



HAL
open science

Arithmetic core generation using bit heaps

Nicolas Brunie, Florent de Dinechin, Matei Istoan, Guillaume Sergent, Kinga Illyes, Bogdan Popa

► **To cite this version:**

Nicolas Brunie, Florent de Dinechin, Matei Istoan, Guillaume Sergent, Kinga Illyes, et al.. Arithmetic core generation using bit heaps. 23rd International Conference on Field Programmable Logic and Applications, Sep 2013, Porto, Portugal. pp.1-8. ensl-00738412v2

HAL Id: ensl-00738412

<https://ens-lyon.hal.science/ensl-00738412v2>

Submitted on 29 Nov 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ARITHMETIC CORE GENERATION USING BIT HEAPS

Nicolas Brunie, Florent de Dinechin, Matei Istoan, Guillaume Sergent,
Kalray LIP (ENS-Lyon/INRIA/CNRS/UCBL)

Kinga Illyes, Bogdan Popa
Universitatea Technica Cluj-Napoca

Abstract—A bit heap is a data structure that holds the unevaluated sum of an arbitrary number of bits, each weighted by some power of two. Most advanced arithmetic cores can be viewed as involving one or several bit heaps. We claim here that this point of view leads to better global optimization at the algebraic level, at the circuit level, and in terms of software engineering.

To demonstrate it, a generic software framework is introduced for the definition and optimization of bit heaps. This framework, targeting DSP-enabled FPGAs, is developed within the open-source FloPoCo arithmetic core generator. Its versatility is demonstrated on several examples: multipliers, complex multipliers, polynomials, and discrete cosine transform.

I. INTRODUCTION AND MOTIVATION

In binary digital arithmetic, a positive integer or fixed-point variable X is represented as follows:

$$X = \sum_{i=i_{\min}}^{i_{\max}} 2^i x_i \quad (1)$$

In all this paper we will call *weight* a power of two such as the 2^i in the above equation: X is represented as a sum of weighted bits. The indices i_{\min} and i_{\max} are the minimum and maximum weights of X .

The product of X by Y can similarly be expressed as a sum of weighted terms:

$$\begin{aligned} XY &= \left(\sum_{i=i_{\min}}^{i_{\max}} 2^i x_i \right) \times \left(\sum_{j=j_{\min}}^{j_{\max}} 2^j y_j \right) \\ &= \sum_{i,j} 2^{i+j} x_i y_j \end{aligned} \quad (2)$$

In all the sequel we use the term *bit heap* to denote a sum of weighted bits such as (2) above. There, each weighted bit on the heap is actually a term $x_i y_j$. However, most of this work will focus on the final summation: How each weighted bit was produced (e.g. as an AND of two input bits here) is mostly irrelevant – it will be addressed only when needed.

We will classically [1] represent bit heaps as 2D dot diagrams such as Fig. 1(c) and following. There, the horizontal axis represents the weights (most significant bits left as usual), and each dot represents one weighted bit participating to the final sum. Here also, how each bit was computed is not shown.

Bit heaps capture bit-level parallelism

Addition within a sum of weighted bits is associative and commutative. For instance, in a dot diagram, the order of the bits in a column (same weight) is irrelevant. Therefore, a representation like $\sum_{i,j} 2^{i+j} x_i y_j$ captures all the intrinsic bit-level parallelism present in this sum. Exploiting this, a fast multiplier may be built as a tree (often called a *compressor tree*) of bit-level adders [2], [3], [4], [5], [1]. More details will

be given in Section IV in the context of FPGAs. In general, the sum of a bit heap can be computed by an architecture operating in space linear with the size of the heap (its total number of bits), and time logarithmic in its maximum height (the maximum number of bits of same weight) [1].

Bit heaps are versatile

Bit heaps are not limited to computing multiplication. It is easy to show that the sum or product of a bit heap are themselves bit heaps (they can be developed as a single big sum). By induction, any multivariate polynomials of fixed-point inputs may be expressed as a single sum of weighted bits. This includes addition and multiplication on complex numbers, sums of products and sums of squares (for linear algebra operators or signal processing transforms), polynomials used to approximate elementary functions, etc. Besides, the bits may come from table lookup or other arbitrary components, which further enlarges the class of functions where a bit heap is relevant.

Expressing such a function as a monolithic bit heap (a single sum), rather than as a composition of adders and multipliers, enables a global optimization instead of several independent local ones. This has been shown to lead to more efficient bit-level implementations [6], [7], [8]. However, there has been so far no attempt to capture this generality in a universal tool. This is the main goal of the present work.

Due to this versatility, this article will demonstrate bit heaps with a variety of shapes. This is why we prefer the phrase *bit heap* over the phrase “bit array” used in the multiplier literature (where bit heaps are indeed lozenges). It also emphasizes that the order is irrelevant in the sum.

Bit heaps enable bit-level algebraic optimizations

The bit heap is also a pertinent tool to assess and improve the bit-level complexity of computing such multivariate polynomials. An enlightening example (to our knowledge unpublished) is a third-order Taylor formula for the sine: $\sin(X) \approx X - X^3/6$, which we use in [9].

