Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

High-Performance Adder Circuit Generators in Parameterized Structural VHDL

Hanspeter Kunz and Reto Zimmermann

Technical Report No. 96/7
August 1996

Abstract

In ASIC design, arithmetic components are usually selected from tool- and technology-dependent libraries providing very limited flexibility and choice of circuit structures. With the possibility of parameterized structural circuit descriptions at the gate-level in VHDL, versatile circuit generators can be implemented which are highly independent of tool platforms and design technologies. This enables the realization of a universal and comprehensive library of efficient arithmetic components in form of a collection of synthesizable VHDL code entities. In a first step, high-performance adder generators were implemented using this method. Additionally, valuable experience was gained with respect to the implementation of circuit generators using parameterized structural VHDL.

This work was funded by MICROSWISS (Microelectronics Program of the Swiss Government).

Abstract

In ASIC design, arithmetic components are usually selected from tool- and technology-dependent libraries providing very limited flexibility and choice of circuit structures. With the possibility of parameterized structural circuit descriptions at the gate-level in VHDL, versatile circuit generators can be implemented which are highly independent of tool platforms and design technologies. This enables the realization of a universal and comprehensive library of efficient arithmetic components in form of a collection of synthesizable VHDL code entities. In a first step, high-performance adder generators were implemented using this method. Additionally, valuable experience was gained with respect to the implementation of circuit generators using parameterized structural VHDL.

1 Introduction

Typical data-processing ASICs implement algorithms involving arithmetic computations. One possibility to describe such arithmetic computations at a high level of abstraction is the usage of *behavioral VHDL*. At this level the addition of two binary numbers A and B is simply written as

```
S <= A + B;
```

During synthesis this abstract description is translated (or mapped) to the structural or gate level. This is done automatically leaving only very limited control to the designer. At the same time, this mapping determines the performance characteristics of the generated circuit, such as speed, area requirements, and power dissipation. In particular, the mapping from the behavioral to the structural level includes the decision for a particular circuit architecture, which greatly influences the properties mentioned above. Put differently, the performance of the final circuit is determined by the quality of the algorithms used for structural synthesis, which in turn depends on the libraries and design tools used.

A viable alternative is the direct implementation of a circuit at the structural level using schematic entry or *structural VHDL*. This holds true especially when efficient circuit structures that satisfy ones special requirements are known. Despite the great progress in the development of algorithms for logic optimization, the potential of these universal techniques is limited to the optimization of random logic and to rather local optimizations within complex and already highly factorized networks. On the other hand, efficient arithmetic circuits base on optimized structures with a high degree of factorization which are obtained by specialized circuit generators rather than generic optimization algorithms. This in turn makes an initial design of arithmetic networks at the structural level necessary, yielding circuits with higher performance at the expense of an increased design effort.

The simplest way to design a circuit with a dedicated architecture is to describe its netlist by way of schematic or textual entry. Such a netlist, however, is neither scalable nor easy to reuse, modify, and maintain. Furthermore, it lacks portability among different cell libraries as well as design tools.

A better approach is to describe the circuit in structural VHDL. Structural VHDL is independent of development environments and libraries, or in other words, it is portable. In structural VHDL, as opposed to behavioral VHDL, *netlist generators* can be described implementing circuits having a dedicated architecture. Furthermore, this can be done in a *parameterized* and thus *scalable* form. Therefore, a comprehensive library of flexible arithmetic components in synthesizable VHDL code would be of interest. ASIC design productivity can be increased considerably by relying on such a library of sophisticated and proven arithmetic components ready for synthesis.

One of the most often used and basic arithmetic operations is the addition of two binary numbers. As SKLANSKY said in 1960 [1]:

> At the present state of the computer art, adders are essential not only for addition, but also for subtraction, multiplication, and division. [...] Addition logic is thus of obvious importance, and has received quite a bit of attention.

This statement is still valid. Efficient implementation of adder circuits has been investigated over a long period of time and by many people. As a result there exists a large number of different circuit architectures with different performance characteristics.

Two particular adder architectures described in the sequel were implemented in a scalable form in structural VHDL. The two major goals were to investigate the suitability of structural VHDL for the description of parameterized arithmetic components on one hand and the realization of an arithmetic library of adder components on the other hand.

This report is organized as follows. Section 2 describes the implemented adder structures. Section 3 introduces some basics regarding the description and generation of logic netlists in structural VHDL. Section 4 reports the two different approaches taken for implementation of the chosen adder structures in structural VHDL. In the remaining sections results and experiences are summarized with outlook towards the development of a comprehensive library of arithmetic components.

2 Adder Structures

The basic theory and the practical implementation of *parallel-prefix addition* are discussed now. More theoretical background can be found in [2][3][4][5][6].

2.1 Parallel-Prefix Addition: Theory

Some combinational circuits can be described in terms of parallel-prefix logic. Carry-propagation in binary addition is a prefix problem [6].

A parallel-prefix logic combines n inputs

$$x_{n-1}, x_{n-2}, \ldots, x_0 \tag{1}$$

using an arbitrary associative operator $\circ$ to n outputs

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_1 \circ y_0 = x_1 \circ x_0 \\ &\vdots \\ y_{n-1} &= x_{n-1} \circ y_{n-2} = x_{n-1} \circ x_{n-2} \circ \ldots \circ x_0 \end{aligned} \tag{2}$$

so that output y_i depends only on inputs $x_{j \leq i}$.

The addition of two n-bit binary numbers $A = a_{n-1}a_{n-2}\cdots a_0$ and $B = b_{n-1}b_{n-2}\cdots b_0$ and an input carry c_{in} can be formulated as

$$\begin{aligned} c_0 &= c_{in} \\ c_{i+1} &= a_i b_i + (a_i + b_i)\, c_i \\ s_i &= a_i \oplus b_i \oplus c_i \\ c_{out} &= c_n \end{aligned} \tag{3}$$

$i = 0, \ldots, n-1$, yielding the sum $S = s_{n-1}s_{n-2}\cdots s_0$ and the carries c_i as intermediate signals.

The key of fast addition is the fast calculation of the carries c_i. Alternatively, they can be expressed according to

$$c_{i+1} = g_i + p_i c_i \tag{4}$$

with the *generate* signal

$$g_i = \begin{cases} a_i b_i & \text{if } 1 \leq i < n \\ a_0 b_0 + a_0 c_0 + b_0 c_0 & \text{if } i = 0 \end{cases} \tag{5}$$

and the *propagate* signal

$$p_i = a_i \oplus b_i \tag{6}$$

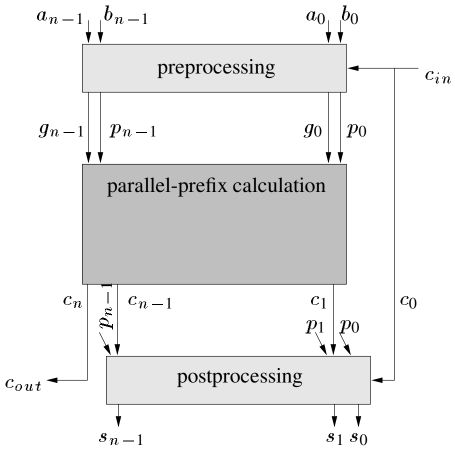


Figure 1: *The three stages of a parallel-prefix addition.*

By recursive substitution the i-th carry can be calculated as

$$c_{i+1} = g_i + \sum_{j=0}^{i-1} \left(\prod_{k=j+1}^{i} p_k \right) g_j + \left(\prod_{k=0}^{i} p_k \right) c_0 \tag{7}$$

and finally the sum bits as

$$s_i = p_i \oplus c_i \tag{8}$$

By defining the operation $\circ$ on ordered bit pairs (g, p)

$$(g_i, p_i) \circ (g_j, p_j) = (g_i + p_i g_j, p_i p_j) \tag{9}$$

the equation (7) can be written as

$$(c_{i+1}, p_0 \cdots p_i) = (g_i, p_i) \circ \cdots \circ (g_0, p_0) \tag{10}$$

Thus, the carries c_i can be calculated using a prefix algorithm where $\circ$ is defined according to equation 9.

