Arithmetic around the bit heap

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

Arithmetic around the bit heap

Florent de Dinechin, Matei Istoan, Guillaume Sergent
LIP (ENS-Lyon/Inria/CNRS/UCBL)
École Normale Supérieure de Lyon
France

Kinga Illyes, Bogdan Popa
Universitatea Tehnica din Cluj-Napoca
Romania

Nicolas Brunie
Kalray

***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. Any multivariate polynomial of binary inputs can be expressed as a bit heap whose bits are simple boolean functions of the input bits. For many large arithmetic designs, viewing them as bit heaps is more relevant than viewing them as a composition of adders and multipliers. It leads to better global optimization at both the algebraic level and the circuit level. However, this notion needs to be supported by tools. This article therefore discusses a generic software framework for the definition, optimization and compression of bit heaps. It is specifically directed towards FPGAs, where complex and application-specific arithmetic circuits must be developped in little time.**

For this purpose, the textbook notion of a bit array is refined in several ways. Firstly, a bit heap should accept bits arriving at various instants in circuit time, and the bit heap compression process must take this timing into account. Secondly, the DSP blocks of recent FPGAs must be integrated in the bit heap view. Thirdly, the management of signed bit heaps is detailed, and shown to entail no overhead. Finally, a new family of elementary compressors on FPGAs improves upon the state of the art.

***Index Terms*—hardware generation; sums of weighted bits; multipliers; operator fusion; FPGA arithmetic;**

I. Introduction and motivation

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

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

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 bits:

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

In all the sequel we use the term *bit heap* to denote a sum of weighted bits. In the multiplier case, each bit is a term $x_i y_j$ that can be evaluated as the AND of two of the input bits. We will classically represent bit heaps as 2D dot diagrams with the weights on the horizontal axis. Figures 1(c) and following are examples of such representations.

A. A bit heap captures bit-level parallelism

The main advantage of this representation is that it captures all the bit-level intrinsinc parallelism present in the multiplication of two binary numbers. Within this bit-level sum, addition is associative, therefore a fast multiplier may be built as a tree (often called a *compressor tree*) of bit-level adders, themselves composed of very few elementary gates [1], [2], [3], [4], [5]. This technique will be detailed in Section IV. In a nutshell, 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).

Bit heaps are not limited to computing multiplication: the sum and product of a bit heap are themselves bit heaps, so this representation enables us to express any multivariate polynomials of fixed-point inputs (possibly signed numbers using two's complement representation, as will be detailed in Section III-C).

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. By expressing all these functions as bit heaps, it is possible to obtain more efficient bit-level implementations than a composition of adders and multipliers [6], [7], [8].

B. A bit heap enables bit-level algebraic optimizations on arithmetic expressions

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: $\sin(X) \approx X - X^3/6$. 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. From (1), we may rewrite X^3 as

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

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 bit-level operations that would be present in two multipliers. It should also be noted that it requires less than half the latency, since the bit heap approach involves only one compression process, where two multipliers would require two compressions in sequence.

Still, as we are interested in $X^3/6$, we may optimize further and rewrite (2) as:

$$\begin{array}{rcl} X - X^3/6 & = & \sum_{i=i_{\min}}^{i_{\max}} 2^i x_i \\ & & -1/3 \sum_{i=i_{\min}}^{i_{\max}} 2^{3i-1} x_i \\ & & - \sum_{i_{\min} \leq i < j \leq i_{\max}} \cdot 2^{i+2j-1} x_i x_j \\ & & - \sum_{i_{\min} \leq i < j < k \leq i_{\max}} \cdot 2^{i+j+k} x_i x_j x_k \end{array} \quad (3)$$

Now we only have only $i_{\max} - i_{\min}$ bits of the bit heap to actually divide by 3. We are interested by some output precision (typically matching the input precision $p = i_{\max} - i_{\min} + 1$. By replacing this $1/3$ with its binary representation $0.1010101...$, suitably truncated, the second line adds about as many bits to the bit heap as the third one. The number of terms is asymptotically dominated by the last line, but it grows in $n^3/36$ [9]. For typical signal-processing precisions ($n < 32$), it remains smaller than one multiplication.