This formula can be evaluated using two standard multiplications (to compute X^3), a multiplication by the constant $1/6$, and a subtraction. However, it can also be expressed directly as a single bit heap as follows. From (1), we may write

$$\begin{aligned} X^3 &= \sum_{i=i_{\min}}^{i_{\max}} 2^{3i} x_i^3 \\ &\quad + \sum_{i_{\min} \leq i < j \leq i_{\max}} 3 \cdot 2^{i+2j} x_i x_j^2 \\ &\quad + \sum_{i_{\min} \leq i < j < k \leq i_{\max}} 6 \cdot 2^{i+j+k} x_i x_j x_k \end{aligned} \quad (3)$$

Here we have a first set of algebraic simplifications to apply, for instance $x_i^k = x_i$ or $2 \cdot 2^w a = 2^{w+1} a$. The multiplication by

3 can be obtained by duplicating bits in the bit heap: $3 \cdot 2^w a = 2^{w+1} a + 2^w a$.

At this point the computation of X^3 already requires about one third of the bit-level operations that would be present in two multipliers. However, as we are interested in $X^3/6$, we may optimize further and rewrite (3) as:

$$X - X^3/6 = \sum_{i=i_{\min}}^{i_{\max}} 2^i x_i - \frac{1}{3} \sum_{i=i_{\min}}^{i_{\max}} 2^{3i-1} x_i - \sum_{i_{\min} \leq i < j \leq i_{\max}} 2^{i+2j-1} x_i x_j - \sum_{i_{\min} \leq i < j < k \leq i_{\max}} 2^{i+j+k} x_i x_j x_k \quad (4)$$

Now we only have only $i_{\max} - i_{\min}$ bits of the bit heap to actually divide by 3. We target some output accuracy, typically matching the input precision $p = i_{\max} - i_{\min} + 1$. One option is to replace $1/3$ with its binary representation $0.1010101\dots$, suitably truncated. Then the second line adds about as many bits to the bit heap as the third one. We use another option, with a variation of the divider by 3 from [10] that adds $3p$ bits to the bit heap, albeit after a delay.

Table I shows properly truncated bit heaps in the context of a sine/cosine implementation [9], and gives the corresponding synthesis results (for smaller input sizes, the whole of $X^3/6$ is simply tabulated).

A versatile bit heap implementation simplifies core generation

Finally, bit heaps are convenient from a software development point of view. The purpose of the FloPoCo project¹ is to generate an infinite set of application-specific operators for FPGA-based computing [11]. The long-term development of such a core generator is challenging for several reasons. Firstly, the operators we want to develop are increasingly complex (currently including filters, elementary functions, etc). Secondly, FloPoCo has the ambition of optimizing all its operators for a wide range of FPGAs from the main vendors. The sheer combinatorics of operators (each heavily parameterized) and targets make this task less and less tractable.

However, most of the advanced operators include products, sums of several terms read from tables or computed as products, etc. A solution to the above issue is therefore to express as much as possible of the operators as bit heaps. We may then delegate their implementation (the actual code generation) to a

¹<http://flopoco.gforge.inria.fr/>




input width	Bitheap	Performance cycles @ frequency	LUTs	Registers
32		6 @ 365 MHz	3044	981
		4 @ 265 MHz	3047	387
40		6 @ 351 MHz	4767	1257
		4 @ 254 MHz	4734	437
48		6 @ 326 MHz	7355	1841
		4 @ 249 MHz	7352	635

TABLE I

$X - X^3/6$ ON VIRTEX-5 FOR THE FAITHFUL SINCOS OPERATOR OF [9]

single, centralized, versatile bit heap framework. When a new FPGA appears, all we need is to update this one piece of code.

This can be viewed as an elegant way to separate 1/ optimizations that are of mathematical nature (the construction of the bit heap), and 2/ optimizations that depend on the target technology and target performance (its compression).

The remainder of this article introduces such a universal bit heap manipulation tool. As a bit heap is a data structure, not an arithmetic operator in the usual sense, we embed this notion in the FloPoCo architecture generator. To obtain an architecture, one first throws bits (now VHDL signals) from various components on a bit heap, then calls a routine that will build the compressor tree.

Outline

As FPGAs offer more and more DSP computing resources, we first need to include these resources in a bit-heap-centric view in Section II. This will actually expose additional optimizations opportunities.

Section III describes in details the data structures used for bit heap manipulation, with a focus on *timing* issues – in coarser operators such as DSP-based multipliers or our $X - X^3/6$ example, bits may arrive on the bit heap from various sources and at different times. Another focus is on *signed* bit heaps handling two’s complement numbers: we generalize classical tricks to show that signed numbers entail very low overhead.

Section IV studies the construction of the compression tree. It shows that the best elementary compressors are based on ternary adders on FPGA architectures that support them, such as the recent Altera circuits.

The interested reader will find in the FloPoCo source code (starting with version 2.4.0) all the details omitted here due to space constraints.

II. BIT HEAPS IN DSP-ENABLED FPGAS

One initial motivation of this work was to improve the implementation of large multipliers using DSP blocks in FloPoCo. This section first reviews the state of the art, points to sub-optimal design choices, and shows how a DSP-aware bit heap view can solve them.

A. FPGA multiplication capabilities

The logic fabric of FPGAs is based on small look-up tables (LUTs). A k -inputs look-up table (LUT k) may implement any boolean function of up to k inputs ($k = 4, 5$, or 6 in current FPGAs). For instance, an AND of up to k bits consumes one LUT k , so each bit of the bit heap for a degree- k polynomial will cost one LUT to compute. Tightly coupled carry propagation logic enables fast carry-ripple additions at the cost of one LUT per addition bit. This logic is designed so that array multipliers can be built at the cost of n^2 LUTs for an $n \times n$ -bit multiplier.

In recent years, the number k of inputs to the LUT has increased from 4 to 6. A 3x3-bit multiplication may now be

implemented by tabulating it in 6 LUT6, instead of accumulating it in 9 LUT6. This is not only smaller, but also as fast as it gets (one LUT delay).

