An FPGA Logic Cell and Carry Chain Configurable as a 6:2 or 7:2 Compressor

HADI PARANDEH-AFSHAR, PHILIP BRISK, and PAOLO IENNE
Ecole Polytechnique Federale de Lausanne (EPFL)

To improve FPGA performance for arithmetic circuits that are dominated by multi-input addition operations, an FPGA logic block is proposed that can be configured as a *6:2* or *7:2* compressor. Compressors have been used successfully in the past to realize parallel multipliers in VLSI technology; however, the peculiar structure of FPGA logic blocks, coupled with the high cost of the routing network relative to ASIC technology, renders compressors ineffective when mapped onto the general logic of an FPGA. On the other hand, current FPGA logic cells have already been enhanced with carry chains to improve arithmetic functionality, for example, to realize fast ternary carry-propagate addition. The contribution of this article is a new FPGA logic cell that is specialized to help realize efficient compressor trees on FPGAs. The new FPGA logic cell has two variants that can respectively be configured as a *6:2* or a *7:2* compressor using additional carry chains that, coupled with lookup tables, provide the necessary functionality. Experiments show that the use of these modified logic cells significantly reduces the delay of compressor trees synthesized on FPGAs compared to state-of-the-art synthesis techniques, with a moderate increase in area and power consumption.

Categories and Subject Descriptors: B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate Arrays*; G.1.0 [**Numerical Analysis**]: General—*Computer Arithmetic*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: FPGA, carry chain, compressor tree, 6:2 compressor, 7:2 compressor

ACM Reference Format:
Parandeh-Afshar, H., Brisk, P., and Ienne, P. 2009. An FPGA logic cell and carry chain configurable as a 6:2 or 7:2 compressor. ACM Trans. Reconfig. Techn. Syst. 2, 3, Article 19 (September 2009), 42 pages. DOI = 10.1145/1575774.1575778. http://doi.acm.org/10.1145/1575774.1575778.

P. Brisk is currently affiliated with the Department of Computer Science and Engineering in the Bourns College of Engineering at the University of California, Riverside.
Author's address: H. Parandeh-Afshar, email: hadi.parandehafshar@epfl.ch; P. Brisk, email: Philip.brisk@gmail.com; P. Ienne, email: paolo.ienne@epfl.ch.

1. INTRODUCTION

Due to their inherent reconfigurability, FPGAs are one feasible hardware platform for low-volume markets, where vendors cannot justify the design, testing, and verification costs of an ASIC. Although an FPGA implementation of a circuit will outperform traditional software, a noticeable performance gap between FPGAs and ASICs remains [Kuon and Rose 2007]. One important area that is ripe for improvement is arithmetic dominated circuits; in particular, due to the peculiar logic cell structure and carry chains in modern FPGAs, addition and multiplication-dominated circuits cannot take advantage of the carry-save representation. One of the fundamental results in computer arithmetic is that addition scales well when the number of inputs increases beyond 2; this was first observed by Wallace [1964] in the context of parallel multiplier design. The key is *not* to use trees of traditional *carry-propagate adders*, that is, circuits that produce the sum of two (signed) binary integers; instead, the integers are aggregated together using a circuit called a *compressor tree*. Numerous methods for compressor tree generation have been published since their introduction in the early 1960s [Wallace 1964; Dadda 1965; Swartzlander 1973; Stenzel et al. 1977; Weinberger 1981; Santoro and Horowitz 1988; Song and De Micheli 1991; Fadavi-Arkedani 1993; Oklobdzija and Villeger 1995; Stelling and Oklobdzija 1996; Stelling et al. 1998; Kwon et al. 2002; Um and Kim 2002; Mora Mora et al. 2006; Verma and Ienne 2007a], mostly in the context of parallel multiplication; more generally, these circuits can also sum $k > 2$ integers.

The architecture of modern FPGAs is generally not well suited to compressor trees. The logic clusters of the *Altera Stratix II-IV* and *Xilinx Virtex-5* FPGAs can be configured to implement ternary (3-input) addition using fast carry chains [Cherepacha and Lewis 1996; Hauck et al. 2000; Frederick and Somani 2006]. The primary advantage of the carry chains is that the carry bits are propagated directly from one cell to its adjacent neighbor, thereby avoiding the overhead of the routing network. This design point favors the use of ternary adder trees rather than compressor trees.

Parandeh-Afshar et al. [2008b, 2008c] showed that compressor trees can be synthesized on FPGAs using a circuit called a *Generalized Parallel Counter (GPC)* [Stenzel et al. 1977]. This *GPC Mapping* approach yields compressor trees whose delay is significantly lower than ternary adder trees, despite the latter's use of the carry chains; however, there is some noticeable increase in the number of logic cells required.

This article, an extension of prior work by Parandeh-Afshar et al. [2008a], introduces and evaluates a new logic cell, based on the *Altera Adaptive Logic Module (ALM)*, that has an additional carry chain, which allows it to be configured as a *6:2* or *7:2 compressor*; this compressor belongs to a well-known class of circuits that have been used for successful synthesis of ASIC multipliers in the past [Weinberger 1981; Song and De Micheli 1991; Oklobdzija and Villeger 1995]. By combining the strengths of the GPC mapping with the use of *6:2* or *7:2* compressors, when possible, faster compressor trees can be realized on the FPGA. Additionally, we compare the power consumption of compressor trees

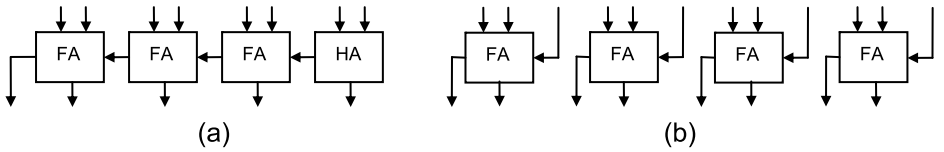


Fig. 1. (a) A ripple-carry adder; (b) a carry-save adder.

mapped onto the proposed logic cells with compressor trees synthesized using ternary adder trees and GPC mapping.

The article is organized as follows. Section 2 begins by introducing a collection of arithmetic primitives (counters, compressors, compressor trees) that are required to understand the remaining sections of the paper. Section 3 summarizes related work in the field of FPGA architecture and mapping, focusing specifically on features designed for enhanced arithmetic performance. Section 4 presents the new logic cell, and Section 5 describes the approach that we used to map circuits onto FPGAs containing the new cell. Our experimental platform, methodology, and results are presented in Sections 6–8. Section 9 concludes the article.

2. ARITHMETIC AND FPGA PRIMITIVES

2.1 Full and Half Adders

At the bit level, a *half-adder (HA)* is a *2*-input, *2*-output circuit that computes the sum of two bits and outputs the result as an unsigned binary integer. A *full-adder (FA)* computes a similar sum for *3* input bits. The lower-order output bit is called a *sum*, and the higher-order output bit is called a *carry*. In the case of an FA, one of the inputs is called a carry-in bit and the high-order output is called a carry-out. Many arithmetic circuits, including adders and multipliers are comprised primarily of HAs and FAs.

2.2 Ripple-Carry and Carry-Save Adders

A *Carry Propagate Adder (CPA)* is a circuit that adds two binary integers; if the integers are signed, two's complement form is assumed. Numerous architectures for carry-propagate adders have been proposed in the past. In modern CMOS technologies, significant differences in critical path delay among the different adder architectures generally do not manifest themselves for small bitwidths, that is, *8*-bits or less.