We conjecture that this can be generalized: high-order terms in polynomials will typically have much lower bit-level complexity in a bit heap than in a naive multiplier based implementation. This is still work in progress: firstly, a proper comparison must be made with Horner scheme which minimizes the arithmetic cost (but has high latency due to a succession of bit heap compressions). Secondly, we need to consider the cost of extra guard bits absorbing the truncation error.

C. An arithmetic generation framework should be centered around a versatile bit heap manipulation tool

Finally, bit heaps are convenient from a software development point of view. With a single, flexible enough implementation of this data structure (including compression methods), we are able to implement efficiently a wide range of operators. The remainder of this article presents such a universal bit heap manipulation tool.

A bit heap is not an arithmetic operator in the usual sense: What is needed is an architecture generator centered on the notion of a bit heap. To obtain an architecture, one first throws bits from various components on the bit heap, then calls a routine that will build the compressor tree. In a bit heap view, we pile bits from various sources before summing them: this enables a global optimization instead of several independent local ones. This article indeed demonstrates bit heaps with a variety of shapes, which is why we prefer the phrase *bit heap* over the phrase "bit array" often used in the literature. It also emphasizes that the order is irrelevant in the sum..

The construction of application-specific operators is especially relevant to FPGA computing. A contribution of this work is to integrate their embedded multipliers and DSP blocks in the bit heap view in Section II.

A second contribution is timing-driven compressor tree construction. The literature mostly focuses on compressor trees for multipliers, where all the partial products $x_i y_j$ can be computed in parallel, so the only timing to manage is that of the compressor tree: it is based on bit-level adders that consume bits from the bit heap and add their sum back to the bit heap at a later time [3], [4]. In coarser operators, however, even the initial bits may arrive on the bit heap from various sources, hence at various instants: some may come directly from the inputs, some from AND terms, some may come from table reads with some latency, some may come from small multipliers with different latency, and the opportunity to chain FPGA DSP blocks efficiently entails that bits will even arrive at various cycles. The construction of the summation tree should be directed by these timing considerations.

With these considerations, section III describes in details the data-structures used for bit heap manipulation. A special focus is on the handling of two's complement numbers: we generalize classical tricks to show that signed numbers entail very low overhead. Section IV presents the construction of the summation tree. A third contribution is to show that the best elementary compressors are based on ternary adders on FPGA architectures that support it, such as the recent Altera circuits.

II. MULTIPLICATION AND BIT HEAPS IN FPGAS

Bit heaps have been invented to implement fast multipliers, and one initial motivation of this work was to improve the implementation of large multipliers on FPGAs.

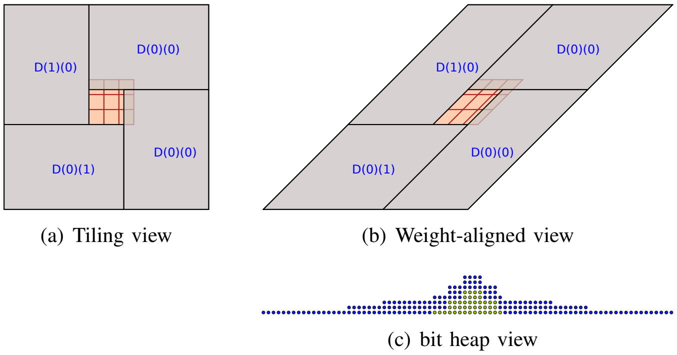
A. FPGA multiplication capabilities

The logic fabric of FPGAs has always been based on small look-up tables (LUTs) tightly coupled to carry propagation logic. A look-up table with k inputs (noted LUTk) may implement any boolean function of up to k inputs (k ranges from 4 to 6 in current FPGAs). For instance, an AND of up to k bits consumes one LUTk, so each bit of the bit heap for a degree-k polynomial will cost one LUT to compute. The carry propagation logic enables fast carry-ripple additions at the cost of one LUT per addition bit. This logic was also always designed to ensure that array multipliers could 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 (to reduce the needs for programmable routing). 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 the most 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 Figure 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, while recent Xilinx chips offer 18x25 signed multipliers.