Back to the bit heap, this is also an efficient way of generating partial products for a logic-based multiplier. We are not aware of this idea in the literature. The smaller squares on figures such as Fig. 2 represent such 3x3 LUT-based multipliers.

Recent FPGAs also include small “hard” multipliers. Altera chips offer 36x36-bit multipliers fracturable into a variety of combinations of 18x18, 12x12 and 9x9 ones, and various relevant sum-of-product configurations. Recent Xilinx chips offer 18x25 signed multipliers followed by 40-bit adders. Starting with Virtex-6, there are also adders on the inputs. Exploiting this variety is the current main challenge of portable arithmetic design in FloPoCo.

B. Building large multipliers as bit heaps

A large multiplication must be decomposed in smaller ones that fit hardware resources. This decomposition is a sum: it can be managed as a bit heap. To illustrate this, consider the following 4-DSP implementation of a 41x41 multiplier on Virtex-5 [12].

$$\begin{aligned}
 XY = & (X_{0:16}Y_{0:23} + 2^{17}X_{17:40}Y_{0:16}) \\
 & + 2^{23}(X_{0:23}Y_{24:40} + 2^{17}X_{24:40}Y_{17:40}) \quad (5) \\
 & + 2^{34}X_{17:23}Y_{17:23}
 \end{aligned}$$

In [12], the process of discovering (5) is called tiling. A multiplier board must be tiled with tiles corresponding to hard multipliers. Smaller, leftover tiles may be implemented as logic, as shown on on Fig. 1.

Furthermore, the adders included in DSP blocks may be used to sum the results of several multipliers. A sequence of multipliers chained by adders without needing any LUT logic is called a *supertile* in [12]. For instance, the two parentheses of (5) correspond to two supertiles for Xilinx Virtex 5 or later. Altera devices have square multipliers and different supertiling capabilities, but the same concepts apply – see [13] for recent examples.

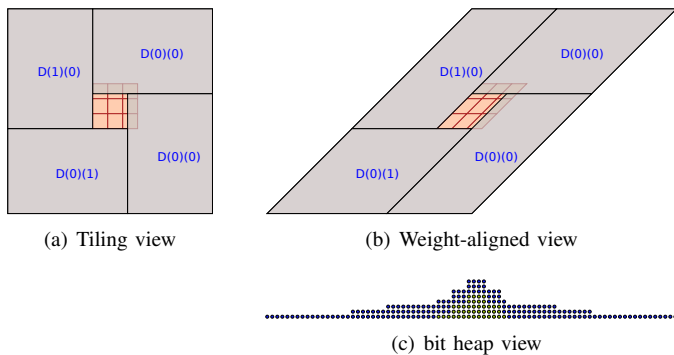


Fig. 1. A possible implementation of a 41×41 multiplier on Virtex-5

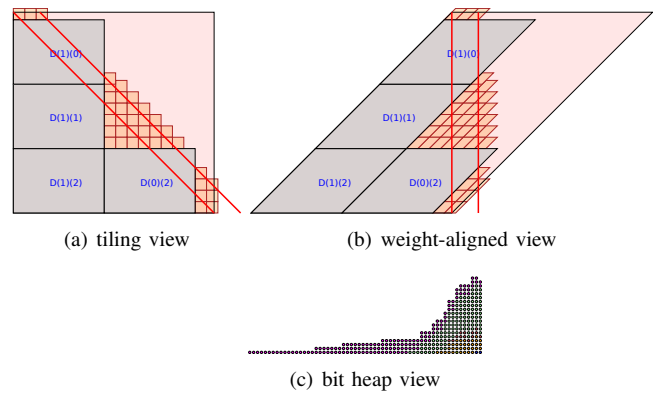


Fig. 2. 53x53-bit multiplier faithful to 53-bit, for Virtex5. The bits between the two red lines are guard bits needed for a faithful result (error strictly smaller than the LSB of the result).

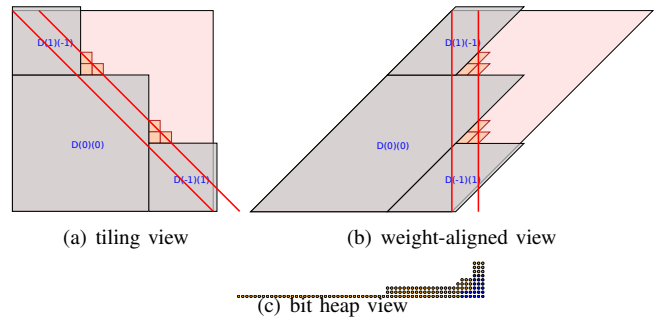


Fig. 3. 53x53-bit faithful multiplier for Stratix IV.

Fig. 1 is generated by FloPoCo. Its bit heap (Fig. 1(c)) receives the results of two supertiles, plus the bits of the central logic-based multiplier decomposed into 3×3 multipliers.

This tiling approach is very versatile. To illustrate it, Fig. 2 shows the tiling of a 53x53-bit truncated multiplier outputting a faithful result on 53 bits. On this figure, we have one potential supertile of size 3, and one of size 1. Fig. 3, inspired by [13], shows the same multiplier implemented for a Stratix IV device. It consists of one 36x36 multiplier, and two 18x18 ones that form a supertile. Also note that the tiling algorithm has a threshold parameter t (set to 0.5 for all our figures). It defines the percentage of multiplier that must be useful to the large multiplication for a DSP block to be used. Set to zero, only logic will be generated. Set to 1, only DSP blocks will be used.