The most straightforward CPA architecture is the *Ripple-Carry Adder (RCA)*, which generally has the smallest area but highest delay compared to the alternatives. Figure 1(a) shows a *4*-bit RCA constructed from FA cells; the carry-in of the least significant FA is *0*, so an HA can be used instead of an FA.

As shown in Figure 1(a), an RCA is a *1*-dimensional array of FAs, where the carry-out of each FA is connected directly to the carry-in of the next; thus, the worst-case critical path delay is through all of the FAs in the design. If an RCA adds two k-bit numbers, the complexity of the critical path delay is $O(k)$. Many faster, but larger, alternative adders have been designed, most with a critical path delay of $O(\log k)$.

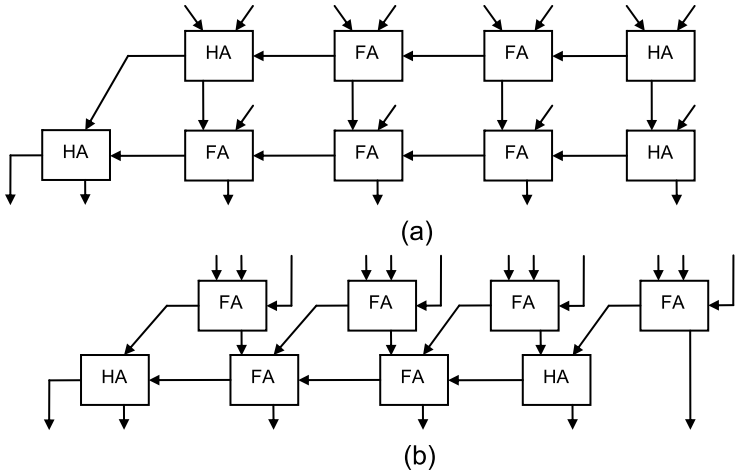


Fig. 2. Two implementations of a 4-bit ternary adder using (a) an adder tree, i.e., two RCAs; and (b) a compressor tree, i.e., a CSA followed by an RCA. The compressor tree implementation eliminates the delay of one half adder (HA) from the critical path.

A *Carry-Save Adder (CSA)*, shown in Figure 1(b), breaks the carry chain; in fact, it is a *1*-dimensional array of *disconnected* FAs. CSAs are generally used in conjunction with CPAs in order to perform efficient n-input addition for $n > 2$.

2.3 Adder and Compressor Trees

Suppose that we want to compute the sum of $n > 2$ binary integers. One approach is to use an *Adder Tree*, that is, a tree of CPAs; the alternative is to build a tree of carry-save adders instead, only using a CPA at the end. Figure 2 shows an example where three four-bit binary integers are added. In Figure 2(a), two RCAs are used; in Figure 2(b), a CSA is followed by an RCA. Let d_{FA} and d_{HA} are the respective delays of full and half adders. The critical path delay of the circuit in Figure 2(a) is $4d_{FA} + 2d_{HA}$, while the critical path delay of the circuit in Figure 2(b) is $3d_{FA} + 2d_{HA}$, an overall savings of d_{FA} compared to Figure 2(a). This savings occurs because the use of the CSA instead of the RCA permits the elimination of one bit from the RCA in Figure 2(b).

The idea of using carry-save addition for fast accumulation dates back to the work of Wallace [1964] and Dadda [1965] who designed fast parallel multipliers; however, the fundamental ideas generalize quite elegantly to multiinput addition as well.

Formally, let $A_1, A_2, \ldots, A_n$ be a set of binary integers to sum. A *Compressor Tree* is a circuit that produces two values, *sum (S)* and *carry (C)*, such that:

$$S + C = \sum_{i=1}^{n} A_i. \tag{1}$$

A CPA then performs the final addition, $S + C$.

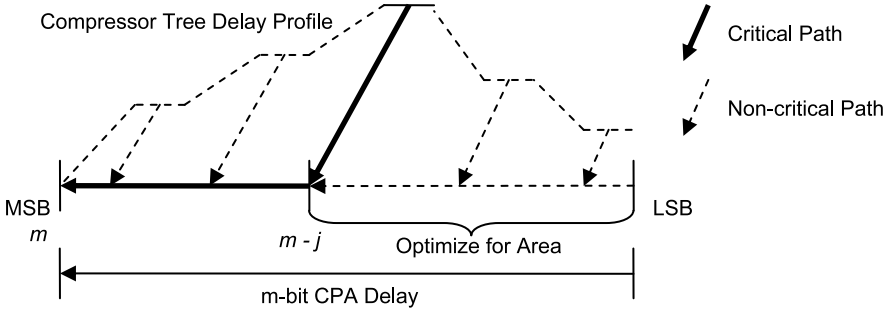


Fig. 3. Illustration of the critical path delay through a compressor tree of a multiplier, including that of the final CPA. The critical path typically includes the j most significant bits of the final CPA; the portion of the final CPA that computes the $m - j$ least significant bits can be optimized for area rather than for speed, as long as it does not become critical.

Wallace and Dadda trees are two specific compressor tree architectures; many others have also been proposed [Swartzlander 1973; Stenzel et al. 1977; Weinberger 1981; Santoro and Horowitz 1988; Song and De Micheli 1991; Fadavi-Arkedani 1993; Stelling and Oklobdzija 1996; Stelling et al. 1998; Kwon et al. 2002; Um and Kim 2002; Mora Mora et al. 2006; Verma and Ienne 2007a].

The superiority of compressor trees over adder trees is one of the most fundamental results of digital arithmetic. Intuitively, it may seem that this is because an adder tree pays the penalty of a carry chain at each level; this is, however, a fallacy, as illustrated by Figure 2 in the preceding discussion. In actuality, the benefit of compressor trees arises from their ability to reduce the bitwidth of the final CPA in the case of multiinput addition. Parallel multipliers in ASIC technology, however, are more complicated.

In multiinput addition, the number of bits to sum at each position is the same. This is not true in the case of parallel multiplication: after a partial product generation or Booth encoding stage, the number of bits to sum tends to be greater among the bit positions in the middle. As illustrated conceptually by Figure 3, the lower-order bits of the final CPA are generally not on the critical path, as the bits that arrive at these positions go through fewer layers of logic within the compressor tree. In other words, the arrival time of the bits at the final CPA is nonuniform, unlike the case of multiinput addition. Based on this observation, Oklobdzija and Villeger [1995] argued that the final CPA of a multiplier should be implemented as a hybrid adder, which uses a small and slow CPA, such as an RCA, for the low-order bits, and a faster adder, such as a carry-select adder for the higher-order bits.

Carry-select adders are particularly useful when the arrival time of bits is nonuniform. Carry-select adders can start to add the bits as soon as they arrive. RCAs, in contrast, cannot, as the output bit at position i depends on the carry-out bit computed at position $i - 1$. That being said, carry-select adders can be constructed from smaller-bitwidth RCAs as building blocks.

The work summarized in this section targets ASIC design methodologies; FPGAs, in contrast, possess fast carry chains, whose usage often dictates the types of adders that perform well on specific device families.

2.4 Parallel Counters

An *m:n parallel counter* (or single-column counter) is a circuit that takes m input bits, counts the number of input bits that are set to one, and outputs the value as an n-bit binary unsigned integer. The output range is $[0, m]$, so the number of output bits is:

$$n = \lceil log_2 (m + 1) \rceil . \tag{2}$$

In the context of compressor trees, HAs and FAs are *2:2* and *3:2* counters respectively. Verma and Ienne [2007a], for example, described an integer linear programming formulation for compressor tree design that uses a library of *m:n* counters, for $2 \leq m \leq 8$.