(a) Tiling view

(b) Weight-aligned view

(c) bit heap view

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

B. Building large multipliers as bit heaps

A large multiplication must be decomposed in smaller ones that fit these blocks, and 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 inspired from [10], depicted on Figure 1.

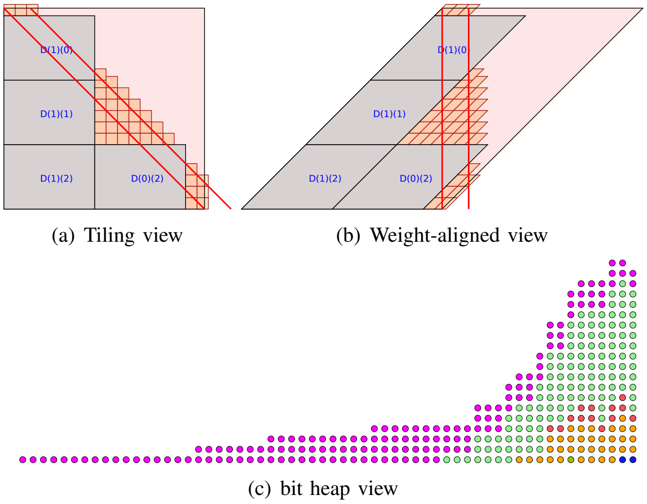
$$\begin{aligned} XY = \quad & (X_{0:16}Y_{0:23} \ + \ 2^{17}X_{17:40}Y_{0:16}) \\ & + 2^{23}(X_{0:23}Y_{24:40} \ + \ 2^{17}X_{24:40}Y_{17:40}) \\ & + 2^{34}X_{17:23}Y_{17:23} \end{aligned} \tag{4}$$

In [10], the process of discovering (4) is called tiling: a multiplier board must be tiled with the tiles corresponding to hard multiplier. Smaller, leftover tiles may be implemented as logic.

Furthermore, each hard multiplier is actually a multiply-accumulator (also usually termed DSP block because of its relevance to Digital Signal Processing applications). The accumulator's adder may be used to sum the results of several multipliers, possibly with a constant shift. A sequence of multipliers chained by adders without needing any LUT logic is called a *supertile* in [10]. For instance, the two parentheses of (4) 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 [11] for examples.

Figure 1 is generated by our tool. Its bit heap (Figure 1(c)) receives the results of supertiles, while the central logic-based tile is itself decomposed into 3 × 3 multipliers.

This tiling approach is very versatile. To illustrate it, Figure 2 shows the tiling of a 53x53-bit truncated multiplier outputting a faithful result on 53 bits. Its bit heap actually computes 8 additional weights to ensure that the sum of the truncated bits entail an error smaller than the LSB of the result. On this figure, we have one potential supertile of size 3, and one of size 1. Figure 3, inspired by [11], 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.

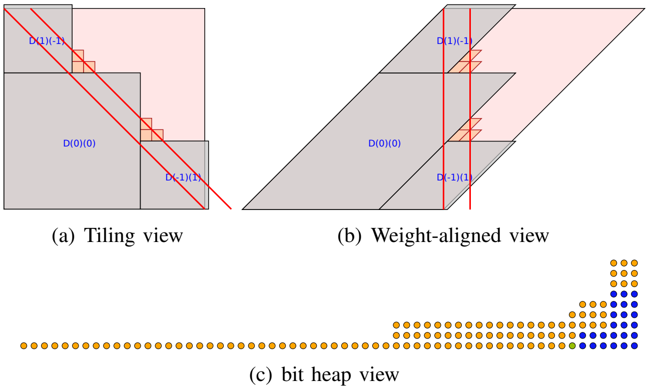


(a) Tiling view

(b) Weight-aligned view

(c) bit heap view

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. Different colors in the bit heap indicate bits arriving at different instants.