C. Bit heap versus components

In [12], a large multiplier is not based on a bit heap, but implemented as a combination of three types of components: DSP multipliers, logic multipliers, and multi-input adders. This entails several inefficiencies in a pipelined design:

1/ artificial synchronization of the bits of intermediate sums, whereas the lower bits could typically be forwarded earlier than the upper bits;

2/ several instance of bit heap compression (one for each logic-based multiplier, plus one for the final multi-addition);

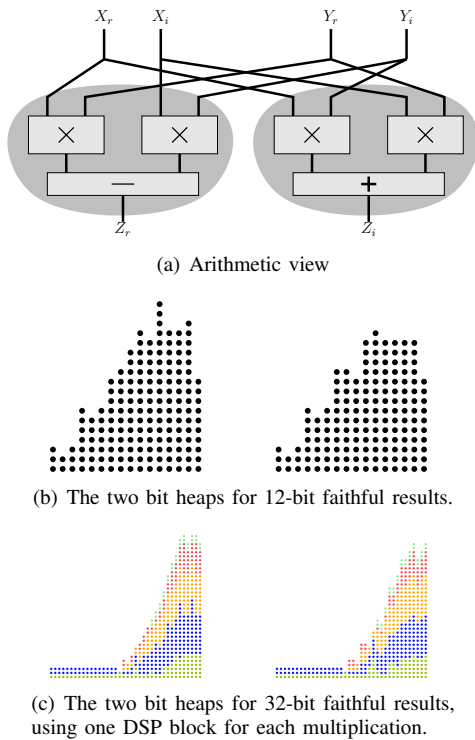


Fig. 4. Complex multiplication as two bit heaps. Different colors in the bit heap indicate bits arriving at different instants.

3/ non-utilization of some of the DSP adders (those on the first DSP of a supertile);

4/ in general, a combination of local optimizations instead of a global optimization.

These inefficiencies are aggravated if the multiplier is itself part of a larger component that could be expressed as a single bit heap. An important case is a sum of products, the simplest example being the complex product depicted on Fig. 4. In this case, additional optimizations opportunities arise.

The first is that supertiles can be built out of tiles coming from different multipliers, as long as they contribute to the same bit heap. Fig. 4 consists of two bit heap, each adding two products. Altera DSP blocks are designed to implement such operations ($a \times b \pm c \times d$) efficiently for precisions of 18 bits. Our key observation is that even for larger sizes (where each multiplier is decomposed in several DSP blocks), we can pair in a DSP corresponding sub-products from the two multipliers in a supertile computing $a \times b \pm c \times d$.

The second optimization is to exploit the adders within the DSP blocks to consume bits from the bit heap (without consideration of where they come from). In the simple multiplier examples, the first DSP adder of a supertile is unused. If we can feed it bits from another source (e.g. bits from the logic-based multiplier, possibly already partly compressed), it will remove them from the bit heap for free. Xilinx DSP adders can input such an external addend. Note that Altera ones cannot, unless otherwise unused multipliers are used as pass-through.

D. Summing up

The good way to manage DSP blocks in a bit heap is therefore to

- consider DSP blocks as unevaluated multiplications in a bit heap, and
- consider supertiling as a step of the compression process.

With this point of view, a multiplier contributes to a bit heap, but may not be alone in doing so. It may contribute bits from logic-based tiles, or DSP-sized tiles. The chaining of such tiles into a supertile is part of the compression process, and it may even consume other bits of the bit heap. The result of a supertile will be itself sent back to the bit heap for further compression.

In the proposed framework, a multiplier is not necessarily a component itself (an entity in VHDL terms). In the complex multiplier, it is a virtual component that throws bits to the bit heap of a larger component. In terms of architecture generation, the (simplified) code of a sum of product will be

```
addBitsToBitHeapForMult(...);
addBitsToBitHeapForMult(...);
compressBitHeap();
```

An adder or subtractor is not necessarily a component, either. The adder and subtractor of Fig. 4 have no corresponding VHDL entities, they are part of the compression.

Conversely, an operator may involve several distinct bit heaps (two on Fig. 4). This will be the general case.

The complex multiplier example also shows that we must be able to add the opposite of a product to the bit heap. An efficient implementation of this, exploiting the fact that DSP blocks include signed multipliers, will be shown in Section III-C.

III. THE UNIVERSAL BIT HEAP

Let us first stress that the bit heap is a dynamical data structure during the circuit generation process: First, several sub-operators may add bits to the bit heap. Eventually, in the construction of the compressor tree, bits will be removed from the bit heap, and new bits will be added, until only the final sum remains.

There are two aspects of time here: operator-generation time (as the generator program runs), and circuit time (using notions such as critical path or, for pipelined designs, cycles). The generator must be able to evaluate the circuit time, for instance as in [11].

A. The data structure

A weighted bit is a data structure consisting essentially of its signal name, its weight, and its instant. A column of the bit heap is represented as a list of weighted bits, ordered by their arrival time (in the circuit time). The complete bit heap data structure essentially consists of a maximum weight, and an array of columns indexed by the weights.

Figures 1(c), 2(c), 3(c), 4(b) or 4(c) represent such data structures, with the weights as horizontal axis, least significant bits to the right, as in standard binary numeration. Different

arrival times are indicated by different colors. These figures are actually Scalable Vector Graphics files that can be opened in a browser, in which case hovering the mouse over one bit shows its signal name and its arrival instant in circuit time.