Let $B = b_{k-1} b_{k-2} \ldots b_0$ be a k-bit unsigned binary integer, where b_{k-1} is the most significant bit, and b_0 is the least significant bit. Each bit b_r contributes a total value of $b_r 2^r$ to the total value of B, i.e., b_r contributes 2^r if it is set, and 0 otherwise. In this context, r is called the *rank* of b_r.

When an *m:n* counter is used to synthesize a compressor tree, all of its inputs have the same rank. A *Generalized Parallel Counter (GPC)* is an extension of an *m:n* counter that can sum bits of multiple ranks [Stenzel et al. 1977]. For example, a $(2, 3; 3)$ GPC can sum up to *2* bits of rank *1* and *3* bits of rank *0*; the maximum output value is $2 \times 2^1 + 3 \times 2^0 = 7$, so *3* output bits are required. The general form of a GPC is $(k_{t-1}, k_{t-2}, \ldots, k_0; s)$, where k_r is the maximum number of bits of rank r that can be summed, and s is the number of output bits. Similar to an *m:n* counter, a *GPC* must satisfy the following property:

$$s = \left\lceil log_2 \left(1 + \sum_{r=0}^{t-1} k_r 2^r \right) \right\rceil . \tag{3}$$

In fact, a sufficiently large *m:n* counter can implement a GPC (although many other implementations also exist). Each GPC input bit of rank r is connected to 2^r inputs of the *m:n* counter; any unused input bits of the *m:n* counter are then driven to 0.

GPCs map efficiently onto FPGAs [Parandeh-Afshar et al. 2008b, 2008c]. Specifically, if the FPGA has k-input LUTs, then k-input GPCs can be mapped onto the LUTs (one LUT is used per GPC output bit) using one logic level.

2.5 Compressors

Compressors (not to be confused with compressor *trees*) are arithmetic components, similar in principle to parallel counters, but with two distinct differences: (1) they have explicit carry-in and carry-out bits; and (2) there may be some redundancy among the ranks of the sum and carry-output bits.

The *4:2* compressor (also called a *4:2* CSA), illustrated in Figure 4, was introduced by Weinberger [1981]; at first sight, this name may appear to be somewhat of a misnomer: although it has *4* input bits and produces *2* sum output bits (out_0 and out_1), it also has a carry-in (c_{in}) and a carry-out (c_{out}) bit (thus, the total number of input/output bits are *5* and *3*); however, it is *not* the same circuit as a *5:3* compressor. All input bits, including c_{in}, have rank *0*; the two output bits have ranks *0* and *1* respectively, while c_{out} has rank *1* as well. Thus,

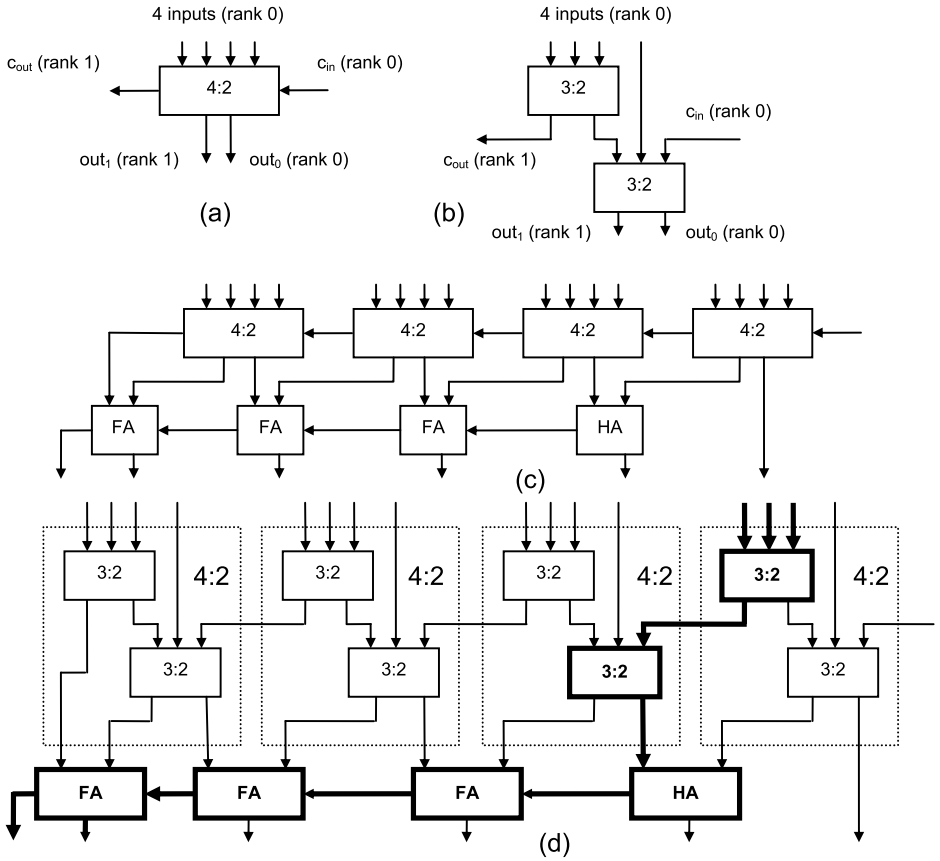


Fig. 4. (a) *4:2* compressor I/O diagram; (b) *4:2* compressor architecture; (c) *4*-ary adder built from an array of *4:2* compressors followed by an RCA; (d) illustration of the interconnect between consecutive *4:2* compressors: although the array has the appearance of an RCA in Figure 4(c), the carry chain only goes through two compressors.

the output of the *4:2* compressor is a redundant number; for example, $out_1 = 0$ and $c_{out} = 1$ is equivalent to $out_1 = 1$ and $c_{out} = 0$ in all cases.

When k *4:2* compressors are connected in a carry chain, a total of $4k$ input bits are *compressed* down to $2k$ output bits plus one additional carry-out bit; the carry-in bit of the first compressor is set to *0*. The primary difference between compressors and counters are the presence of carry bits in the former; it is also important to recognize that a compressor tree can be constructed from compressors, counters, or both.

Figure 4(a) shows the inputs and outputs of the *4:2* compressor labeled with their ranks; Figure 4(b) shows one *4:2* compressor architecture, which is constructed using two *3:2* counters. Figure 4(c) shows a *4*-bit *4*-input adder, consisting of four *4:2* compressors in a *1*-dimensional array followed by a four-bit RCA. At first glance, the array of *4:2* compressors appears to have the same structure as an RCA, as the c_{out} bit of each *4:2* compressor is connected to the c_{in} bit of the subsequent one; however, this is not actually the case, as shown in