Note that the $\circ$ operator is *associative* but *not commutative*.

2.2 Parallel-Prefix Addition: Implementation

In practice parallel-prefix addition is carried out in three consecutive steps: the *preprocessing*, the *parallel-prefix carry calculation* and the *postprocessing stage* (see Fig. 1). The preprocessing stage implements the equations (5) and (6), while the postprocessing stage realizes equation (8). We will discuss these two simple stages later and focus now on the parallel-prefix carry computation.

Performing the parallel-prefix calculation is equivalent to evaluating equation (10) for each bit position i,

$0 \le i < n$. Since the $\circ$ operator is not commutative the order of the operands must no be changed. Due to the associativity of the $\circ$ operation its evaluation must not necessarily be done serially

$$\underbrace{(g_3,p_3)\circ\underbrace{((g_2,p_2)\circ(\underbrace{(g_1,p_1)\circ(g_0,p_0)}))}}$$

but can be carried out in any order, e.g.

$$\underbrace{(\underbrace{(g_3,p_3)\circ(g_2,p_2)})\circ(\underbrace{(g_1,p_1)\circ(g_0,p_0)})}$$

In particular, the $\circ$ operations can be evaluated according to a binary tree structure. Thereby, evaluations on different branches of the tree are done in parallel, while the height of the tree is determined by the maximum number of evaluations in series. This gives a measure for the overall evaluation time which is of complexity $O(\log n)$.

For the computation of all n carries c_i, n binary evaluation trees are required having an overall area complexity of $O(n^2)$. By sharing subtrees the circuit complexity can be reduced down to $O(n \log n)$. Various schemes for the combination of subtrees exist, resulting in different parallel-prefix algorithms. These algorithms can best be visualized using *directed acyclic graphs* with the graph nodes representing the logic cells performing the $\circ$ operations and with the graph edges representing the circuit nodes for the signal connections. In order to avoid confusion, cells denote circuit cells (or graph nodes) and nodes denote circuit nodes (or graph edges) in the sequel.

In order to capture the graph structure of the parallel-prefix algorithms we have to extend our mathematical notation. The vector

$$\mathbf{v}_{i,j} = (g_{i,j}, p_{i,j}) \tag{11}$$

denotes the generate-propagate signal pair from the cell (i,j) to the cell $(i, j+1)$, where i is the bit number and j

$$1 \le j \le h \tag{12}$$

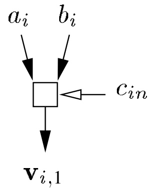
represents the row number in the graph (h is the height of the graph).

Now we take a closer look at the three stages of parallel-prefix addition. We use a notation first proposed by BRENT and KUNG [4] and extended by LINDKVIST and ANDERSSON [6], and make some further extensions.

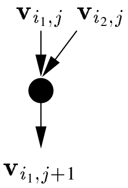
The *preprocessing* stage generates the signals the parallel-prefix algorithm operates on, namely the generate and propagate signals

$$\left.\begin{array}{l} g_{i,1} = g_i \\ p_{i,1} = p_i \end{array}\right\} \mathbf{v}_{i,1} \tag{13}$$

according to equations (5) and (6). In our graph notation this logic is depicted by *square* cells.



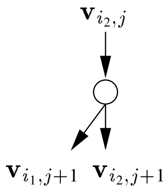
Based on the vectors $\mathbf{v}_{i,1}$ the *parallel-prefix* stage computes the carries c_i. For regularity reasons the parallel-prefix graphs are composed of three types of cells. The *black* cells



perform the $\circ$ operation

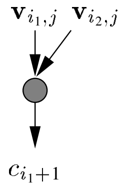
$$\mathbf{v}_{i_1,j+1} = \mathbf{v}_{i_2,j} \circ \mathbf{v}_{i_1,j} \quad (i_1 > i_2) \tag{14}$$

while the *white* cells



are empty, i.e. they simply copy the input to their output(s).

The *grey* cells



are basically simplified black cells. They perform the last $\circ$ operation on bit i, and their output $g_{i_1,j+1}$ corresponds to the carry c_{i_1+1}. The calculation of $p_{i_1,j+1}$ is omitted since this signal is not used. Thus, the grey cells perform the reduced $\circ'$ operation

$$c_{i_1+1} = \mathbf{v}_{i_2,j} \circ' \mathbf{v}_{i_1,j} \quad (i_1 > i_2) \tag{15}$$

$$(g_{i_2,j}, p_{i_2,j}) \circ' (g_{i_1,j}, p_{i_1,j}) = g_{i_1,j} + p_{i_1,j} g_{i_2,j} \tag{16}$$

All the carries c_i are computed at the end of the parallel-prefix stage. Finally the sum bits s_i are calculated according to equation (8). This *postprocessing* is performed by the *triangle* cells.

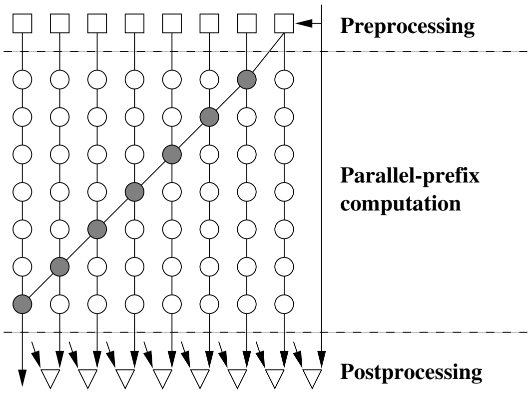
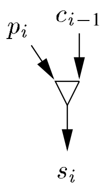


Figure 2: *8-bit ripple-carry adder represented as parallel-prefix graph.*



Parallel-prefix addition can now be illustrated using simple graphical representations. As an example, Fig. 2 shows the prefix structure of an 8-bit ripple-carry adder, which actually is a *serial*-prefix algorithm. Various algorithm properties are visible in this graph. The number of subsequent cells a node is connected to corresponds to its fan-out, and the number of edges corresponds to the amount of wires. The number of rows denotes the maximum number of $\circ$ evaluations to be performed in series and can be interpreted as the delay or the number of pipeline stages in a pipelined realization of the algorithm. Because all operations in a row are executed in parallel, the number of black cells in one row corresponds to the degree of parallelism in that step. In particular, the effective speed of a realization of an algorithm is determined by the number of stages and by the fan-out of the cells.

There exists a wide range of proposed parallel-prefix algorithms. The two parallel-prefix algorithms used here are the one proposed by SKLANSKY [7] (Fig. 3) and the one by BRENT and KUNG [4] (Fig. 4). The properties of these two addition algorithms are summarized in Table 1.

SKLANSKY's prefix algorithm, first used for conditional-sum addition [7], is one of the most common prefix algorithms. This algorithm has minimal depth but the fan-out increases exponentially towards the final stages. The maximum fan-out is linear to the number of operand bits.

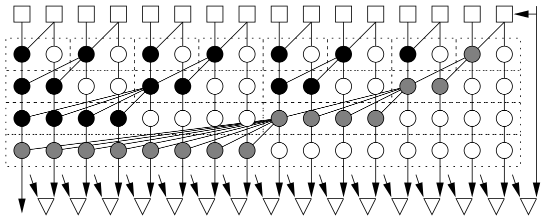


Figure 3: SKLANSKY*'s prefix algorithm.*

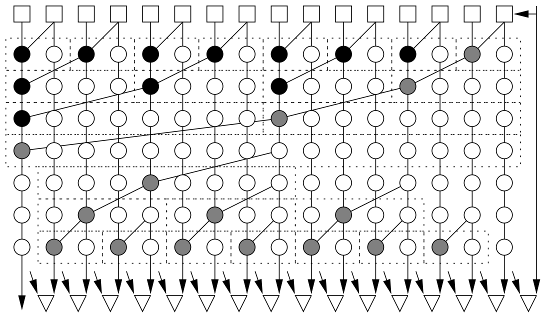


Figure 4: BRENT *and* KUNG*'s prefix algorithm.*

BRENT and KUNG's prefix algorithm has low fan-out (i.e. $O(\log n)$ instead of $O(n)$) but twice the depth of the SKLANSKY algorithm. BRENT and KUNG's prefix algorithm is quite area efficient due to the small number of black cells (remember that the white cells contain no logic) and due to the low wiring requirements.