B. Managing constant weighted bits

Some of the bits added to a bit heap are constant bits, e.g. coming from

- rounding truncated products added to the bit heap,
- rounding the final result of the bit heap,
- managing two's complement signed numbers, as explained below in III-C.

It would be wasteful to dedicate hardware to compressing such constant bits, as their sum is known in advance. Therefore, the bit heap data structure also includes a multiple-precision integer which gathers all the constant bits. The actual value of the sum of all the constant bits is added to the bit heap just before compression.

This is an old trick, widely used in the design of Baugh-Wooley signed multipliers [14] and multi-input adders [1]. However the addition of the constant bits was usually computed by hand, which for complex bit heaps soon becomes error-prone. Here it is performed by the generator, which is simpler, safer and more flexible.

As a side effect, note that fixed-point rounding computations can often be merged in the global compression for free.

C. Managing signed numbers

Signed numbers are classically represented using two's complement notation [1]. It is possible to design bit heaps operating directly on positive and negative bits [15], but this section shows that managing two's complement signed numbers in a bit heap costs very little, as most of the overhead can be hidden in the constant bit vector.

1) *Sign extension using the constant vector:* Most numbers added to the bit heap do not extend to its full range of weights. When such a number is signed, it must be sign-extended: its MSB must be replicated all the way up to the MSB of the bit heap [1]. Done naively, this could add many bits to the bit heap, some of them with large fanout. We may use here a classical trick from two's complement multipliers: to replicate bit s from weight p to weight q , we add the complement of s at weight p . Then we add $2^q - 2^p$ to the constant bit vector (this correspond to a string of 1s stretching from bit p to bit q). The reader may check that this performs the necessary sign extension both when $s = 0$ and $s = 1$.

All these constant bits are compressed into the constant vector: The overhead of a bit heap accepting signed numbers, with respect to an unsigned bit heap, is at most one line of bits.

Relatedly, subtracting a number to the bit heap resumes to adding the bitwise complement (with sign extension), and a constant 1 to the LSB of the subtrahend.

2) *Adding or subtracting a product of signed numbers:* Suppose we now have a bit heap whose result will be a two's complement numbers, and we want to either add or subtract the product XY , with X and Y being two's complement numbers on p and q bits respectively:

$$\begin{aligned} X &= -2^{p-1}x_{p-1} + \sum_{i=0}^{p-2} 2^i x_i \\ Y &= -2^{q-1}y_{q-1} + \sum_{j=0}^{q-2} 2^j y_j \end{aligned}$$

The product will be written

$$\begin{aligned} XY &= 2^{p+q-2}x_{p-1}y_{q-1} \\ &\quad -2^{p-1}x_p \sum_{j=0}^{q-2} 2^j y_j \\ &\quad -2^{q-1}y_q \sum_{i=0}^{p-2} 2^i x_i \\ &\quad + (\sum_{i=0}^{p-2} 2^i x_i) \times (\sum_{j=0}^{q-2} 2^j y_j) \end{aligned} \quad (6)$$

This product is a $p+q$ -bit number with the sign bit at position $p+q-1$. This weight doesn't appear in the above equation but appears as a carry out of the sum.

Adding this product to a bit heap can be performed by adding separately these four addends. The last line is a standard product of unsigned numbers, and can be added to the bit heap as is. The same holds for the first line – weight $p+q-2$ is not the sign bit. The two intermediate lines must be subtracted from the bit heap, which can be done by complementation and sign extension as described above.

Subtracting a product from the bit heap (as needed on the real side of the complex multiplier) is similar, and is “left as an exercise to the reader”. There is one trap: the negated product of two unsigned terms is not always negative, it may be zero, which has a positive sign.

We now want to exploit DSP blocks which are able to compute signed multiplications. A generalization of the previous technique consists in decomposing a p -bit input $X = -2^{p-1}x_{p-1} + \sum_{i=0}^{p-2} 2^i x_i$ as $X = 2^{p-k}X_h + X_l$ where

- X_h is formed of the k leading bits (including the sign bit), and may be considered as a k -bit signed number,
- X_l is formed of the $p-k$ least significant bits, and is an unsigned number.

We may similarly split $Y = 2^{q-k'}Y_h + Y_l$. Here k and k' must be understood as the size of a signed product: $(k, k') = (18, 25)$ on Xilinx, $(k, k') = (18, 18)$ on Altera. The product XY now becomes

$$\begin{aligned} XY &= 2^{p+q-k-k'}X_hY_h && \text{(signed} \times \text{signed)} \\ &\quad + 2^{p-k}X_hY_l && \text{(signed} \times \text{unsigned)} \\ &\quad + 2^{q-k'}X_lY_h && \text{(unsigned} \times \text{signed)} \\ &\quad + X_lY_l && \text{(unsigned} \times \text{unsigned)} \end{aligned}$$

With this rewriting we may now use, for the lines involving a signed number, signed multipliers offered by DSP blocks. Adding $-XY$ to a bit heap is similar.

IV. TIMING-DRIVEN BIT HEAP COMPRESSION

This section deliberately focuses on the various tradeoffs involved, as the algorithmic details may be found in the (still evolving) FloPoCo source code. It will be illustrated by Fig. 5, which describes the timing-driven compression of the bit heap of Fig. 2.

A. Basics

Bit heap compression is based on a set of *elementary compressors*. An elementary compressor inputs a few bits from the bit heap, and replaces them with their sum, on fewer bits. The simplest example is the full adder: It computes the sum of three bits of weight w , and rewrites this sum as two bits of weights $w + 1$ and w . It is also called (3:2) compressor. On Fig. 5, we have highlighted two (6:3) compressors that rewrite the sum of 6 bits as a 3-bit number.