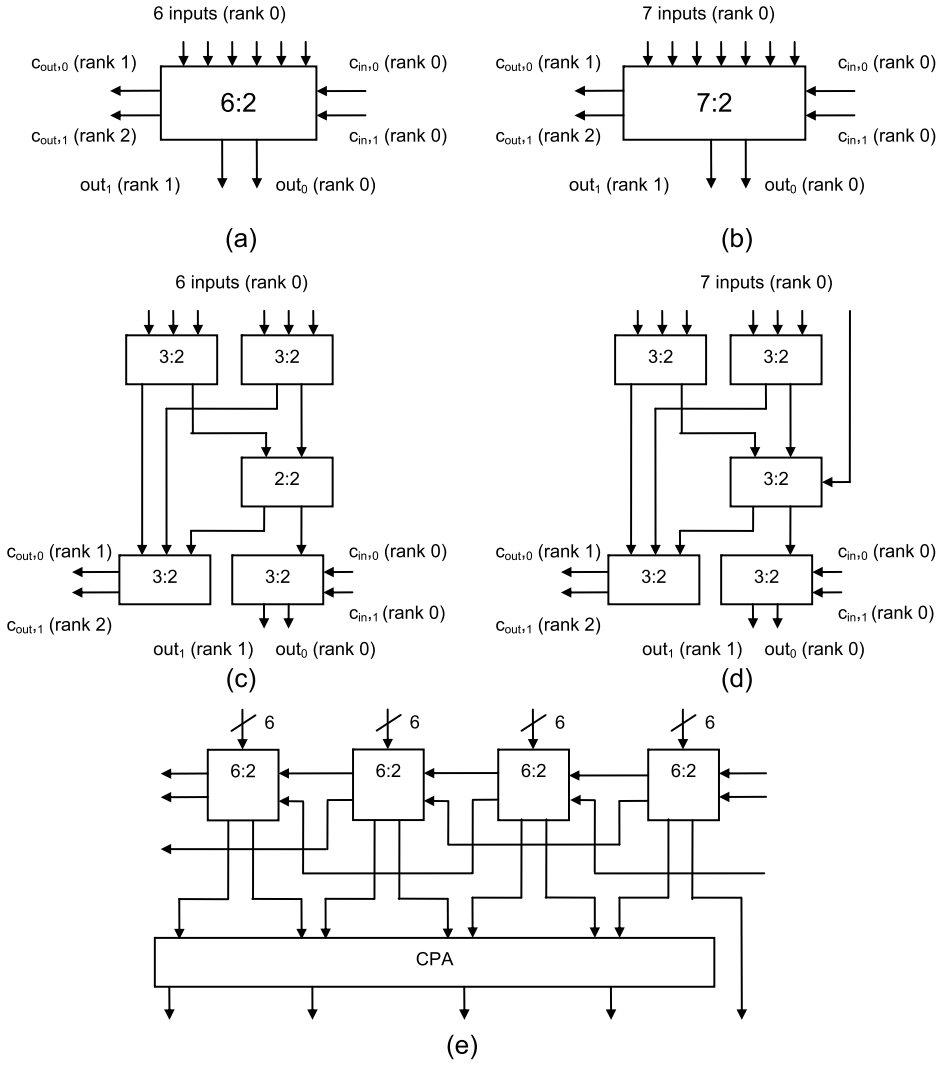


Fig. 5. (a)/(b) *6:2/7:2* compressor I/O diagram; (c)/(d) *6:2/7:2* compressor architecture; (e) illustration of the interconnection pattern between consecutive *6:2* compressors (it is the same for *7:2* compressors).

Figure 4(d); the fact that there is no direct path from a carry-in to a carry-out prevents the formation of a ripple-carry structure.

The new FPGA logic cell described in this paper has two variants that can respectively be configured as a *6:2* or a *7:2* compressor, which generalize the *4:2* compressor cell whose use is shown in Figure 4. Figure 5(a) and (b) show the basic I/O structure of the *6:2* and *7:2* compressors. Figure 5(c) and (d) show the circuit-level architecture; the only difference is that a *2:2* counter in the *6:2* compressor is upgraded to an *3:2* counter in the *7:2* compressor, and the 7^{th} input is connected to one of the inputs the aforementioned *3:2* counter.

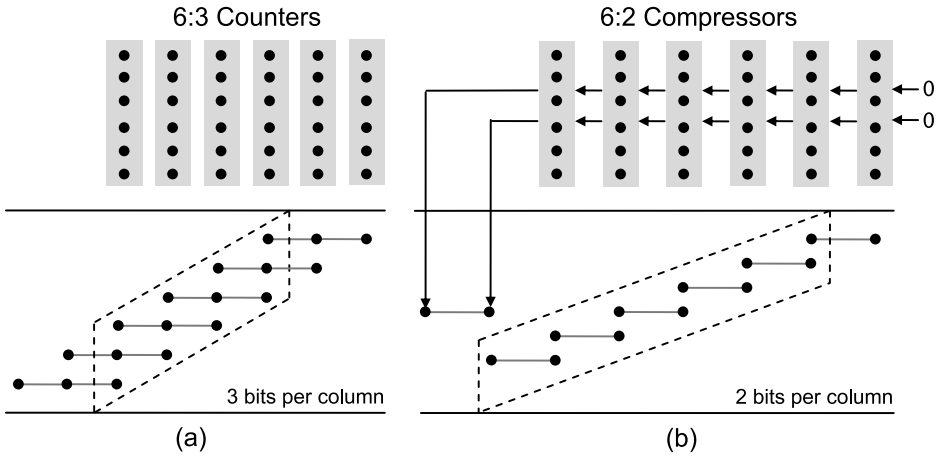


Fig. 6. (a) Covering a set of columns with *6:3* counters yields 3 bits per column in the output; (b) using *6:2* compressors reduces the number of bits per column to 2. Contiguous columns covered with *6:3* counters can be converted to *6:2* compressors.

Figure 5(e) shows the interconnect structure. Consider the i^{th} compressor in sequence. The rank1 carry output bit ($c_{out,0}$) connects to carry-input $c_{in,0}$ of the $(i+1)^{st}$compressor; also, the rank *2* carry output bit ($c_{out,1}$) connects to carry-input $c_{in,1}$ of the $(i+2)^{nd}$ compressor.

From Figure 5(c) and (d), we can see that there is no direct path from either of the carry-in bits of the *6:2* or *7:2* compressor to one of the carry-out bits; similar in principle to Figure 4(d), this prevents the formation of a ripple-carry chain between compressors.

2.6 Compression Ratio

Let I and O be the number of inputs and outputs produced by a counter, GPC, or compressor; for compressors, I and O do not include the carry-in and carry-out bits. The *compression ratio (CR)* is defined as $CR = I/O$. For example, a *6*-input, *3*-output GPC has $CR = 6/3 = 2$, while a *6:2* compressor has $CR = 6/2 = 3$. The CR tends to be higher for compressors than counters. Figure 6(a) shows compression using *6:3* counters; which produce three output bits per column, while *6:2* compressors, shown in Figure 6(b), produce two output bits per column; the other output bits are propagated down the carry chain.

3. RELATED WORK

3.1 Compressor Tree Synthesis in ASIC Technology

Compressor trees for partial product accumulation were introduced by Wallace [1964] and Dadda [1965], who built them from CSAs; HAs were used at points where only *2* bits in the same column need to be compressed. Fadavi-Ardekani [1993] recognized that the bits produced by a compressor tree may arrive at different times at the final adder, and designed a specific adder for this purpose; however, this work assumed that all partial product bits arrive to the compressor tree at the same time. Stelling et al. [1996, 1998] relaxed this

assumption, and developed appropriate techniques to build the compressor tree and designed the final adder appropriately.

Due to the importance of wire delays in deep submicron technology, Um and Kim [2002] proposed a two-phase layout-aware compressor tree synthesis technique that strives for a much more regular interconnect topology than the compressor trees produced by the 3-greedy algorithm of Stelling et al. [1998].