The graphs illustrate the simple and highly regular structure of both prefix algorithms. The regularity of the two prefix algorithms is fundamental for a parameterized description in structural VHDL, as will be seen in the sequel.

3 Structural VHDL

The VHDL hardware description language allows the description of hardware at two levels of abstraction, the behavioral and the structural level. In order to generate a logic netlist, a behavioral description has to be translated into an RTL (register transfer level) description at the structural level. This mapping process is referred to as *VHDL synthesis*. Behavioral VHDL abstracts from the circuit's logic structure and allows the designer to concentrate on the circuit's behavior. Compared to structural hardware description, the behavioral level allows for much easier and more abstract description of complex circuits and systems, has advantages concerning code understandability, maintenance, and reuse, and is

property	BRENT & KUNG	SKLANSKY
max. fan-out	$\log n$	$\frac{n}{2}$
area	$2n - \log n - 2$	$\frac{1}{2} n \log n$
depth	$2(\log n - 1)$	$\log n$

Table 1: *Properties of* BRENT *and* KUNG*'s and* SKLANSKY*'s parallel-prefix addition algorithms.*

substantial for more efficient simulation. As a matter of fact, the whole design process gets accelerated.

In behavioral VHDL the function of a circuit is described, but not its structure. The structure is generated automatically through VHDL synthesis, and its quality depends on the used *synthesis tool*. For common structures like adders these synthesis tools usually include netlist generators for a set of possible architectures, e.g. for ripple-carry and a carry-lookahead adders. If the synthesis tool encounters an addition operation in the code to be synthesized, one of these generators is called. If the designer wants to include another circuit architecture at this point, a description in structural VHDL must be incorporated.

Structural VHDL allows the simple description of flat or hierarchical netlists. Additionally, common language constructs for conditions and repetition as well as generic parameters can be used for the implementation of netlists generators with some degree of flexibility. The main VHDL constructs used for structural circuit description are now presented. Examples are given in pseudo VHDL code, i.e. unimportant code details are not included.

3.1 Simple Logic Expressions

Simple logic expressions can be written in VHDL as *concurrent signal assignments*. Equation (6) is written as

```
p(i) <= a(i) xor b(i);
```

where `p`, `a` and `b` are arrays of signals and `p(i)` denotes the `i`-th component of the array `p`.

3.2 Repetition and Parameterization

Equation (6) requires signal assignments for an entire range of bits. This can be described using the *repetition generate statement*:

```
square_cells: for i in 0 to 15 generate
  p(i) <= a(i) xor b(i);
end generate square_cells;
```

In equation (5) two different cases have to be distinguished for calculation of g_0 and $g_{i \geq 0}$. In structural VHDL this can be done using a *conditional generate statement* (i.e. `if ... generate`) or a *conditional signal assignment*:

```
square_cells: for i in 0 to 15 generate
  g(i) <= a(0) and b(0) or
          a(0) and ci or
          b(0) and ci
    when i = 0
    else a(i) and b(i);
  p(i) <= a(i) xor b(i);
end generate square_cells;
```

If the range of `i` is known only at synthesis time we need to *parameterize* the repetition statement above. Suppose the range of `i` is parameterized with the *generic parameter* `n`:

```
...  generic (n : integer);  ...

square_cells: for i in 0 to n-1 generate
  ...
end generate square_cells;
```

3.3 Components and Instantiation

Hierarchical structuring and component reuse can be realized by packaging subcircuits into VHDL *entities* and using them through *component instantiations*. In our example the parameterized VHDL description of equations (5) and (6) can be summarized in an entity as follows:

```
entity ppgpgen is
  generic (n : integer);
  port (a,b : in std_logic_vector
                          (n-1 downto 0);
        ci : in std_logic;
        g,p : out std_logic_vector
                          (n-1 downto 0));
end ppgpgen;

architecture structural of ppgpgen is
begin
  square_cells: for i in 0 to n-1 generate
    g(i) <= a(0) and b(0) or
            a(0) and ci or
            b(0) and ci
      when i = 0
      else a(i) and b(i);
    p(i) <= a(i) xor b(i);
  end generate square_cells;
end structural;
```

The entity `ppgpgen` can now be instantiated within another architecture by declaration of the component's interface in the architecture header

```
component ppgpgen
  generic (n : integer);
  port (a,b : in std_logic_vector
```

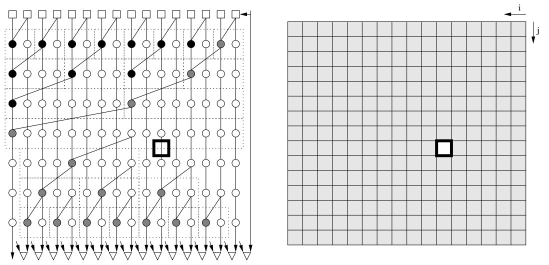


Figure 5: *Two-dimensional array of vectors* $\mathbf{v}_{i,j}$ *as basic data structure.*

```
                (n-1 downto 0);
         ci : in std_logic;
         g,p : out std_logic_vector
                 (n-1 downto 0));
end component;
for all : ppgpgen
  use entity ppgpgen(structural);
```

followed by the instantiation in the architecture body

```
square_cell_row : ppgpgen
  generic map (n);
  port map (a,b,ci,g,p);
```

For further details please refer to the literature on VHDL [8][9].

4 Implementation

In order to generate the logic for a parallel-prefix adder, its graph representation is implemented by mapping the graph nodes onto logic gates and the graph edges onto connecting wires. This can be achieved by visiting each cell and generate the corresponding logic and connections. From a programming point of view, this two-dimensional graph can be processed using two nested loops. The organization of these two loops or, in other words, the strategy for traversing the graph does not affect the resulting circuitry. On the other hand, it has an effect on the VHDL code structure implementing the traversing scheme, though in a rather subtle manner, as will be seen in the sequel.

4.1 Basic data structure

The basic data structure for a parallel-prefix adder description in structural VHDL is a two-dimensional array (matrix) of signal pairs (vectors $\mathbf{v}_{i,j}$) denoting the outputs of the cells in the graph representation (Fig. 5). In practice this array of vectors is replaced by two two-dimensional arrays for the signals $g_{i,j}$ and $p_{i,j}$, respectively. Thus, generating a parallel-prefix circuit can be regarded as interconnecting these signals with the appropriate logic. Again, this process does not depend on the order in which the cells of the graph are visited and their logic generated.

The VHDL synthesis tool used (Compass) did not allow the usage of two-dimensional arrays. However, an $n \times m$ two-dimensional array A can easily be mapped onto an $n \cdot m$ one-dimensional array B using a simple index calculation.

$$\begin{array}{rcl} A & \leftrightarrow & B \\ a(i,j) & \leftrightarrow & b(i + j \cdot n) \end{array} \tag{17}$$

Two different approaches for traversing the graph representation of parallel-prefix addition are now described. They also demonstrate the subtle influence of this underlying strategy on the code complexity of structural VHDL.

4.2 First Approach: Bit-Slice Technique

Because the netlists to be generated are parameterized with the number of operand bits n, the construction of an adder from n *bit-slices* was the most obvious approach. Thus, an adder is generated by one central loop:

```
bit_slice : for i in 0 to n-1 generate
  ...
end generate bit_slice;
```

Inside this loop the three stages of parallel-prefix addition described earlier are generated for one bit position. Therefore, the graph is traversed as illustrated in Fig. 6.