The compression of a bit heap therefore consists of several *stages*. In each stage, the bit heap is paved with as many compressors as possible that compress it in parallel (we have highlighted only two of them on Fig. 5). The outputs of these compressors, plus possibly bits that have not been compressed in previous stages, plus possibly new bits arrived on the bit heap in between, form a new bit heap ready for the next stage.

There are therefore two broad issues to address: the choice of an elementary compressor library, and the algorithm that assembles them to form a full compression tree.

On FPGAs, the state of the art, concerning the first issue, is [8] (also see references therein). Their main contribution is to show that large elementary compressors can be built efficiently using the fast-carry logic.

With respect to the second issue, there are many technological parameters to take into account (starting with the compressor library and the performance of its various elements). In additions, there are several possible objective function: one may want to optimize for delay, or for area for instance. Most suggested solutions are somehow greedy: try to exploit the most efficient elementary compressors first, try to compress as much as possible in one stage. The algorithm in [8] works along these lines, as does the one we have implemented.

B. Elementary compressors for FPGAs

We use the notation of [1] and [8] to describe elementary compressors called *generalized parallel counters* in [8]. The full adder is denoted GPC(3:2). A GPC($k_1, k_0 : r$) inputs k_0 bits of weight 0 and k_1 bits of weight 1 and rewrites their sum on r bits. This notation can be generalized to more input columns but there is a diminishing return, and [8] only considers two columns.

Table II analyses the theoretical costs of some elementary compressors, including the best from [8], on an FPGA model that matches modern FPGAs from both Xilinx and Altera: the elementary cell is a LUT6 which may be used as two independent LUT5, combined with some dedicated fast carry propagation. More accurate, low-level synthesis results for both Altera and Xilinx may be found in [8]. In this table, we report

- the compression factor $\gamma = \frac{\text{number of input bits}}{\text{number of output bits}}$. A higher γ means fewer stages.
- the number of bits removed from the heap by a compressor. For instance the GPC(3:2) removes only one bit, while the GPC(6:3) removes 3 bits;
- the cost in LUT6 of the compressor. For instance, the GPC(3:2) uses a LUT6 as two independent LUT3, one for

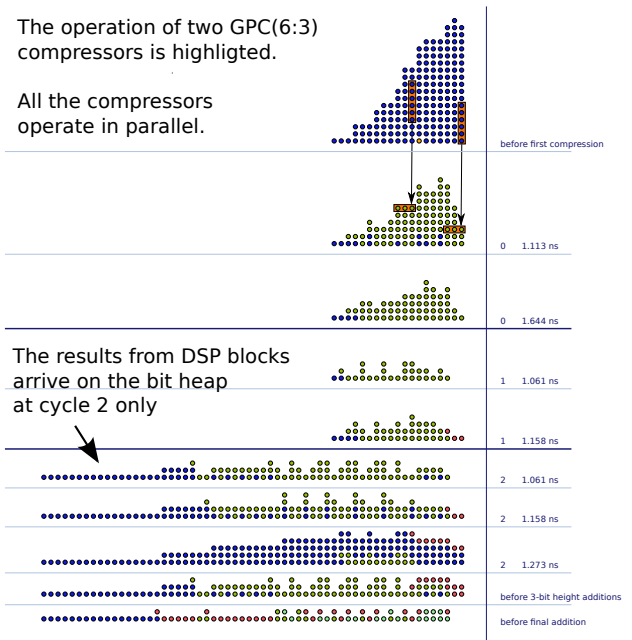


Fig. 5. Timed view of the compression of the bit heap of Fig. 2 for 400MHz. Horizontal lines separate compression stages, bold lines separate clock cycles. On the right, the numbers indicate the clock cycle and the delay within a cycle.

name	γ	δ_{bits}	LUT6	$\delta_{bits}/LUT6$	delay
Naive LUT-based compressors					
GPC(3:2)	1.5	1	1	1	τ_L
GPC(6:3)	2	3	3	1	τ_L
GPC(1,5:3)	2	3	3	1	τ_L
Arithmetic-based compressors from [8]					
GPC(1,5:3)	2	3	2	1.5	$2\tau_L + 2\tau_{cp}$
GPC(7:3)	2.33	4	3	1.33	$3\tau_L + 2\tau_{cp}$
GPC(3,5:4)	2	4	3	1.33	$3\tau_L + 2\tau_{cp}$
Adders as compressors					
adder(n)	2	n	n	1	$\tau_L + n\tau_{cp}$
adder3(n)	$3\frac{n}{n+2}$	$2n - 2$	$n + 2$	$\frac{2n-2}{n+2}$	$\tau_L + n\tau_{cp}$
adder3(16)	2.66	30	18	1.66	$1.5\tau_L$

TABLE II
THEORETICAL COMPARISON OF SEVERAL ELEMENTARY COMPRESSORS.
FOR THE ACTUAL COSTS ON STRATIX III AND VIRTEX5, SEE [8].

each of its output bits: it costs one LUT6. The GPC(6:3) needs one LUT6 for each of its output bits;

- the cost per removed bit, which is the relevant measure of area-efficiency of a compressor;
- the delays, expressed in terms of τ_L , the combined delay of a LUT and the local routing surrounding it, and τ_{cp} , the delay of a carry propagation of one bit. It should be noted that $\tau_{cp} \ll \tau_L$, typically $\tau_L \approx 30\tau_{cp}$. For instance, the 3 LUT6 of a GPC(6:3) operate in parallel, and thus have a delay of $1\tau_L$. Conversely, the arithmetic compressors from [8] involve a chain of two or three LUT6 connected by the global routing.