Verma and Ienne [2007a] developed an integer linear program (ILP) that could optimally synthesize compressor trees from a library of *m:n* counters. To bound the runtime of the synthesis procedure, they limited *m* to the range *[2, 8]*. Previously, *m:n* counters, like compressor trees, were built from CSAs, or libraries of smaller *m:n* counters. Through efficient logic synthesis techniques for arithmetic circuits [Verma and Ienne 2007b], they found that better *m:n* counters could be constructed from basic gates, rather than smaller counters. The availability of a library of highly optimized counters was important to the success of their ILP formulation; another contributing factor was that the ILP could optimize for the delay profile of any final adder.

GPCs have also been used in the past to build efficient compressor trees for parallel multipliers [Stenzel et al. 1977]. Mora Mora et al. [2006] described a multiplier generation approach for ASICs that implemented GPCs using ROMs, with the restriction that all input columns to the GPC have the same number of bits.

The *4:2* compressor [Weinberger 1981], was subsequently used by Santoro and Horowitz [1988] in a 64 × 64 parallel multiplier. Over the years, various researchers have proposed the use of larger compressors and counters as well, including Kwon et al. [2002] (*5:2, 5:3)* and Song and De Micheli [1991] (*9:2, 27:5)*.

A *column* is a set of bits having the same rank, *r*, at some level in a compressor tree; all of the inputs to a FA or an HA in a compressor tree belong to the same column. The FA or HA produces two output bits, one of rank *r*, one of rank $r + 1$. The delay through the FA or HA to the rank *r* output is called the *vertical* propagation delay, as the delay is confined to one column; the delay of the rank $r + 1$ output is called the *horizontal* propagation delay, as it passes from one column to the next. The use of compressors in favor of counters shifts some of the vertical propagation delay into horizontal propagation delay. Thus, the critical path through a compressor tree travels in both the horizontal and vertical direction before arriving at the final CPA. The compressor cells can be designed in order to minimize the difference between horizontal and vertical delays.

Interestingly enough, a CPA actually has a higher compression ratio than an *m:n* counter, a GPC, or a compressor. To take advantage of this fact, Oklobdzija and Villeger [1995] advocate the inclusion of CPAs within compressor trees: the vertical propagation delay will dominate; however, at places where the horizontal propagation delay is noncritical, the use of internal CPAs within the compressor tree maximizes the compression ratio. This technique has some notable ramifications for FPGAs: due to the presence of carry chains within logic clusters (see Sections 3.3 and 3.4), horizontal propagation is naturally faster than vertical propagation, which must use the FPGA routing

network. This differentiates compressor tree synthesis on FPGAs from the same problem in VLSI.

The challenge is that we cannot take advantage of the fast horizontal propagation on the critical paths of the compressor tree without resorting to CPAs. To address this concern, we design and evaluate new logic blocks that can be configured as *6:2* and *7:2* compressors. These logic blocks have a higher compression ratio than *m:n* counters and *GPC*s, and employ new carry chains that can exploit fast horizontal propagation.

3.2 FPGA Architecture

This section describes a number of proposals to improve the arithmetic and logical capabilities of FPGA logic cells. The most enduring idea has been the integration of carry chains into FPGA logic cells along with LUTs. Carry chains include fast connections between adjacent logic cells that are used for carry propagation; this permits the elimination of most of the routing delays that would otherwise be present.

The *Altera Stratix II-IV Adaptive Logic Module (ALM)* employs a carry chain based on ripple-carry adders (RCAs). The new logic cells proposed in this work features a new type of carry chain intended to allow a logic cell, such as the ALM, to be configured as a *6:2* or *7:2* compressor; the ALM will be described in greater detail in Section 3.3. The carry chains used in the *configurable logic blocks (CLBs)* of the *Xilinx Virtex*-4/5 include programmable multiplexors and *xor* gates to send propagate and generate signals to adjacent CLBs to enable parallel-prefix style addition [Parhami 1999].

Hauck et al. [2000] proposed more complicated carry chains that can implement *Brent-Kung*, *carry-select*, and *carry-lookahead* addition. Different logical constructs were needed for different cells in the chain, making them nonuniform. This creates integration challenges because it is difficult to lay out a regular fabric consisting of irregular cells. This would require a large manual effort to design each individual cell at the transistor level, and would complicate the layout process for the entire chip.

Frederick and Somani [2006] proposed a uniform logic block with carry chains that could efficiently implement a *carry-skip adder*; a similar bidirectional carry-skip chain was earlier proposed by Cherepacha and Lewis [1996, Figure 6]. Kaviani et al. [1998] and Leijten-Nowak and Van Meerbergen [2003] developed ALU-like blocks that support arithmetic functions such as addition, subtraction and (partial) multiplication.

Distributed Arithmetic (DA) [Mirzaei et al. 2006] is a paradigm for implementing effective hardware for DSP systems that uses LUTs instead of multipliers. Grover et al. [2002] developed a special DA-oriented LUT structure (*DALUT*) specifically for *multiply-accumulate (MAC)* operations. In addition to two 4-input LUTs, their DALUT cell included arrays of *xor* gates, bit-level adders and shift accumulators, shift registers, and a CPA to add partial summations and carries. Brisk et al. [2007] reported that DSP/MAC blocks are not good candidates for implementing multioperand addition. The logic cell described here is intended to address this shortcoming.

Most FPGAs are hybrid-reconfigurable, as they embed ASIC components such as multipliers, more complex DSP blocks, and standard I/O interfaces into a reconfigurable fabric Zuchowski et al. [2002]. Kastner et al. [2002] developed techniques for a compiler to examine a set of applications to identify good candidates for these embedded cores. Their analysis, however, was limited to 2-operation combinations of addition and multiplication, and they did not use compressor trees for multioperand addition.

A *K-input macro gate* [Cong and Huang 2005] is similar to a LUT, but it cannot implement all 2^K logic functions, and therefore has reduced delay and area. Hu et al. [2007] suggested that FPGA cells could benefit from the inclusion of both LUTs and macro gates. Similar to Kastner et al., they developed an automated method to profile a set of applications to find good macro-gate candidates. They did not, however, consider arithmetic-dominated functions or fast carry chains between macro gates.

The *Field Programmable Counter Array (FPCA)* [Brisk et al. 2007; Cevrero et al. 2008] is a programmable IP used to accelerate multi-input addition in FPGAs. The FPCA is similar to an FPGA, but replaces LUTs with *m:n* counters instead. In a hybrid FPGA/FPCA, a compressor tree is mapped onto the FPCA, while all other operations are mapped onto the FPGA. As suggested by Kuon and Rose [2007], the cost of routing data to and from the FPCA may limit its performance benefit. The new FPGA cell proposed here is much less ambitious, and exploits carry chains rather than logical structures for effective local routing; furthermore, the I/O interface to the logic cell does not change.

3.3 The Altera Stratix II-IV Adaptive Logic Module (ALM)

This new logic cell proposed in this article is a modified version of the *Adaptive Logic Module (ALM)* employed the *Altera Stratix II-IV* series of FPGAs. Each ALM contains an *Adaptive LUT (ALUT)*. An ALUT is comprised of two six-input LUTs (6-LUTs) with four shared inputs and shared configuration bits; in other words, they must implement the same logic function. Additionally, the ALM contains a carry chain that performs efficient ripple carry addition, and bypassable flip-flops that facilitate either combinational or sequential circuits. The two 6-LUTs are also fracturable, meaning that each can be decomposed into two or more smaller LUTs. The ALM also includes a 7^{th} input bit, but can only implement a selected set of 7-input functions.