The generation of the logic for the pre- and the post-processing cells is simple and straightforward and does not change for different adder word lengths. Things get more complicated for generating the logic for the cells of the parallel-prefix stage. Basically, the cells and interconnections of the parallel-prefix stage are generated by a second loop which is nested in the top-level bit-slice loop and which processes the individual rows of the prefix graph. The corresponding pseudo code looks as follows:

```
bit_slice : for i in 0 to n-1 generate

  square_cell: ...

  prefix_nodes: for j in 1 to h generate
    ...
  end generate prefix_stage;

  triangle_cell: ...

end generate bit_slice;
```

The addition operand word length n does not only affect the width of the graph but also its depth. Thus,

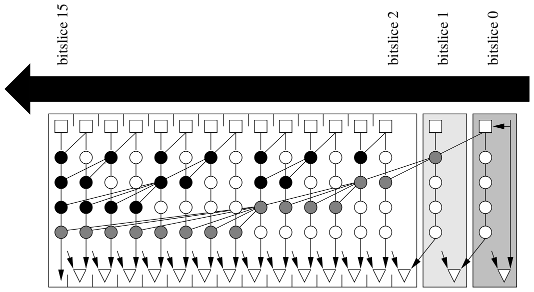


Figure 6: *Graph traversing scheme using the bit-slice technique.*

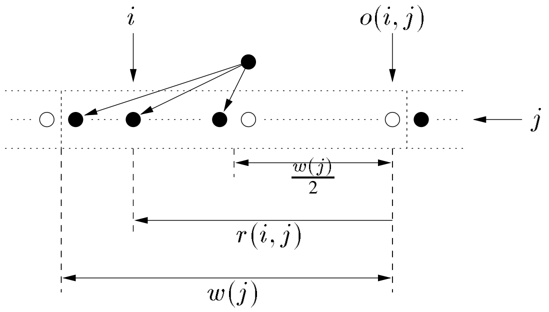


Figure 7: *Building block of* SKLANSKY*'s prefix algorithm.*

both nested loops depend on the adder length. Within the two loops, a decision has to be made whether a cell (i,j) is a white, a black, or a grey cell and what its interconnections are.

The required description of the parallel-prefix graph representation must be parameterizable with the given operand word length. The idea to obtain a simple and regular description is to divide the graph into building blocks, as depicted in the Figs. 3 and 4 by the dashed rectangles. These building blocks all have highly regular and similar structures and differ only in size, which can be captured by one simple parameterized description. SKLANSKY's prefix algorithm, for example, is built using one single building block while BRENT and KUNG's prefix algorithms uses two different ones. Based on these building blocks a scalable description for the two parallel-prefix adders has been implemented in structural VHDL, resulting in the desired *netlist generators*.

Some details of the generation process and the resulting VHDL code are now examined. Let us concentrate on colored and white cells, where colored cells are either black or grey ones. The SKLANSKY's prefix algorithm is chosen as example due to its very regular structure. Let i be the current bit number

$$0 \leq i < n$$

and j the current row in the parallel-prefix stage

$$1 \leq j \leq \lceil \log_2 n \rceil = h$$

where $\lceil x \rceil$ denotes the next higher natural number of x, if x is not natural itself. Let $\mathcal{M}$ be the set of all pairs (i,j) corresponding to a colored cell in the graph. The decision for a given pair (i,j) whether it corresponds to a white or a colored cell consists of several steps (see also Fig. 7). First, the length

$$w(j) = 2^j \tag{18}$$

of the building blocks in the current row j is calculated. Then the building block of row j is determined in which the i-th bit is located. Let the building blocks be numbered from right to left starting with 0. Then the building block containing bit i has number $b(i,j)$,

$$b(i,j) = \left\lfloor \frac{i}{w(j)} \right\rfloor \tag{19}$$

where $\lfloor x \rfloor$ denotes the next lower natural number of x, if x is not natural itself. Using the bit number

$$o(i,j) = b(i,j)w(j) \tag{20}$$

of the first bit of building block $b(i,j)$, the relative bit position

$$r(i,j) = i - o(i,j) \tag{21}$$

of bit i within this building block can be determined. Obviously the range of $r(i,j)$ is

$$0 \leq r(i,j) < w(j)$$

The relative bit number $r(i,j)$ specifies the type of cell to be generated. The set $\mathcal{M}$ of all pairs (i,j) corresponding to colored cells is

$$\begin{aligned} \mathcal{M} &= \left\{(i,j) : r(i,j) \geq \frac{w(j)}{2}\right\} \\ &= \left\{(i,j) : i - \left(\left\lfloor \frac{i}{2^j} \right\rfloor\right) 2^j \geq 2^{j-1}\right\} \end{aligned} \tag{22}$$

or in other words, all cells in the upper half of a building block are colored (Fig. 7).

Thus, generating the parallel-prefix logic for all pairs (i,j) bases on determining whether the current cell is an element of $\mathcal{M}$ and, if so, to generate the appropriate logic.

Additionally, the determination of the connections also needs calculation. Assume the cell corresponding to (i, j) is a colored cell. Then its two input nodes are its direct neighbor one row above $(i, j-1)$ and the node

$$\left(o(i,j) + \frac{w(j)}{2}, j-1\right) = \left(\left(\left\lfloor \frac{i}{2^j} \right\rfloor\right) 2^j + 2^{j-1} - 1, j-1\right) \quad (23)$$

as depicted in Fig. 7.

A white cell is only connected to its neighbor one row above $(i, j-1)$.

The following VHDL code results from implementation of the above equations:

```
bit_slice : for i in 0 to n-1 generate

  square_cell: ...

  prefix_nodes: for j in 1 to h generate

    g(j*n + i) <=
        g((j-1)*n + i) or
        g((j-1)*n + (i/2**j-1)*2**j
                  + 2**(j-1) - 1) and
        p((j-1)*n + i)
      when i-(i/2**j)-1)*2**j >= 2**(j-1)
      else
        g((j-1)*n + i);

    p(j*n + i) <=
        p((j-1)*n + i) and
        p((j-1)*n + (i/2**j-1)*2**j
                  + 2**(j-1) - 1)
      when i-(i/2**j)-1)*2**j >= 2**(j-1)
      else
        p((j-1)*n + i);

  end generate prefix_stage;

  triangle_cell: ...

end generate bit_slice;
```

The "/" operator denotes integer division in the above index calculations.

Unfortunately, it was not possible to structure the code any further by implementing the functions $w(j)$, $b(i,j)$, $o(i,j)$, and $r(i,j)$ separately, because the used synthesis tool does not allow any function calls in index calculations or condition expressions.

A VHDL netlist generator for the BRENT and KUNG prefix algorithm can be written in a very similar way with slightly different index calculations and condition expressions.

Two parameterized adders, one implementing SKLANSKY's and the other BRENT and KUNG's prefix algorithm were realized using the principles described so far.

The synthesis of the resulting code was very time and memory consuming using the synthesis tools by Synopsys Inc. Synthesis using the design tools by Compass

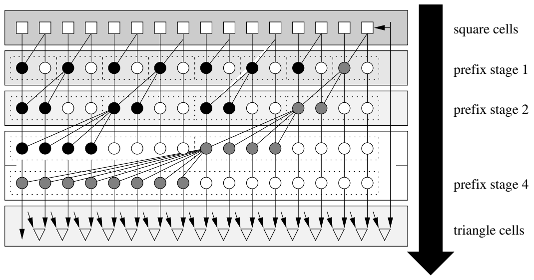


Figure 8: *Graph traversing scheme using the building-blocks technique.*

Design Automation was not successful at all, particularly because the synthesizer did not allow division operations in index calculations (equation (19)). Therefore, another approach was chosen which works without division and which turned out to be more efficient to synthesize or to be synthesizable at all, respectively.

4.3 Second Approach: Building-Blocks Technique

Because the bit-slice technique used in the first approach lead to unsatisfactory results, an alternative approach was chosen. Here, the array is not constructed column-wise from bit-slices, but row-wise from individual prefix stages. The prefix stages themselves are composed of appropriate *building blocks*. The outer loop now processes individual prefix stages.

```
generate square cells;

prefix_stage: for j in 1 to h generate
  ...
end generate stage;

generate triangle cells;
```

Thus, the graph is traversed as illustrated in Fig. 8. The generation of the square and triangle cells now has to be carried out in separate loops, as can be seen in the next code fragment.