(a) Tiling view

(b) Weight-aligned view

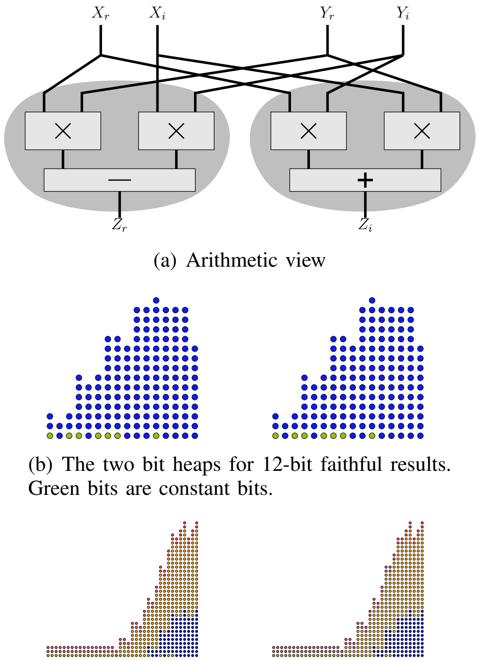
(c) bit heap view

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

C. Bit heap versus components

In [10], 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, among which:

- artificial synchronization of the bits of intermediate sums, whereas the lower bits could typically be forwarded earlier than the upper bits;
- several instance of bit heap compression (one for each logic-based multiplier, plus one for the final multi-addition);
- non-utilization of some of the DSP adders (those on the first DSP of a supertile)
- in general, a combination of local optimizations instead of a global optimization.



(a) Arithmetic view

(b) The two bit heaps for 12-bit faithful results. Green bits are constant bits.

(c) The two bit heaps for 32-bit faithful results, using one DSP block for each multiplication

Fig. 4. Complex multiplication as two bit heaps

These inefficiencies are aggravated if the multiplier is itself part of a larger component that could be a single bit heap such as the complex product of Figure 4.

D. Several multipliers contributing to a bit heap

There are additional optimizations opportunities for bit heaps spanning more than one multiplier.

The first consists in considering the supertiling process globally, at the level of the bit heap. Supertiles can be built out of tiles coming from different multipliers, as long as they contribute to the same bit heap. The simplest example is the complex product depicted on Figure 4. It consists of two bit heap, each adding two products. Of course, DSP blocks are designed to implement such operations efficiently for a precision of 18 bits, so this approach is mostly relevant for larger (or much smaller) precisions. This opportunity arises in any sum-of-product.

The second 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 an external addend. Altera ones cannot, unless otherwise unused multipliers are used as pass-through.

E. 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.

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 Figure 4 have no corresponding VHDL entities, they are part of the compression process.

Conversely, an operator may involve several distinct bit heaps (two on Figure 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). For timing-oriented bit heap compression, the generator must be able to evaluate the circuit time, for instance as in [12].

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.

Figures 1(c), 2(c), 3(c), 4(b) or 4(c) represent these data structures. The different arrival times are indicated by the 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

Most of the bits added to a bit heap are the result of some computation (e.g. the partial products in the multiplier), but very often we need to add 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 [13] and multi-input adders [5]. 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 computation, often presented as a separate, final step to the computation, can be merged in the global compression.

C. Managing signed numbers

Signed numbers are classically represented using two's complement notation [5]. It is possible to design bit heaps operating directly on positive and negative bits [14], 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 [5]. Done naively, this could add many bits to the bit heap, some of them with large fanout. At first sight, these bits are not constant, but we may use here a classical trick from two's complement multipliers: to replicate bit s from weight p to weight q, we just add $2^q - 2^p$ to the constant bit vector (this correspond to a string of 1s stretching from bit p to bit q), and only one variable bit, the complement of s, at weight p. The reader may check that this performs the necessary sign extension both when $s = 0$ and $s = 1$.

As 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 horizontal 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:

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

The product will be written

$$\begin{aligned} XY = \quad & 2^{p+q-2}x_{p-1}y_{q-1} \\ & -2^{p-1}x_p \textstyle\sum_{j=0}^{q-2} 2^j y_i \\ & -2^{q-1}y_q \textstyle\sum_{i=0}^{p-2} 2^i x_i \\ & +(\textstyle\sum_{i=0}^{p-2} 2^i x_i) \times (\textstyle\sum_{j=0}^{q-2} 2^j y_j) \end{aligned} \tag{5}$$

Note that 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 as described above. This involves a sign extension up to position $p+q-1$, or up to the MSB of the bit heap if the product is added to a wider bit heap.