The ALM has four operating modes, two of which use the carry chains. In *Arithmetic Mode*, each 6-LUT is decomposed into two independent 4-LUTs, which perform a small amount of pre-adder logic, followed by the carry chains. Arithmetic mode implements effective adders, (sequential) counters, accumulators, parity functions, and comparators.

In *Shared Arithmetic Mode*, the ALM is configured as a *2*-bit ternary adder. The fracturable LUTs are configured as a carry-save adder (CSA), that is, a *3:2* compressor, and the carry chain functions as the final adder. Shared arithmetic mode was designed to efficiently implement soft multipliers (as opposed to using DSP blocks) and correlators.

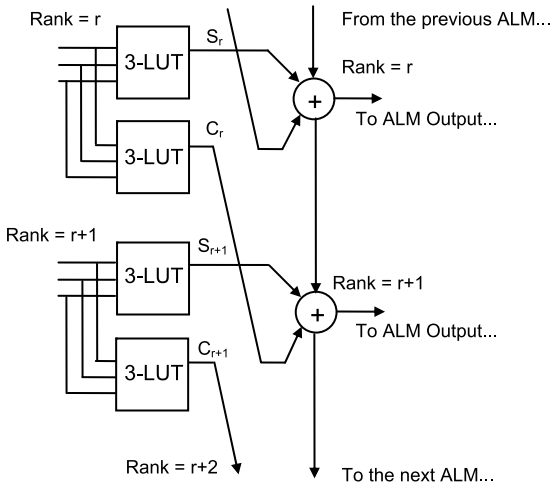


Fig. 7. The Altera Stratix II/III Adaptive Logic Module (ALM) shown in "Shared Arithmetic Mode."

Figure 7 illustrates the ALM configured in shared arithmetic mode. It is important to note that the *6*-LUTs in the ALM are decomposed into smaller LUTs of *3*- and *4*-inputs; only the smaller LUTs are shown in Figure 7.

The modification to the ALM proposed in this article is similar to shared arithmetic mode, but implements a *6:2* or *7:2* compressor. Similar to shared arithmetic mode, the fracturable LUTs are configured as a CSA; but the interconnection of FAs in the carry chain differs from the ripple-carry chain. We chose to provide a second carry chain in addition to the ripple-carry chain; in principle, both carry chains could be merged, but this would introduce multiplexers into the ripple-carry chain. We opted for the second carry chain in order to achieve better performance.

3.4 Synthesizing Compressor Trees on FPGAs

The compressor tree synthesis techniques summarized in Section 3.1 are intended for ASIC design flows. Due to the specific logic and routing architectures of modern high performance FPGAs, these techniques are not likely to yield favorable results if used in a synthesis flow targeting an FPGA. Since the primary role of carry chains has been to facilitate efficient carry-propagate addition, conventional wisdom held that adder trees would yield better results than compressor trees synthesized on an FPGA. This is not necessarily true.

Poldre and Tammemae [1999] synthesized *4:2* compressors onto the four input LUTs of the Xilinx Virtex FPGAs, exploiting the carry chains to propagate the carry-in/carry-out bits. Parandeh-Afshar et al. [2008b, 2008c] developed a general compressor tree synthesis method that mapped GPCs with *6* inputs and *3* or *4* outputs onto FPGA logic cells built from *6*-LUTs. Limiting the number of GPC inputs to *6* ensures that at most one layer of LUTs is required to implement each GPC. On an Altera Stratix II, the delay of a compressor tree built from GPCs was 27% faster than that that of an

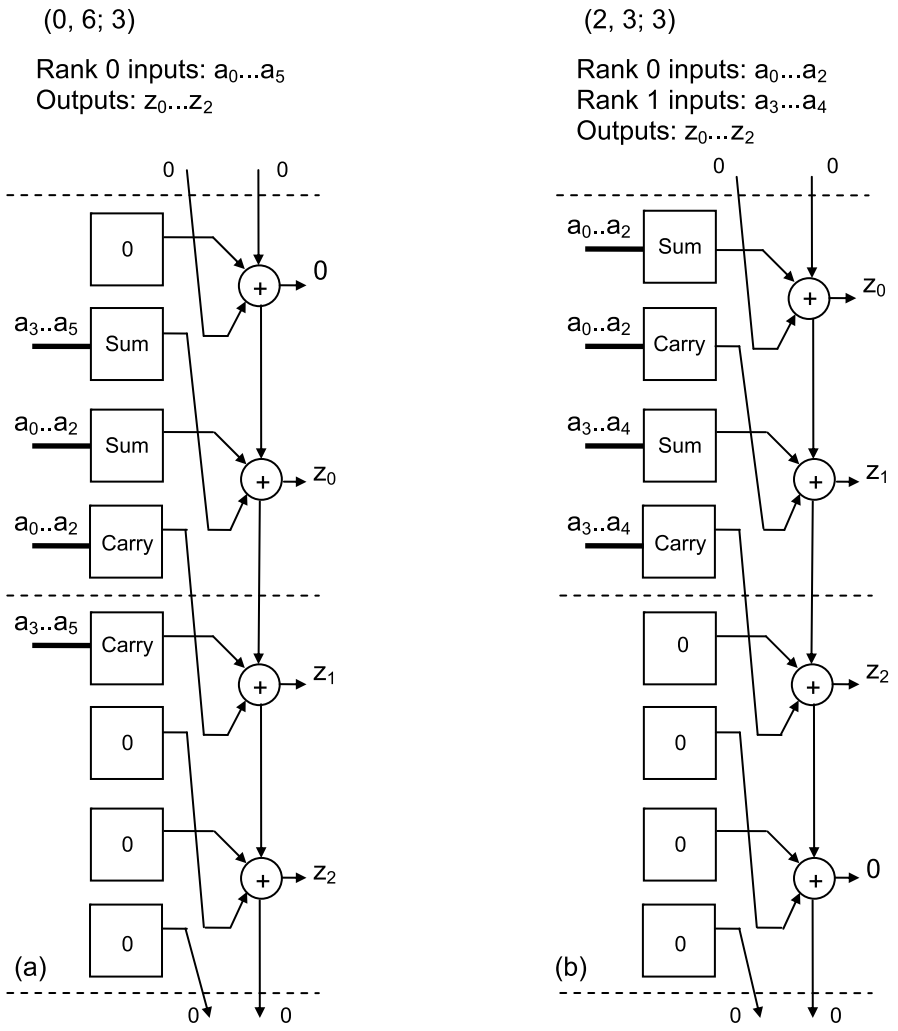


Fig. 8. (a) (0, 6; 3) and (b) (2, 3; 3) GPCs mapped onto two ALMs using shared arithmetic mode.

adder tree. The GPC mapping, however, increased the ALM count by 47%, on average.

In principle, a *6*-input, k-output GPC could be synthesized on k *6*-LUTs, where each *6*-LUT computes a single output bit. As the two *6*-LUTs in a Stratix II-IV ALM must implement the same function, this would require k ALMs, where only one of the two 6-LUTs available in each ALM is used. Parandeh-Afshar et al. [2009], however, proposed a more efficient mapping that uses LUTs in conjunction with carry chains, reducing the number of ALMs required to $\lceil k/2 \rceil$. In many cases, it is possible to map these components onto ALMs using either arithmetic or shared arithmetic mode.