A second (inner) loop is now used for visiting all bits within the current row.

```
generate square cells;

prefix_stage: for j in 1 to h generate

  bit: for i in 0 to n-1 generate
    ...
  end generate bit;

end generate stage;

generate triangle cells;
```

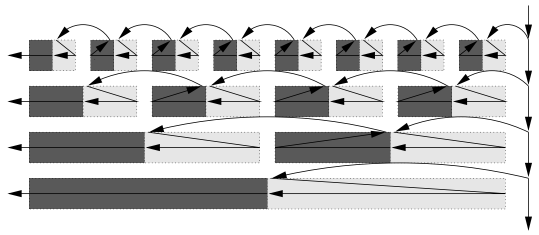


Figure 9: *Traversing scheme using three levels of nested loops.*

This solution, however, requires exactly the same decisions and index calculations that led to the mentioned synthesis problems in the first approach. The basic idea in our second approach is the separate processing of individual building blocks by a third loop. Instead of having only one loop per row requiring complex building block and bit position calculations, the second-level loop now visits all building blocks while two third-level loops process all white and black cells within a building block (Fig. 9). By choosing appropriate loop structures and boundaries, the index calculations within the loops become much simpler and require no division operations anymore. Put differently, the granularity of the generate-loops was increased in a way that the determination of the cell type and the connections for each individual bit position is straightforward.

Developing a VHDL netlist generator for SKLANSKY's prefix algorithm according to this loop structure is now quite simple. The first-level loop processes the rows of the parallel-prefix stage.

```
prefix_stage: for j in 1 to h generate
  ...
end generate stage;
```

The second-level loop processes the building blocks within a row,

```
  group: for gr in 0 to m(j) - 1 generate
    ...
  end generate group;
```

where

$$m(j) = 2^{h-j} \tag{24}$$

corresponds to the number of building blocks in stage j. Since all white cells appear in the first and all colored cells in the second half of a building block, two loops are used at the third level, one for the white and one for the colored cells.

```
    white_cells:
      for w in 0 to w(j)/2 - 1 generate
      ...
    end generate white_cells;

    colored_cells:
      for c in w(j)/2 to w(j) - 1 generate
      ...
    end generate colored_cells;
```

Here, $w(j)$ again denotes the building block size of stage j (Fig. 7).

The complete pseudo code now is:

```
generate square cells;

prefix_stage: for j in 1 to h generate

  group: for gr in 0 to 2**(h-j) - 1 generate

    white_cells:
      for w in 0 to 2**(j-1) - 1 generate
      ...
    end generate white_cells;

    colored_cells:
      for c in 2**(j-1) to 2**j - 1 generate
      ...
    end colored_cells;

  end generate group;

end generate stage;

generate triangle cells;
```

No conditional signal assignments are used anymore, since the white and the colored cells are generated in separate loops. Index calculations are simpler (no division operations) but involve three loop variables (`j`: prefix stage, `gr`: building block within stage, and `w` or `c`: cell within building block).

The elaborated generator code for the SKLANSKY parallel-prefix stage using the second approach looks as follows:

```
square_cells: ...

prefix_stage: for j in 1 to h generate

  group: for gr in 0 to 2**(h-j) - 1 generate

    white_cells:
      for w in 0 to 2**(j-1) - 1 generate

      white_cell: if gr*2**j + w < n generate
        g(j*n + gr*2**j + w) <=
          g((j-1)*n + gr*2**j + w);
        p(j*n + gr*2**j + w) <=
          p((j-1)*n + gr*2**j + w);
      end generate white_cell;

    end generate white_cells;

    colored_cells:
      for c in 2**(j-1) to 2**j - 1 generate

      colored_cell:
```

```
        if gr*2**j + c < n generate
        g(j*n + gr*2**j + c) <=
          g((j-1)*n + gr*2**j + c) or
          (p((j-1)*n + gr*2**j + c) and
          g((j-1)*n + gr*2**j + 2**(j-1) - 1);
        p(j*n + gr*2**j + c) <=
          p((j-1)*n + gr*2**j + c) and
          p((j-1)*n + gr*2**j + 2**(j-1) - 1);
      end generate colored_cell;

    end generate colored_cells;

  end generate group;

end generate prefix_stage

triangle_cells: ...
```

Here, the description of the adder again reflects the three stages of preprocessing, parallel-prefix calculation, and postprocessing.

The two conditions

```
white_cell: if gr*2**j + w < n generate

colored_cell: if gr*2**j + c < n generate
```

appearing in the code above cut the building blocks down to the word length n.

Again, a VHDL netlist generator for BRENT and KUNG's prefix algorithm can be written using the same principles resulting in a similar code. It is slightly more complex, however, because BRENT and KUNG's prefix algorithm requires two different types of building blocks.

The simpler index calculations and the absence of any division operations in this second approach resulted in a VHDL generator code which was synthesizable efficiently on both software platforms used.

4.4 Subtraction, Comparison, Addition Flags

Subtraction and generation of various addition flags are often required as well. The basic addition flags are:

V: occurrence of an overflow,

N: the sign of the result S (signed[1]),

C: ($= c_{out}$) A is lower than B (unsigned),

LT: A is less than B (signed),

Z: the result S is zero.

The adaption of a given adder structure for subtraction or for flag generation is quite simple and requires only little additional logic, with the exception of fast zero detection.

[1] All signed numbers in this context are 2's complement numbers. Note that unsigned and 2's complement numbers are treated exactly the same in binary addition.

4.4.1 Subtraction

Given an n-bit adder

$$(c_{out}, S) = add(A, B, c_{in}) \tag{25}$$

where A, B and S are n-bit binary numbers of either unsigned or 2's-complement form, an ordinary addition $A + B$ is calculated as

$$(c_{out}, S) = add(A, B, 0) \tag{26}$$

Subtraction $A - B$ is realized by inverting all bits of the operand B and adding a one (i.e. $c_{in} = 1$):

$$(c_{out}, S) = add(A, \overline{B}, 1) \tag{27}$$

An adder/subtractor can be realized by conditionally inverting all operand bits of B by an additional level of XOR-gates.

4.4.2 Overflow Flag V

For unsigned addition, an overflow can simply be detected by testing the carry-out c_{out}. For signed addition the overflow flag is computed as

$$V = c_{n-1} \oplus c_{n-2} \tag{28}$$

Since the implemented parallel-prefix adders generate the final carries for all bit positions (i.e. also c_{n-2}), the calculation of this flag requires only one additional XOR-gate.

4.4.3 Sign Flag N

A sign bit only makes sense if signed operands are used. In that case the sign flag is simply the MSB

$$N = s_{n-1} \tag{29}$$

of the result S.