To subtract the 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 (hence positive) terms may be zero, which has a positive sign: this case does need a sign extension.

We now want to exploit DSP blocks which are able to compute signed multiplications. Altera DSPs can be configured to compute 18x18 bit multiplications signed or unsigned. Xilinx DSPs (from Virtex5 on) can be configured to compute 17x24-bit unsigned, or 18x25-bit signed – here the unsigned computation simply consists in forcing the sign bit to zero.

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

$$\begin{aligned} XY = \quad & 2^{p+q-k-k'}X_h Yh && \text{– signed} \times \text{signed} \\ & +2^{p-k}X_h Y_l && \text{– signed} \times \text{unsigned} \\ & +2^{q-k'}X_l Y_h && \text{– unsigned} \times \text{signed} \\ & +X_l Y_l && \text{– unsigned} \times \text{unsigned} \end{aligned}$$

Note that this equation is a generalization of (5). With this rewriting we may now use, for the lines involving a signed number, signed multipliers offered by DSP blocks. All it takes is a sign-extension of such products to the MSB of the bit heap.

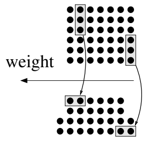


Fig. 5. One stage of compression using 3:2 compressors

IV. Timing-driven compression

This section will be illustrated by Figure 6, which describes the timing-driven compression of the bit heap of Figure 2. The compression of the lower weights begins during the DSP multiplication time: On this figure, the bits produced by the DSP are initially not present: they only appear at cycle 2.

Note that such a timing-oriented compressor is not simply a retimed version (in the Leiserson and Saxe sense [15]) of the classical one: its structure itself is guided by the timing.

A. Basics

Bit heap compression is based on a set of *elementary compressors*. These basic building blocks take a few bits from the bit heap, and replace them with their sum, hopefully on fewer bits. The most basic elementary compressor is the full adder, or (3:2) compressor: It computes the sum of three bits of same weight w, and rewrites this sum as two bits of sizes $w + 1$ and w Traditionally, 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 (Figure 5). The outputs of a stage, and possibly bits that have not been compressed in previous stages, form a new bit heap that is input to 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 present below.

B. Elementary compressors for FPGAs

We use the notation of [5] 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 diminushing return, and [8] only considers two columns.

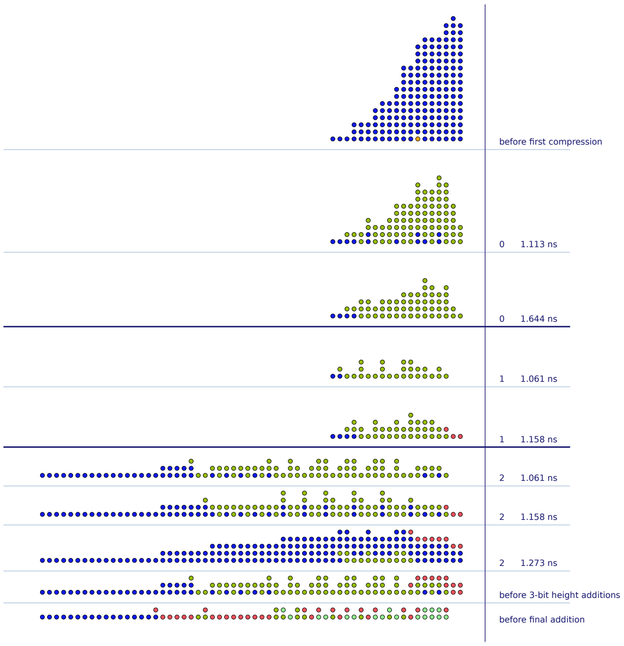


Fig. 6. Timed view of the compression of the bit heap of Figure 2 for a target 400MHz frequency.

Table I analyses the theoretical costs of some elementary compressors on an FPGA model that corresponds to modern FPGAs from both Xilinx and Altera: the elementary cell is a LUT6 which may be used as two independent LUT5, combined with some logic and routing dedicated to fast carry propagation. In this table, we report

- the compression factor γ, which is the quotient of input bits divided by 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 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 $\tau_{cp} << \tau_L$, 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 $1\tau_L$. Conversly, the arithmetic compressors from [8] involve a chain of two or three LUT6 connected by the global routing.