The main use of this table is to help us decide which is the best compressor at each stage of the compression process. Area-wise, we want to use the compressors with the highest

	cost of partial products	cost of compression (parallel counters) (ternary adders)	
24x24	384	505	283
32x32 trunc.	510	379	287

TABLE III
ALM COSTS FOR LOGIC-BASED MULTIPLIERS (PARALLEL COUNTER
VERSUS TERNARY ADDERS) ON ALTERA STRATIX IV

ratio of bits removed per LUT6. Delay-wise, we want to minimize the number of stages (higher γ) and the delay of one stage (lower delay). For instance, for the same bit/LUT ratio, it is preferable to use GPC(6:3) than GPC(3:2), as it will lead to fewer stages.

Our contribution is to observe that the ternary adder, when supported (e.g. on recent Altera FPGAs) is a serious contender. It has, for sizes between 5 and 30, the best ratio of compressed bits per LUTs, the best compression factor γ , and a delay between τ_L and $2\tau_L$ thanks to $\tau_L \approx 30\tau_{cp}$. This is consistently better than the arithmetic compressors of [8]. Table III shows that the area gain over parallel counters may be close to 50% for the same delay.

This is not in contradiction with the main conclusion of [8], which was that GPCs are more efficient than ternary adder trees: they consider the implementation of a bit heap compressor as a single tree of large ternary adders. There, many of the inputs to these adders must be padded with zeroes, which reduces the area efficiency. What we advocate here is the use of small ternary adders, whose size is chosen to pave the bit heap efficiently (without padding) while keeping the delay small.

A final advantage of the ternary adder is that it easy to express in a portable way ($a+b+c$ in Verilog or VHDL).

With all these considerations, our current implementation mostly ignores [8], and considers only the naive LUT6-based elementary compressors GPC(3:2), GPC(6:3), GPC(1,5:3), and small binary and ternary adders, the latter only on recent Altera targets.

C. Supertiling (DSP compression)

To compress a bit heap including multiplier tiles (possibly from several multipliers as described in Section II), the first step is to attempt to chain these into supertiles, using DSP adders.

Some of the bits produced along a supertile are reversed to the bit heap, tagged with the instant at which they are available. For instance, when we use two chained DSPs to compute $(a \times b) + 2^s(c \times d)$ where s is a constant shift (17 on Xilinx, 18 on Altera), the lower bits of the sum are those of $a \times b$ and are sent to the bit heap, while only the upper bits are sent to the DSP adder.

Conversely, in the beginning of a supertile, we look for bits available in the bit heap at the instant when the first DSP multiplication finishes. These bits will be registered and added for free using the adder of the first DSP of the supertile.

The exact supertiling algorithm depends on the target. On

Xilinx, the DSP blocks are independent and can be chained only using 17-bit shifts.

On Altera, the DSP block granularity is larger: a unit actually includes four 18x18 multipliers and an adder tree that has several possible configurations, the flexibility being limited by the number of physical output bit of the unit (around 72 bits). One useful configuration is the 36x36 multiplier, and we generate this as a single tile (see Fig. 3(a)). Another is the “complex multiplier” configuration, with two parallel sums of (independent) 18x18 products. The supertiling opportunities for the 9x9 and 12x12 configurations are unclear at this point.

To sum up, this step of the compression (in generation time) adds bits to the bit heap, and may remove some, too. We are then left with a classical (but timed) bit heap.

D. Bit-level compression

After supertiling, we have a main loop that advances a global (circuit) time. At each iteration, we look for sets of bits ready for compression in the bit heap. We tile such bits with the best possible compressors (more on their choice below). Applying a compressor means generating the corresponding hardware and removing its inputs bits from the bit heap. Output bits are sent to the bit heap where they arrive at a later circuit time (the max of the times of the inputs, plus the delay of the compressor). In a pipelined compressor, this may entail a change of cycle, or not. The output bits of one stage are not allowed to be processed in the same stage, to prevent the construction of carry propagation in one stage.

There is a heuristic decision to take to apply sub-optimal (area-wise) compressors. In the beginning of the loop, we prefer not to do it, hoping that the later arrival of more bits will enable the use of the most efficient compressors. In the last stages, however, sub-optimal compressors should be used, otherwise the delay will increase. A good strategy, inspired by Dadda [3], is to first evaluate the optimum delay, and then use the fewest possible sub-optimal compressors to reach this optimal delay.

The loop is stopped when no column holds more than 3 bits. There are several options for the final addition of these bits, depending on the target hardware and target frequency:

- it may use a ripple-carry ternary adder. This is the best option if the target FPGA supports it, and if the bit heap width is small enough for this addition to fit in one cycle.
- otherwise we may use one row of GPC(3:2) compressors, then any high-speed adder from [16].

In both case, this final addition is started as early as possible: if the final addition of the LSB bits can overlap the compression of the MSBs, we do it.