4.4.4 'Less Than' Flag LT

The 'less that' (signed) and 'lower than' (unsigned) flags are only valid after performing a subtraction. The 'lower than' bit for unsigned operands is simply the carry-out c_{out}. The 'less than' flag for signed addition is

$$LT = N \oplus V \tag{30}$$

4.4.5 Zero Flag Z

Zero detection on the addition result S is very simple according to the following logic formula:

$$Z = \overline{s_0 + \cdots + s_{n-1}} \tag{31}$$

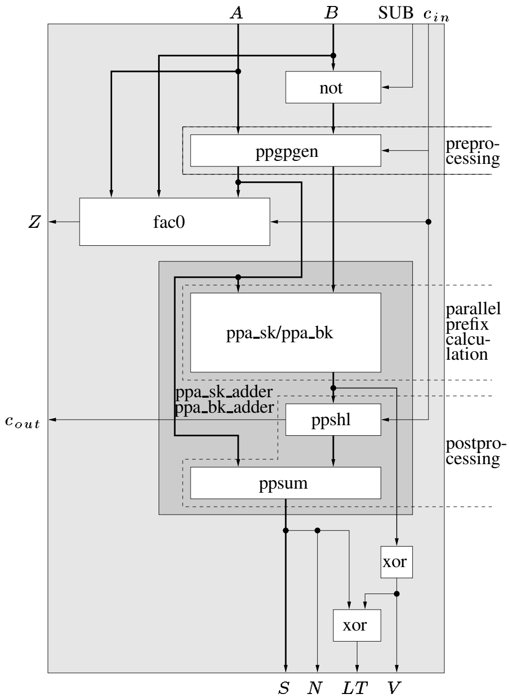


Figure 10: *Schematic of a universal adder/subtractor with flag generation.*

This computation, however, is rather slow since evaluation has to wait until the addition result is stable. Another approach does without carry-propagation and results in much faster zero flag generation [10]. In a first step, a zero flag z_i is generated for each bit position i by examining the bits i and $i-1$. These flags are then combined to the final zero flag Z. The underlying equations are

$$\begin{aligned} v_i &= a_i + b_i && (0 \leq i < n-1) \\ z_0 &= \neg\,(p_0 \oplus c_{in}) && \\ z_i &= \neg\left(v_{i-1} \oplus p_i\right) && (1 \leq i < n) \\ Z &= z_0 \cdots z_{n-1} && (32) \end{aligned}$$

4.5 Adder/Subtractor Generator

Putting everything together results in a netlist generator for the universal adder/subtractor with flag generation depicted in Fig. 10. As was demonstrated it is possible to realize the entire generator in structural VHDL. However, by adding more flexibility – such as selection of individual circuit features by the user – the realization becomes very circumstantial and the interface rather unfriendly if implemented entirely in VHDL. Another approach using the Perl script language [11] was used instead. The implemented script generates the top-level VHDL code with all the user-selected features. Two examples of code generated by this Perl script are found in Appendix B.

The VHDL code of the blocks depicted in the schematic of Fig. 10 are found in Appendix A. Note that the names used in the code are not consistent with the names used in the text.

5 Results

It was possible to implement netlist generators in structural VHDL for two high-performance adder structures in a parameterized fashion. Only the second approach using a more sophisticated graph traversing scheme resulted in efficiently synthesizable generator code. This leads us to the conclusion that, at the current status of synthesis tools, the parameterized structural description of arbitrary circuits in VHDL is not a priori possible.

Due to some fundamental limitations of today's synthesis tools as well as of the VHDL language itself, not the entire flexibility desired for realization of customized adder circuits can be implemented in structural VHDL efficiently. An additional implementation level had to be incorporated into the circuit generation process, instead. This step was realized using a Perl script which generates the top-level VHDL templates including the customized circuit interface and the user-selected adder features.

6 Experiences

One of the major goals of this work was the exploration of the possibilities for implementation of circuit generators in structural VHDL. From a theoretical point of view, no fundamental limitations exist in VHDL which would disallow the parameterized description of arbitrary circuits. In reality and under consideration of state-of-the-art synthesis tools (in our case primarily Compass, but also Synopsys), however, the following essential deficiencies were showing up:

- Function calls are not allowed in constant declarations and constant expressions of generate-statements (Compass AsicSynthesizer). As a consequence, the depth of the parallel-prefix stage (which is the logarithm of the word length) cannot be calculated within the VHDL code but has to be given at the instantiation through a generic parameter. This problem is not present in Synopsys.

- On one hand, arithmetic and logic operations are used to describe a circuit's behavior and thus have to be synthesized. On the other hand, these operations are also used in index calculations and control statements (i.e. condition and interval expressions of generate-statements), where the operations are evaluated once during synthesis and do not represent any logic to be synthesized. Apparently, these two possible occurrences of arithmetic/logic operations are not properly distinguished in today's synthesis tools. The usage of complex arithmetic operations in synthesis control statements leads to unacceptably high synthesis runtimes or, even worse, is restricted. As an example, Compass does not allow division operations within array index calculations and constant expressions of generate-statements, which is a severe but not mandatory limitation (no such restriction exists in Synopsys). The second implementation approach described in this report was chosen to circumvent this deficiency. Such a work-around, however, does not always exist.

- As a general observation, synthesis of parameterized structural VHDL code seems to be much less runtime efficient than synthesis of fixed code.

Additionally, the realization of flexible netlist generators is circumstantial if implemented fully in structural VHDL, even if the above limitations are neglected. From all these observations we can conclude that the most promising approach for implementing flexible arithmetic circuit generators is a two-level approach. In the first level a conventional programming language is used for generating fixed or weakly parameterized structural VHDL code. This code is then used as input to actual hardware synthesis in the second level. Note that this approach also allows the implementation of a sophisticated user interface for easy access of a comprehensive and flexible circuit components library.

7 Conclusions

Netlist generators for high-performance adders were realized using a combination of efficient and flexible Perl scripts and a set of synthesizable and parameterized structural VHDL code entities. Subtractors and adders with various addition flags are included as well.

Valuable experiences were made with respect to parameterized structural VHDL and the implementation of netlist generators. Based on the knowledge gained, the realization of a comprising netlist generator library for arithmetic components is planned for the near future.

References

[1] J. Sklansky, "An evaluation of several two-summand binary adders", *IRE Trans. Electron. Comput.*, vol. EC-9, no. 6, pp. 213–226, June 1960.

[2] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations", *IEEE Trans. Comput.*, vol. 22, no. 8, pp. 783–791, Aug. 1973.

[3] R. E. Ladner and M. J. Fischer, "Parallel prefix computation", *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[4] R. P. Brent and H. T. Kung, "A regular layout for parallel adders", *IEEE Trans. Comput.*, vol. 31, no. 3, pp. 260–264, Mar. 1982.

[5] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders", in *Proc. 8th Computer Arithmetic Symp.*, Como, May 1987, pp. 49–56.

[6] H. Lindkvist and P. Andersson, "Techniques for fast CMOS-based conditional sum adders", in *Proc. IEEE Int. Conf. Comput. Design: VLSI in Computers and Processors*, Cambridge, USA, Oct. 1994, pp. 626–635.

[7] J. Sklansky, "Conditional sum addition logic", *IRE Trans. Electron. Comput.*, vol. EC-9, no. 6, pp. 226–231, June 1960.

[8] IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*, 1987.

[9] Z. Navabi, *VHDL Analysis and Modeling of Digital Systems*, McGraw-Hill, New York, 1993.

[10] J. Cortadella and J. M. Llaberia, "Evaluation of A + B = K conditions without carry propagation", *IEEE Trans. Comput.*, vol. 41, no. 11, pp. 1484–1488, Nov. 1992.

[11] L. Wall and R. L. Schwartz, *Programming Perl*, O'Reilly & Associates, Sebastopol, CA, 1991.

A Listings

A.1 ppa_sk_adder

```
entity ppa_sk_adder is
  generic (n : integer;
           m : integer);
  port (G,P : in Std_Logic_Vector(n-1 downto 0);
        CI : in Std_Logic;
        S : out Std_Logic_Vector(n-1 downto 0);
        CO : out Std_Logic;
        C : out Std_Logic_Vector(n-1 downto 0));
end ppa_sk_adder;

------------------------------------

architecture ppa_sk_adder of ppa_sk_adder is

  component ppa_sk
    generic (n : integer;
             m : integer);
    port (G0,P0 : in  Std_Logic_Vector(n-1 downto 0);
          Gm : out Std_Logic_Vector(n-1 downto 0));
  end component;
  for all : ppa_sk
    use entity arithmetik.ppa_sk(ppa_sk);

  ----------------------------------

  component ppshl
    generic (n : integer);
    port (GI : in Std_Logic_Vector(n-1 downto 0);
          CI : in Std_Logic;
          GO : out Std_Logic_Vector(n-1 downto 0);
          COUT : out Std_Logic);
  end component;
  for all : ppshl
    use entity arithemtik.ppshl(ppshl);

  ----------------------------------

  component ppsum
    generic (n : integer);
    port (G,P : in Std_Logic_Vector(n-1 downto 0);
          S : out Std_Logic_Vector(n-1 downto 0));
  end component;
  for all : ppsum
    use entity arithmetik.ppsum(ppsum);

  ----------------------------------

  signal Gm,Gs : Std_Logic_Vector(n-1 downto 0);

begin

  sklansky : ppa_sk
    generic map (n,m)
    port map (G,P,Gm);
  C <= Gm;

  shl : ppshl
    generic map (n)
    port map (Gm,CI,Gs,CO);

  sum : ppsum
    generic map (n)
    port map (Gs,P,S);

end ppa_sk_adder;
```