We synthesize in this table some of the best compressors from [8]. There are more there, and they are more accurate, with actual low-level synthesis of these compressors both on Altera and Xilinx. This synthetic table assumes an FPGA that would take the best from both their Altera and Xilinx results.

name	γ	δ_{bits}	LUT6	δ_{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 I

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

bit heap	partial product generation	compression (parallel counters)	(ternary adders)
24x24	384	505	283
32x32 trunc.	510	379	287

TABLE II

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

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 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 is a serious contender as 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 II 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. Table II gives some actual synthesis results in this case.

C. Supertiling (DSP compression)

The first step of architecture generation is to generate DSP-based hardware for the list of multiplier blocks. This simply consists in looking for DSPs that can be chained into supertiles using the 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). This is in particular the case when we use two chained DSPs to compute $(a*b)+((c*d) << s)$ where s is a constant shift (17 on Xilinx, 18 on Altera): the lower bits of the sum are those of $a*b$, and can be sent to the bit heap while the upper bits are added to the result of next multiplication.

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 4 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 Figure 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

The remaining step of the compression is currently quite simple: First, a list of elementary compressors is built, ordered by efficiency. This is target-specific, and older FPGAs with smaller LUTs are also supported. The current order is a lexicographic ordering of γ, then δ_{bits}/LUT, then δ_{bits}, then delay. There is actually little trade-off here between area and delay.

Then 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, and matching the inputs of our best compressors. Applying a compressor means generating the corresponding hardware, removing its inputs bits from the bit heap, and adding its outputs bits in the bit heap, tagged with a later time. This time is computed as 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 are then ready to be processed in following iterations of the main loop. However, they cannot be processed in the current iteration, which prevents the construction of carry propagation in one stage.

TABLE III
COMPLEX PRODUCTS TARGETTING 250MHZ ON VIRTEX-5

Approach	Performance cycles @ frequency	LUTs	Registers	DSPs
precision = 12 bits				
Mult+Add	2@328	874	188	0
Bitheap	2@328	836	108	0
precision = 24 bits				
Mult+Add	2@273	532	280	8
Bitheap	2@248	368	488	8
precision = 32 bits				
Mult+Add	2@273	2048	960	8
Bitheap	2@251	1860	462	8
precision = 64 bits				
Mult+Add	3@212	4048	4584	44
Bitheap	3@245	3348	3309	44

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 arrival of more bits, later, will enable the use of the most efficient compressors. In the latter stages, however, sub-optimal compressors should be used, otherwise the delay will increase. A good strategy, inspired by Dadda[2], 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.

There is no doubt this heuristic could be refined further. In particular, we could add the possibility to optimize specifically for area, or for delay. In [8], this seems to be managed purely by changing the ranking of the elementary compressors.

E. Results

Table III shows some synthesis results for the complex product. This is essentially a toy example stressing everything in the framework, and the results are not yet fully satisfactory. Still, it demonstrates, on this minimal example, the cost saving of a single bit heap over the instantiation of several adder and multiplier components, all other things being equal (in particular each multiplier using the same compression code as the fused bit heap).

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.

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.

A promising direction is to work on the approximation of a function directly by a bit heap. In addition, we believe that expressing the complexity of elaborate arithmetic circuits in terms of bit heaps is more pertinent than expressing them in terms of adders and multipliers. For such polynomials, there are also probably many opportunities to pre-compress sums of terms sharing the same inputs in small tables [17].

Shorter-term future work includes refinements to the presented framework, in particular the timing modelling, and its open-source distribution.

REFERENCES

[1] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, pp. 14 –17, 1964.

[2] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[3] 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.

[4] 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.

[5] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[6] E. E. Swartzlander, "Merged arithmetic," *IEEE Transactions onComputers*, 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] "A000601," On-Line Encyclopedia of Integer Sequences, http://oeis.org/A000601.

[10] 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.

[11] B. Pasca, "Correctly rounded floating-point division for DSP-enabled FPGAs," in *Field Programmable Logic and Applications*, 2012.

[12] 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.

[13] 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.

[14] 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.

[15] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5 – 35, 1991.

[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.