire-adaptive. For comparison, the modified Dadda array has a maximum gate and

wire delay of 122 units for this size. For 64 bit multiplication, the delays for the wire-adaptive, TDM and Dadda arrays are 234, 258 and 266 respectively, while the corresponding figures for 80 bits are 270, 300 and 302. So it makes sense to take account of wire delay while generating reduction arrays. Making use of the input delay profile during circuit generation further improves the resulting array.

7 Taking account of constraints on available tracks

In real circuits, there are typically constraints on the number of tracks available for cross-cell wiring. Here, again, we are concerned only with the sum and carry wires, and we do not consider the problem of routing the partial product wires to the correct positions. The circuits that we have seen so far took no account of constraints on wiring tracks. Here, we demonstrate the versatility of our approach to writing array generators by incorporating such a constraint.

In each cell, we would like to limit the number of sum and carry wires that can cross the cell to be a maximum of *tr*, the number of available tracks. This can be done by using clever wiring in new versions of the `s2` and `s3` wiring cells (see Fig. 2). Let us consider the case of `s3`. When the number of sum or carry wires crossing the cell below it is in danger of becoming too large, `fcon tr` moves one or two such wires leftwards, as necessary, so that they are consumed by the cell, instead of crossing it. The function `move (m,n) p` ensures that there are at least `n` elements satisfying `p` in the first output list, which should be of length `m`. The new version of `s2` is `hcon tr`, which is similar to `fcon tr`.

```
fcon tr as
   | (l <  tr+1)  = move (3,0) p as
   | (l == tr+1)  = move (3,1) p as
   | (l == tr+2)  = move (3,2) p as
  where
    l = length (filter p as)
    p (c,(i,b)) = not b
```

`adaptC` is the same as `adapt`, apart from the addition of the `tr` parameter and the replacement of `sep2` and `sep3` by the two adaptive wiring cells, `hcon tr` and `fcon tr`, in the tuple of building blocks. The result is a remarkably powerful reduction array generator. Varying the available number of tracks gives us even more variety in the arrays produced, and allows us to trade off delay against wiring resources. The function `mAdCon` takes as parameters the number of cross-cell tracks, the wire delay to use during generation and simulation, and the size of the two numbers being multiplied; it returns the maximum delay on an output in the reduction tree, for zero input delay. For 18 bits, and ignoring wire delays, having zero tracks gives a linear array with 42 xor-gate delays, and the delay decreases as more tracks are made available, reaching the minimum 9 gate delays for 7 or more tracks.

```
mAdCon tr d1 d2 n = maxDel (adaptC tr (hIB,fIB,cross d2) (d1,pps 0 n)) n

Main> [(i,mAdCon i 0 0 18) | i <- [0..8]]
[(0,420),(1,235),(2,155),(3,125),(4,110),(5,100),(6,100),(7,90),(8,90)]
```

Finer delay modelling, for example the use of calls to external analysis tools, would give better multiplier performance. Here, we assumed that delay increases linearly with wire length. We will investigate a model in which delay is proportional to the square of the length. This would be easy to incorporate. We will also consider the automatic generation of buffers on long wires.

8 Related work

The most widely used partial product reduction tree is the Wallace tree [15]. It can be thought of as being made from horizontal blocks that are carry-save adders, rather than from vertical `compress` blocks as here. However, the Dadda tree [5], particularly in the modified form shown here, gives comparable performance in both delay and power [6].

Luk and Vuillemin presented, analysed and implemented a family of fast recursively-defined multiplication algorithms [8]. One of these is a linear array of bit-convolver circuits, where the bit-convolver has the same function as our `compress` cells. The bit-convolvers were recursively defined as binary trees, and laid out with the root in the middle and the two smaller trees above and below it (as is standard in adder trees). This corresponds, also, to implementing the entire multiplier as a rectangular array with two half-sized recursive calls sandwiching the final adder. The corresponding implementation compared well with Wallace trees for smaller sizes. The emphasis in the work of Luk and Vuillemin is also on the generation of multipliers from readable descriptions containing some geometric information. The importance of parameterised generators is stressed, and formal verification (of fixed size circuits) is also considered. This work is an important inspiration for ours. An adaptive tree should be faster than a static binary one, but it may be that we need to lay out the tree with the root in the middle. This needs to be investigated in future work.