A.2 ppa_bk_adder

```
entity ppa_bk_adder is
  generic (n : integer;
           m : integer);
  port (G,P : in Std_Logic_Vector(n-1 downto 0);
        CI : in Std_Logic;
        S : out Std_Logic_Vector(n-1 downto 0);
        CO : out Std_Logic;
        C : out Std_Logic_Vector(n-1 downto 0));
end ppa_bk_adder;

------------------------------------

architecture ppa_bk_adder of ppa_bk_adder is

  component ppa_bk
    generic (n : integer;
             m : integer);
    port (G0,P0 : in  Std_Logic_Vector(n-1 downto 0);
          Gm : out Std_Logic_Vector(n-1 downto 0));
  end component;
  for all : ppa_bk
    use entity arithmetik.ppa_bk(ppa_bk);

  ----------------------------------

  component ppshl
    generic (n : integer);
    port (GI : in Std_Logic_Vector(n-1 downto 0);
          CI : in Std_Logic;
          GO : out Std_Logic_Vector(n-1 downto 0);
          COUT : out Std_Logic);
  end component;
  for all : ppshl
    use entity arithemtik.ppshl(ppshl);

  ----------------------------------

  component ppsum
    generic (n : integer);
    port (G,P : in Std_Logic_Vector(n-1 downto 0);
          S : out Std_Logic_Vector(n-1 downto 0));
  end component;
  for all : ppsum
    use entity arithmetik.ppsum(ppsum);

  ----------------------------------

  signal Gm,Gs : Std_Logic_Vector(n-1 downto 0);

begin

  brent_kung : ppa_bk
    generic map (n,m)
    port map (G,P,Gm);

  C <= Gm;

  shl : ppshl
    generic map (n)
    port map (Gm,CI,Gs,CO);

  sum : ppsum
    generic map (n)
    port map (Gs,P,S);

end ppa_bk_adder;
```

A.3 ppgpgen

```
entity ppgpgen is
  generic (n : integer);
  port (A,B : in Std_Logic_Vector(n-1 downto 0);
        CI : in Std_Logic;
        G,P : out Std_Logic_Vector(n-1 downto 0));
end ppgpgen;

------------------------------------

architecture ppgpgen of ppgpgen is
begin
  square_cell0 :
    G(0) <= (CI and A(0)) or (CI and B(0)) or
       (A(0) and B(0));
    P(0) <= A(0) xor B(0);
```

```
  square_cells : for sc in 1 to n-1 generate
    G(sc) <= A(sc) and B(sc);
    P(sc) <= A(sc) xor B(sc);
  end generate square_cells;
end ppgpgen;
```

A.4 ppa_bk

```
entity ppa_bk is
  generic (n : integer;
           m : integer);
  port (G0,P0 : in  Std_Logic_Vector(n-1 downto 0);
        Gm : out Std_Logic_Vector(n-1 downto 0));
end ppa_bk;

-----------------------------------

architecture ppa_bk of ppa_bk is

  signal G,P : Std_Logic_Vector(0 to (2*m)*n - 1);

begin
  input: for i in 0 to n-1 generate
    P(i) <= P0(i);
    G(i) <= G0(i);
  end generate input;
  stage_scheme1: for st in 1 to m generate
    group: for gr in 0 to 2**(m-st) - 1 generate
      white_cells: for w in 0 to 2**st - 2 generate
        white_cell: if gr*2**st + w < n generate
          G(st*n + gr*2**st + w) <=
            G((st-1)*n + gr*2**st + w);
          P(st*n + gr*2**st + w) <=
            P((st-1)*n + gr*2**st + w);
        end generate white_cell;
      end generate white_cells;
      colored_cells:
        if gr*2**st + 2**st - 1 < n generate
        grey_or_black_cell:
          G(st*n + gr*2**st + 2**st - 1) <=
            G((st-1)*n + gr*2**st + 2**st - 1) or
            (P((st-1)*n + gr*2**st + 2**st - 1) and
            G((st-1)*n + gr*2**st + 2**(st-1) - 1));
        grey_cell: if gr = 0 generate
          P(st*n + 2**st - 1) <=
            P((st-1)*n + 2**st - 1);
        end generate grey_cell;
        black_cell: if gr > 0 generate
          P(st*n + gr*2**st + 2**st - 1) <=
            P((st-1)*n + gr*2**st + 2**st - 1) and
            P((st-1)*n + gr*2**st + 2**(st-1) - 1);
        end generate black_cell;
      end generate colored_cells;
    end generate group;
  end generate stage_scheme1;

  stage_scheme2: for st in m+1 to 2*m - 1 generate
    group: for gr in 0 to 2**(st - m) - 1 generate
      group0: if gr = 0 generate
        white_cells1:
          for w in 1 to 2**(2*m - st) - 1 generate
          white_cell: if w - 1 < n generate
            G(st*n + w - 1) <= G((st-1)*n + w - 1);
            P(st*n + w - 1) <= P((st-1)*n + w - 1);
          end generate white_cell;
        end generate white_cells2;
      end generate group0;
      other_groups: if gr > 0 generate
        white_cells2:
        for w in 0 to 2**(2*m - st - 1) - 1 generate
          white_cell2:
          if gr*2**(2*m - st) + w - 1 < n generate
            G(st*n + gr*2**(2*m - st) + w - 1) <=
              G((st-1)*n + gr*2**(2*m - st) + w - 1);
            P(st*n + gr*2**(2*m - st) + w - 1) <=
              P((st-1)*n + gr*2**(2*m - st) + w - 1);
          end generate white_cell2;
        end generate white_cells2;
        grey_cell:
          if gr*2**(2*m - st) +
            2**(2*m - st - 1) - 1 < n generate
          G(st*n + gr*2**(2*m - st)
              + 2**(2*m - st - 1) - 1) <=
            G((st-1)*n + gr*2**(2*m - st) +
              2**(2*m - st - 1) - 1) or
            (P((st-1)*n + gr*2**(2*m - st) +
              2**(2*m - st - 1) - 1) and
            G((st-1)*n + gr*2**(2*m - st) - 1));
          P(st*n + gr*2**(2*m - st) +
          2**(2*m - st - 1) - 1) <=
            P((st-1)*n + gr*2**(2*m - st) +
              2**(2*m - st - 1) - 1);
        end generate grey_cell;
        white_cells:
          for w in 2**(2*m - st - 1) + 1
            to 2**(2*m - st) generate
          white_cell:
            if gr*2**(2*m - st) + w - 1 < n generate
            G(st*n + gr*2**(2*m - st) + w - 1) <=
              G((st-1)*n + gr*2**(2*m - st) + w - 1);
            P(st*n + gr*2**(2*m - st) + w - 1) <=
              P((st-1)*n + gr*2**(2*m - st) + w - 1);
          end generate white_cell;
        end generate white_cells;
      end generate other_groups;
    end generate group;
    msb: if 2**m = n generate
      G(st*n + n - 1) <= G((st-1)*n + n - 1);
      P(st*n + n - 1) <= P((st-1)*n + n - 1);
    end generate msb;
  end generate stage_scheme2;
  output: for o in 0 to n-1 generate
    Gm(o) <= G(n*(2*m-1) + o);
  end generate output;
end ppa_bk;
```