Figures 8 and 9, for example, shows three *6*-input, *3*-output GPCs mapped onto two ALMs using shared arithmetic mode. In fact, these are the only three

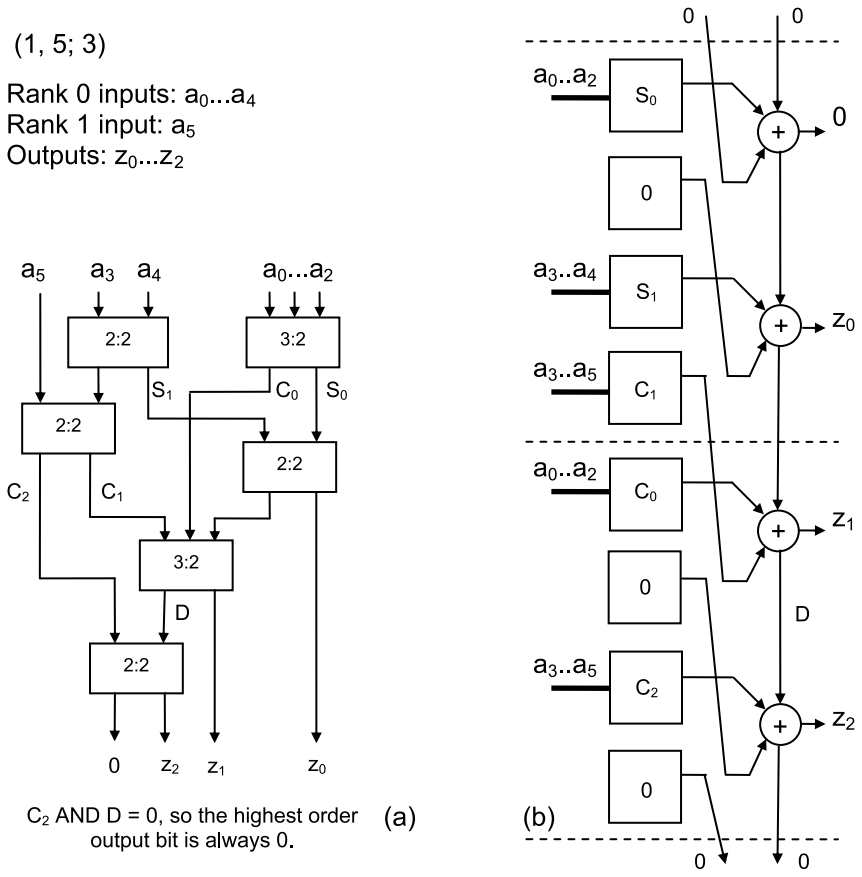


Fig. 9. (a) a (1, 5; 3) GPC implemented using full and half adders, and (b) mapped onto two ALMs using shared arithmetic mode. The internal signals S0, C0, S1, C1, and C2 in (a) are computed by LUTs in (b). Signals C2 and D are never 1 at the same time, so the carry output of the adder that produces output bit z2 is always 0.

6-input, *3*-output GPCs that will be used by the *GPC* mapping heuristic, which is described in Section 5.3.

Specifically, these are the only *6*-input, *3*-output *covering GPCs*; the definition of a covering GPC will be formalized in Section 5.1. The GPC mapping heuristic only employs covering GPCs; all other GPCs are either redundant or unreasonable, for reasons that will be discussed in Section 5.1.

4. NEW FPGA LOGIC CELL AND CARRY CHAIN

Figure 10(a) shows our proposed new FPGA logic cell, which is presented as an extension of the ALM used in Altera's Stratix II-IV line of high-performance FPGAs. The components required for "shared arithmetic mode" are also shown in this figure. The left-hand side of Figure 10(a) shows four *3*-LUTs, which are part of Altera's "fracturable" *6*-LUT architecture. The carry chain on the right-hand-side is the traditional carry chain that is used to implement ternary

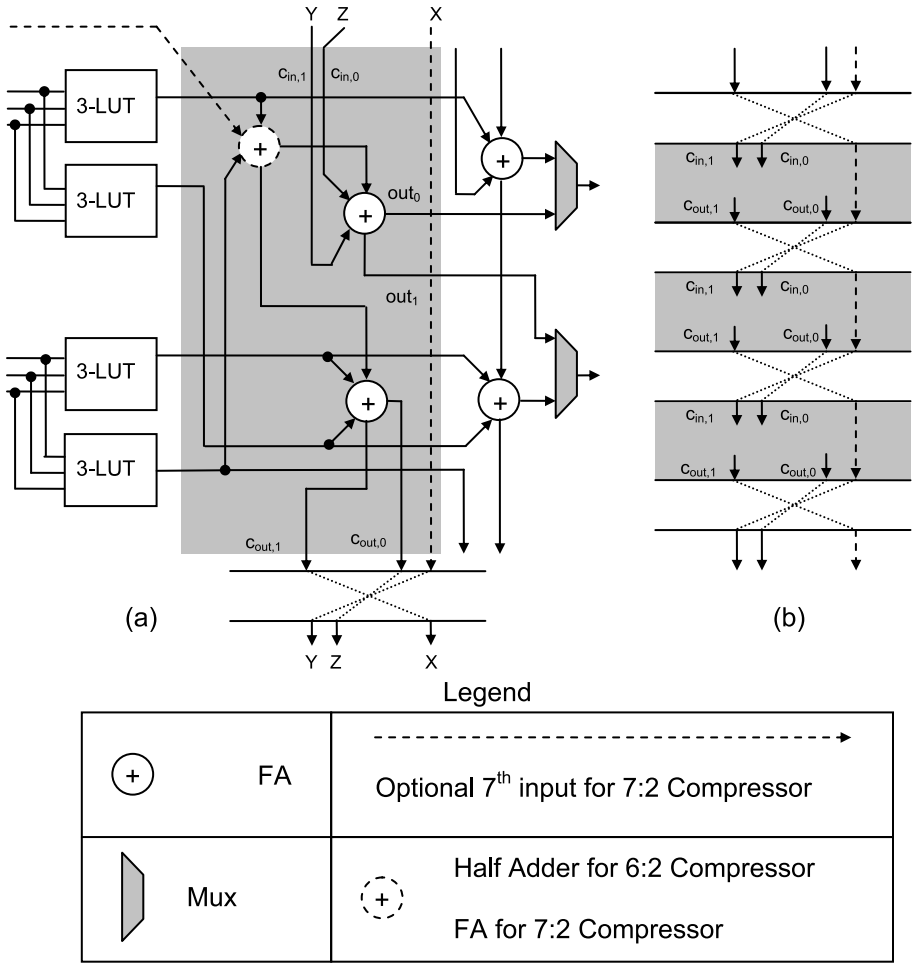


Fig. 10. (a) Enhanced version of the "Shared Arithmetic Mode" of the Altera ALM; a new carry chain, shown in gray, allows the ALM to be configured as a *6:2* or *7:2* compressor. Two additional multiplexers are required to select between the two "sum" outputs of the *6:2* compressor and ternary adder (already present in the ALM); (b) pattern of carry-propagation for the *6:2* and *7:2* compressor.

addition, using the four *3*-LUTs configured as a carry-save adder. The novel features of the new logic cell are the carry chain in the center (gray background), which can implement a *6:2* or *7:2* compressor, and the two multiplexers shown in gray on the right-hand side of Figure 10(a), which selects between the outputs of the two carry chains.