E. Results

Table I showed results for a truncated $X - X^3/6$, Table IV shows some synthesis results for the complex product, and Table V shows results for LUT-based discrete cosine transforms. These examples stress all the aspects of the framework. Performance does not yet always match expectations, but the complex product demonstrates the cost saving of a single bit

Approach	Performance	LUTs	Registers	DSPs
precision = 12 bits, logic-only multiplications				
Mult+Add	3 cycles @ 279 MHz	762	60	0
Bitheap	2 cycles @ 284 MHz	693	67	0
precision = 16 bits				
Mult+Add	2 cycles @ 254 MHz	1364	80	0
Bitheap	3 cycles @ 330 MHz	110	100	4
Bitheap	2 cycles @ 265 MHz	1264	150	0
precision = 24 bits				
Mult+Add	4 cycles @ 251 MHz	469	611	8
Bitheap	4 cycles @ 300 MHz	317	297	8
Bitheap	3 cycles @ 251 MHz	590	407	4
precision = 32 bits				
Mult+Add	4 cycles @ 250 MHz	768	1176	12
Bitheap	6 cycles @ 251 MHz	558	644	12
Bitheap	5 cycles @ 257 MHz	778	712	8
Bitheap	3 cycles @ 223 MHz	1778	608	4
precision = 64 bits				
Mult+Add	4 cycles @ 250 MHz	2114	2615	36
Bitheap	6 cycles @ 250 MHz	1153	1406	36
Bitheap	6 cycles @ 250 MHz	1532	1557	28
Bitheap	5 cycles @ 227 MHz	3356	1159	20

TABLE IV
FAITHFUL COMPLEX PRODUCTS ON VIRTEX-5

Taps	Performance	LUTs	Registers
precision = 16 bits			
8	3 cycles @ 250 MHz	1019	86
16	2 cycles @ 204 MHz	1999	108
32	3 cycles @ 177 MHz	4234	150
precision = 32 bits			
8	3 cycles @ 198 MHz	2996	180
16	4 cycles @ 172 MHz	5845	383
32	3 cycles @ 171 MHz	11801	463

TABLE V
FAITHFUL COSINE TRANSFORMS ON VIRTEX-5

heap over the instantiation of several adder and multiplier components, all other things being equal (in particular each multiplier using the compression code of the fused bit heap).

All these operators are faithful, i.e. their error with respect to an infinitely accurate computation is strictly smaller than the LSB of the result. This focus on accuracy is a distinct feature of FloPoCo, and the bit heap approach also makes the needed error analysis easy, but space is missing to detail this.

V. CONCLUSION

This article suggests to place the notion of bit heap at the center of arithmetic design, especially for coarse operators. The benefit is to expose, at different levels (from algebraic to circuit), a global optimization instead of several local optimizations. A versatile tool for manipulating and implementing such bit heaps is discussed. It also makes for shorter, more robust and better organized code in the open-source FloPoCo arithmetic generator.

This approach is directed toward arithmetic designers. An alternative approach is to use bit heaps in high-level design tools: Compilers optimizations that transform arithmetic circuits to maximize the benefit of carry-save notation [7] could

be used to build bit heaps automatically, in which case the proposed framework would serve as a back-end.

Short-term future work includes refinements to this bit heap framework, in particular the supertiling and compression heuristics. This will improve the absolute performance of the generated architectures. We are also working on relevant interfaces to the various trade-offs involved.

A promising direction is to work on the approximation of a function directly by a bit heap. In this context, we intend to automate algebraic optimizations such as those developed for $X - X^3/6$. There are also many opportunities to pre-compress, in small tables [17] that would fit LUTs, sums of terms sharing the same inputs.

Looking further, bit heaps could be a pertinent tool to address open bit-level complexity questions such as: *How many bits does one need to flip to compute a faithful 16-bit sine function?*

REFERENCES

- [1] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.
- [2] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, pp. 14–17, 1964.
- [3] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.
- [4] V. Oklobdzija, D. Villeger, and S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294–306, 1996.
- [5] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi, "Optimal circuits for parallel multipliers," *IEEE Transactions on Computers*, vol. 47, no. 3, pp. 273–285, 1998.
- [6] E. E. Swartzlander, "Merged arithmetic," *IEEE Transactions on Computers*, vol. C-29, no. 10, pp. 946–950, 1980.
- [7] A. K. Verma, P. Brisk, and P. Ienne, "Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1761–1774, 2008.
- [8] H. Parendeh-Afshar, A. Neogy, P. Brisk, and P. Ienne, "Compressor tree synthesis on commercial high-performance FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, 2011.
- [9] F. de Dinechin, M. Istoan, and G. Sergent, "Fixed-point trigonometric functions on FPGAs," in *Highly-Efficient Accelerators and Reconfigurable Technologies*, Mar. 2013.
- [10] F. de Dinechin and L.-S. Didier, "Table-based division by small integer constants," in *Applied Reconfigurable Computing*, Hong Kong, Mar. 2012, pp. 53–63.
- [11] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, July 2011.
- [12] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, "Multipliers for floating-point double precision and beyond on FPGAs," *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 73–79, 2010.
- [13] B. Pasca, "Correctly rounded floating-point division for DSP-enabled FPGAs," in *Field Programmable Logic and Applications*, 2012.
- [14] C. Baugh and B. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Transactions on Computers*, vol. C-22, no. 12, pp. 1045–1047, Dec. 1973.
- [15] G. Jaberipur, B. Parhami, and M. Ghodsi, "An efficient universal addition scheme for all hybrid-redundant representations with weighted bit-set encoding," *Journal of VLSI Signal Processing*, vol. 42, pp. 149–158, 2006.
- [16] H. D. Nguyen, B. Pasca, and T. Preusser, "FPGA-specific arithmetic optimizations of short-latency adders," in *Field Programmable Logic and Applications*, 2011, pp. 232–237.
- [17] H. Hassler and N. Takagi, "Function evaluation by table look-up and addition," in *12th Symposium on Computer Arithmetic*. Bath, UK: IEEE, 1995, pp. 10–16.