The original TDM multiplier generation method considers gate delay in choosing how to wire up a partial product reduction array to minimise maximum gate delay [11]. When our adaptive array works with the standard gate delay model and zero input and wire delay, it is mimicking exactly the original TDM method, and achieves identical gate delays. The original TDM uses what the authors call a 3-greedy algorithm; the three fastest wires are chosen for connection to the next full-adder. Like our adaptive array, the basic TDM method can be adapted to take account of the input delay profile. Later work showed that the 3-greedy method produces near-optimal results for maximum delay, but that a more sophisticated 2-greedy method produces better delay profiles because it allows a more global optimisation [13]. The analysis and the new generation algorithms are elegant, but the resulting method uses expensive search, which neither the basic TDM nor our approach requires. We are interested in seeing how far we can get with symbolic evaluation, which requires no search and so scales up well.

The TDM method and its more sophisticated successors do not currently take account of wire delay and the authors mention that this is a natural next

step. The methods do not either take account of constraints on tracks. They could, presumably, be adapted to do so. Al-Twaijry et al present a partial product reduction tree generator that aims to take account of both wire delay and constraints on available tracks [1]. This work seems close to ours. The big difference is in the method of writing the generators. Ours are short, and retain the structure of the array, with parameterisation coming from local changes inside the cells, whereas the standard approach is to write C code that generates the required net-list. We are hopeful that our generators will be more amenable to formal verification.

Our approach can be seen as deciding on the placement of the `hcells`, `fcells` and `wcells` in advance, and then making a multiplier with the required properties by forming the wiring inside those cells during generation, making use of context information such as input delay and constraints on tracks. Thus, it is important to keep that placement when producing the final circuit. Some approaches aim, instead, to produce unplaced net-lists that will give short wires when given to a synthesis tool that does the placement [14]. Both our approach and the TDM methods use half- and full-adders as the building blocks. The work of Um and Kim uses entire carry-save adders as building blocks, arguing that this gives more regular interconnections. A final adjustment phase aims to get some of the advantages of using a finer granularity. Our approach is very different because we aim to give the writer of the generator control over placement, and thus over degree of regularity. We are not restricted to the triangular placement described here, and intend to experiment with other layouts.

To our knowledge, the best approach to the design of reduction trees under constraints on tracks is based on the over-turned stairs (OS) trees [10]. The OS trees are a sequence of increasingly sophisticated recursive constructions of delay-efficient trees of carry-save adders that are designed to minimise cross-cell tracks. Although the method of counting cross-cell tracks is not identical to ours (since we consider a carry-in that is not used by the current cell to cross the cell), we believe that our simple constrained reduction trees have longer delay than the reported results (which give only gate delays) for higher order OS trees for larger sizes. We will consider ways to improve our algorithm, and will do a more detailed comparison.

9 Discussion

We have given a short parameterised structural description of a partial product reduction tree made from `hcells`, `fcells` and `wcells` (see Figs. 2 and 3). We then showed how a great variety of multipliers can be obtained by varying the building blocks (`iS`, `iC` etc.) of those cells, while retaining the same overall structure. First, we made standard arrays, including a modified version of the Dadda tree. Because of the surprising regularity of the Dadda array, a colleague, Henrik Eriksson, was able to lay it out with ease in a manual design flow. The measured results for delay and power consumption are promising and a paper is in preparation. It is usual to dismiss both Dadda and Wallace trees as difficult

to lay out. We have shown that a Dadda tree can be made regular. In Lava, we performed a simple delay estimation on the various arrays by using non-standard interpretation. We replaced circuit components, including wires, by non-standard versions that estimated their delay, and then performed simulation.

The next step was to use the same kind of delay modelling *during circuit generation*. The idea of *clever circuits*, that is circuits that can adapt to context information, was combined with the delay modelling to control the production of *wiring cells*. This allowed us to produce fast arrays that can adapt to gate and wire delay, and to the delay profile of their inputs. Next, we showed that further use of clever wiring allowed the generator to adapt to constraints on wiring resources. A very high degree of parameterisation is achieved by *local* changes in the cells. This was made possible by the initial insights into the general structure of reduction arrays. The result is a powerful circuit generator that looks exactly like a structural description of a simple multiplier, and so is very small. We consider this to be a very promising approach to the writing of data-path generators. The combination of clever circuits with the connection pattern approach that we have long used in Lava seems to give a sudden increase in expressive power. Here, we used clever wiring to get parameterisation, but in other applications, we will need to control the choice of components, or even of recursive decompositions. The latter arises, for example, in the generation of parallel prefix circuits, where different recursive decompositions give different area, delay and fanout tradeoffs. We are currently investigating ways to choose recursion patterns dynamically during circuit generation, partly because we plan to use these methods to make the fast adder that is needed to complete a multiplier design. Such an adder must adapt to the delay profile of the reduction tree that feeds it.