A.5 ppa_sk

```
entity ppa_sk is
  generic (n : integer;
           m : integer);
  port (G0,P0 : in  Std_Logic_Vector(n-1 downto 0);
        Gm : out Std_Logic_Vector(n-1 downto 0));
end ppa_sk;

-----------------------------------

architecture ppa_sk of ppa_sk is

  signal G,P : Std_Logic_Vector(0 to (m+1)*n - 1);

begin
  input: for i in 0 to n-1 generate
    P(i) <= P0(i);
    G(i) <= G0(i);
  end generate input;
  stage: for st in 1 to m generate
    group: for gr in 0 to 2**(m-st) - 1 generate
      white_cells:
        for w in 0 to 2**(st-1) - 1 generate
        white_cell: if gr*2**st + w < n generate
          G(st*n + gr*2**st + w) <=
            G((st-1)*n + gr*2**st + w);
          P(st*n + gr*2**st + w) <=
            P((st-1)*n + gr*2**st + w);
        end generate white_cell;
      end generate white_cells;
      colored_cells:
        for c in 2**(st-1) to 2**st - 1 generate
        colored_cell: if gr*2**st + c < n generate
          grey_or_black_cell:
            G(st*n + gr*2**st + c) <=
              G((st-1)*n + gr*2**st + c) or
              (P((st-1)*n + gr*2**st + c) and
              G((st-1)*n + gr*2**st + 2**(st-1) - 1));
          grey_cell: if gr = 0 generate
            P(st*n + c) <= P((st-1)*n + c);
```

```
        end generate grey_cell;
        black_cell: if gr > 0 generate
          P(st*n + gr*2**st + c) <=
            P((st-1)*n + gr*2**st + c) and
            P((st-1)*n + gr*2**st + 2**(st-1) - 1);
        end generate black_cell;
      end generate colored_cell;
    end generate colored_cells;
  end generate group;
end generate stage;
output: for o in 0 to n-1 generate
  Gm(o) <= G(n*m + o);
end generate output;
end ppa_sk;
```

A.6 ppshl

```
entity ppshl is
  generic (n : integer);
  port (GI : in Std_Logic_Vector(n-1 downto 0);
        CI : in Std_Logic;
        GO : out Std_Logic_Vector(n-1 downto 0);
        COUT : out Std_Logic);
end ppshl;

------------------------------------

architecture ppshl of ppshl is
begin
  msb : COUT <= GI(n-1);
  shl : for s in n-1 downto 1 generate
    GO(s) <= GI(s-1);
  end generate shl;
  lsb : GO(0) <= CI;
end ppshl;
```

A.7 ppsum

```
entity ppsum is
  generic (n : integer);
  port (G,P : in Std_Logic_Vector(n-1 downto 0);
        S : out Std_Logic_Vector(n-1 downto 0));
end ppsum;

------------------------------------

architecture ppsum of ppsum is
begin
  sum : for su in n-1 downto 0 generate
    S(su) <= G(su) xor P(su);
  end generate sum;
end ppsum;
```

A.8 fac0

```
entity fac0 is
  generic (n : integer);
  port (A,B,P : in Std_Logic_Vector(n-1 downto 0);
        CI : in Std_Logic;
        E : out Std_Logic);
end fac0;

------------------------------------

architecture fac0 of fac0 is

    function ReduceAnd (INPUT: Std_Logic_Vector)
             return Std_Logic is
        variable RESULT : Std_Logic;
    begin
        -- evaluate logical function
        RESULT := '1';
        for J in INPUT'range loop
             RESULT := INPUT(J) and RESULT;
             exit when RESULT = '0';
        end loop;
        return RESULT;
    end;

  signal V,Z : Std_Logic_Vector(n-1 downto 0);

begin
  required_carry: for r in 0 to n-2 generate
    V(r) <= A(r) or B(r);
  end generate required_carry;
  zero0: Z(0) <= not (P(0) xor CI);
  zero: for ze in 1 to n-1 generate
    Z(ze) <= not (V(ze-1) xor P(ze));
  end generate ze;
  E <= ReduceAnd(Z);
end fac0;
```

B Examples

B.1 add_bk32_c

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library COMPASS_LIB;
use COMPASS_LIB.COMPASS.ALL;

-----------------------------

entity add_bk32_c is
  port(A,B : in Std_Logic_Vector(31 downto 0);
       CI : in Std_Logic;
       S : out Std_Logic_Vector(31 downto 0);
       CO : out Std_Logic);
end add_bk32_c;

-----------------------------

architecture add_bk32_c of add_bk32_c is

  component ppgpgen
    generic (n : integer);
    port (A,B : in Std_Logic_Vector(n-1 downto 0);
          CI : in Std_Logic;
          G,P : out Std_Logic_Vector(n-1 downto 0));
  end component;
  for all : ppgpgen
    use entity arithmetik.ppgpgen(ppgpgen);

  component ppa_bk_adder
    generic (n : integer;
             m : integer);
    port (G,P : in Std_Logic_Vector(n-1 downto 0);
          CI : in Std_Logic;
          S : out Std_Logic_Vector(n-1 downto 0);
          CO : out Std_Logic;
          C : out Std_Logic_Vector(n-1 downto 0));
  end component;
  for all : ppa_bk_adder
    use entity arithmetik.ppa_bk_adder(ppa_bk_adder);

  signal G,P,BB,SS,C : Std_Logic_Vector(31 downto 0);

begin

  BB <= B;

  preprocessing: ppgpgen
                 generic map(32)
                 port map(A,BB,CI,G,P);

  bk: ppa_bk_adder
      generic map (32,5)
      port map (G,P,CI,SS,CO,C);

  S <= SS;

end add_bk32_c;
```

B.2 addsub_sk8_cvznl

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library COMPASS_LIB;
use COMPASS_LIB.COMPASS.ALL;

-----------------------------

entity addsub_sk8_cvznl is
  port(A,B : in Std_Logic_Vector(7 downto 0);
       CI : in Std_Logic;
       SUB : in Std_Logic;
       S : out Std_Logic_Vector(7 downto 0);
       N : out Std_Logic;
       Z : out Std_Logic;
       V : out Std_Logic;
       LT : out Std_Logic;
       CO : out Std_Logic);
end addsub_sk8_cvznl;

-----------------------------

architecture addsub_sk8_cvznl of addsub_sk8_cvznl is

  component ppgpgen
    generic (n : integer);
    port (A,B : in Std_Logic_Vector(n-1 downto 0);
          CI : in Std_Logic;
          G,P : out Std_Logic_Vector(n-1 downto 0));
  end component;
  for all : ppgpgen
    use entity arithmetik.ppgpgen(ppgpgen);

  component ppa_sk_adder
    generic (n : integer;
             m : integer);
    port (G,P : in Std_Logic_Vector(n-1 downto 0);
          CI : in Std_Logic;
          S : out Std_Logic_Vector(n-1 downto 0);
          CO : out Std_Logic;
          C : out Std_Logic_Vector(n-1 downto 0));
  end component;
  for all : ppa_sk_adder
    use entity arithmetik.ppa_sk_adder(ppa_sk_adder);

  component fac0
    generic (n : integer);
    port (A,B,P : in Std_Logic_Vector(n-1 downto 0);
          CI : in Std_Logic;
          E : out Std_Logic);
  end component;
  for all : fac0
    use entity arithmetik.fac0(fac0);

  signal NN : Std_Logic;
  signal VV : Std_Logic;
  signal G,P,BB,SS,C : Std_Logic_Vector(7 downto 0);

begin

  process(B,SUB)
  begin
    if SUB = '0' then
      BB <= B;
    else
      BB <= not B;
    end if;
  end process;

  preprocessing: ppgpgen
                 generic map(8)
                 port map(A,BB,CI,G,P);

  sk: ppa_sk_adder
      generic map (8,3)
      port map (G,P,CI,SS,CO,C);

  zero: fac0
        generic map(8)
        port map (A,BB,P,CI,Z);

  S <= SS;

  VV <= C(7) xor C(7-1);

  V <= VV;

  NN <= SS(7);

  N <= NN;

  LT <= NN xor VV;

end addsub_sk8_cvznl;
```