We would like to verify the generators once and for all, for all sizes and parameters. Our hope is to be able to verify a generator as though it were a circuit description, exploiting the fact that the generator is structured as a circuit (and is short and simple). To do this, we expect to build on the seminal work by Hunt and his co-workers on the use and verification of circuit generators [3] and on recent work on the verification of parameterised circuits [2]. We will probably have to develop verification techniques that are specialised to our style of writing circuit generators. We are interested in investigating the use of a first order theorem prover to automatically prove the base case and step of the necessary inductive proofs.

The ideas presented here could be used to implement module generators that provide designers with access to a library of highly flexible adaptive data-paths, without demanding that the designer construct the modules himself. This would allow the incorporation of these ideas into more standard design flows. This vision is compatible with the timing-driven module-based design flow that was outlined recently by Mo and Brayton [9].

Our next step will be to implement and analyse a fast adaptive multiplier in our Wired system, in which geometry is considered, and the exact placement of each wire is accounted for.

Acknowledgements

This work is supported by the Semiconductor Research Corporation (task id. 1041.001), and by the Swedish funding agency Vetenskapsrådet. We gratefully acknowledge an equipment grant from Intel Corporation.

References

1. H. Al-Twaijry and M. Aloqeely: An algorithmic approach to building datapath multipliers using (3,2) counters, *IEEE Computer Society Workshop on VLSI*, IEEE Press, Apr. 2000.
2. C. Berg, C. Jacobi and D. Kroening: Formal verification of a basic circuits library, *IASTED International Conference on Applied Informatics*, ACTA Press, 2001.
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 and M. Sheeran: A Tutorial on Lava: A Hardware Description and Verification System, Apr. 2000. http://www.cs.chalmers.se/~koen/Lava/tutorial.ps
5. L. Dadda: Some Schemes for Parallel Adders, *Acta Frequenza*, vol. 34, no. 5, May 1965.
6. H. Eriksson: Efficient Implementation and Analysis of CMOS Arithmetic Circuits, PhD. Thesis, Chalmers University of Technology, Dec. 2003.
7. H. Eriksson, P. Larsson-Edefors and W.P. Marnane: A Regular Parallel Multiplier Which Utilizes Multiple Carry-Propagate Adders, *Int. Symp. on Circuits and Systems*, 2001.
8. W. K. Luk and J. E. Vuillemin: Recursive Implementation of Optimal Time VLSI Integer Multipliers, *VLSI'83*, Elsevier Science Publishes B.V. (North-Holland), Aug. 1983.
9. F. Mo and R.K. Brayton: A Timing-Driven Module-Based Chip Design Flow, *41st Design Automation Conference*, ACM Press, 2004.
10. Z.-J. Mou and F. Jutand: "Overturned-Stairs" Adder Trees and Multiplier Design, *IEEE Trans. on Computers*, Vol. 41, No. 8, Aug. 1992.
11. V.G. Oklobdzija, D. Villeger and S.S. Liu: A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach. *IEEE Trans. on Computers*, vol. 45, no. 3, Mar. 1996.
12. M. Sheeran: Finding regularity: describing and analysing circuits that are not quite regular, *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME*, LNCS 2860, Springer-Verlag, 2003.
13. P.F. Stelling, C.U. Martel, V.G. Oklobdzija and R. Ravi: Optimal Circuits for Parallel Multipliers, *IEEE Trans. on Computers*, vol. 47, no. 3, Mar. 1998.
14. J. Um and T. Kim: Synthesis of Arithmetic Circuits Considering Layout Effects, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 11, Nov. 2003.
15. C. S. Wallace: A suggestion for a fast multiplier, *IEEE Trans. on Computers*, EC-13, 2, Feb. 1964.