Similar to ternary addition, the new carry chain requires the four *3*-LUTs to be configured as a carry-save adder. To implement a *7:2* compressor, three FAs (and a 7^{th} LUT input) are required; to implement a *6:2* compressor, one of the FAs (outlined with a dashed line) becomes a half (two-input) adder, and the 7^{th} input bit is not used.

Three carry-in/carry-out bits are also required; they are labeled X, Y, and Z in Figure 10(a). The carry-out labeled $X/Y/Z$ connects to the corresponding carry-in labeled $X/Y/Z$ of the next compressor in the chain. A detailed picture of the carry chains across several logic cells is shown in Figure 10(b).

In principle, the FAs used in the two carry chains could be shared; this design choice was illustrated by Parandeh-Afshar et al. [2008a, Figure 5(a)]; although doing this could slightly reduce area, it requires that multiplexers be inserted into the carry chain, significantly increasing the critical path delay; as our goal is to increase performance, this design point is nonideal, especially since the area of the multiplexers offsets the area savings from sharing FAs.

There are two primary advantages of providing an FPGA logic cell that can be configured as a compressor compared to synthesizing GPCs on LUTs. The first advantage, which was illustrated in Figure 6, is that a *k:2* compressor will have a higher compression ratio than a $k-$input GPC.

In some but certainly not all cases, this can reduce the number of levels of logic in the compressor tree. The second advantage involves area utilization. Each ALM contains two 6-LUTs with dependent inputs. A GPC with six inputs and three outputs, including a *6:3* counter, requires two ALMs, while only one of our proposed logic cells, which is marginally larger than an ALM, is required to realize a *6:2* compressor. Reducing the number of logic cells, moreover, may allow for a tighter placement of logic cells on the device, which, in turn, reduces wirelength and routing delay; our experiments confirm this hypothesis. Using similar reasoning, the use of *7:2* rather than *6:2* compressors further increases the compression ratio, and may also reduce the number of logic cells required since each cell can consume an additional bit.

Consider the i^{th} compressor in the chain. Carry-in bits $c_{in,0}$ and $c_{in,1}$ are driven by the the *rank 1* carry-out of the $(i-1)^{st}$ compressor and the rank *2* carry-out of the $(i-2)^{nd}$ compressor, respectively; likewise, the rank *1* and carry-out of the i^{th} compressor drives carry-in, $c_{in,0}$, of the $(i+1)^{st}$ compressor, and the rank *2* carry-out drives carry-in, $c_{in,1}$, of the $(i+2)^{nd}$.

When an ALM is configured as a two-bit ternary adder in shared arithmetic mode, six input bits are used, so no modifications to the I/O interface are required to implement a *6:2* compressor. The *7:2* compressor, in contrast, requires an extra input bit. This is not a problem, as the ALM contains eight architecturally visible inputs; either of the two remaining inputs can be used as the seventh input when the ALM is configured as a *7:2* compressor.

5. COMPRESSOR TREE SYNTHESIS ON THE NEW LOGIC CELL

This section describes a mapping heuristic that can synthesize compressor trees targeting the logic cell shown in Figure 10(a). This heuristic is an extension of an earlier one proposed by Parandeh-Afshar et al. [2008b], which targeted the Altera Stratix II FPGA.

Compressor trees synthesized using an ASIC design flow produce two outputs that are summed using a CPA. Since ternary CPAs are available in Stratix II for the same delay and area as binary CPAs, the heuristic outputs compressor trees that produce three outputs instead of two. The remainder of the

compressor tree is synthesized using GPCs. The number of outputs per GPC was limited to four, ensuring that each GPC can be implemented using at most four *6*-LUTs (or fewer, if shared arithmetic mode can be exploited). This section extends the mapping heuristic to include the possibility of configuring the logic cells as *6:2* or *7:2* compressors as well.

5.1 GPC Classification

By convention, we require that a GPC must have at least *2*-input bits. For example, *(0, 1; 1)* and *(1, 0; 2)* are not GPCs.

Some GPCs are considered *unreasonable* by the heuristic because they can always be replaced with another more sensible choice. GPCs, such as *(3, 1; 3)*, have one rank *0* input bit, which is always passed directly to the least significant output bit, that is, the value of the input bit determines whether the output is odd/even; such a GPC is considered to be unreasonable. Another class of unreasonable GPCs are those for which the number of input bits is less than or equal to the number of output bits, for example, *(2, 1; 3)*; these GPCs are unreasonable because they do not perform any compression.

A third class of unreasonable GPCs are those that have no rank *0* input bits, for example, *(2, 0; 3)*. In this case, the *rank 1* input bits could be converted to rank *0* input bits of a smaller counter that produces fewer output bits, for example, *(0, 2; 2)*.

A *primitive* GPC is one that satisfies input/output constraints of M and N and is reasonable. In theory, the number of primitive GPCs is exponential in M and N; limiting M and N to small constant values ensures tractability. With N output bits, the sum, where input bits are weighted by rank, of the input bits cannot exceed $2^N - 1$; this ensures that the number of primitive GPCs is finite.

A *covering* GPC is a primitive GPC whose functionality cannot be implemented by another primitive GPC. For example, a *(2, 3; 3)* GPC can implement a *(1, 3; 3)* GPC by setting one rank *1* input bit to zero. For example, there are just three covering GPCs having six inputs and three outputs: *(0, 6; 3)*, *(1, 5; 3)*, and *(2, 3; 3)* (see Figures 8 and 9). All other GPCs satisfying these I/O constraints are either unreasonable, for example, *(3, 1; 3)*, or can be covered by one of the three covering GPCs already listed.

5.2 GPC Library Construction

The mapping heuristic uses a library of GPCs having at most M inputs and N outputs. This library is computed once for each target FPGA and stored in a text file. The library is read from the text file each time a set of compressor trees are synthesized.

First, the primitive GPCs are enumerated and added to the library. Second, the set of covering GPCs are identified and marked as such.

Third, the primitive GPCs are sorted in nondecreasing order of compression ratio. Each set of primitive GPCs having the same compression ratio is sorted in nondecreasing order of the number of inputs. The total ordering of primitive

GPCs favors a high compression ratio as the first criterion and the number of bits consumed as a second.

Parandeh-Afshar et al. [2008b] used $M = 6$ and $N = 4$ to target the Altera Stratix II FPGA. Limiting the number of inputs to $M = 6$ ensures that only one layer of ALMs is required to implement the counter, regardless of whether the GPC is synthesized on LUTs or uses shared arithmetic mode, that is, Figures 8 and 9. Limiting the number of outputs to $N = 4$ ensures that at most four ALMs are required for each GPC, under the worst case assumption that each output bit is computed using a *6*-LUT; fewer ALMs are required when shared arithmetic mode can be used [Parandeh-Afshar et al. 2009].

The mapping heuristic, described in the following section, converts chains of consecutive *(0, 6; 3)* GPCs (*6:3* counters) into *6:2* compressors, whenever possible. Unfortunately, this approach cannot be used for *7:2* compressors, as $M = 6$ prevents *7*-input GPCs from inclusion in the library. To support *7:2* compressors, a *(0, 7; 3)* GPC is added to the library, but no other *7*-input GPCs are included. Chains of *(0, 7; 3)* GPCs are converted to *7:2* compressors; when a *7*-input GPC is not contained in a chain, it is converted to GPCs with at most *6* inputs, as described in the following section.

5.3 Mapping Heuristic

The input to the mapping heuristic is: (1) an ordered array of integers, k_i, where the i^{th} integer is the number of bits of rank i to sum, e.g., k_0 bits of rank 0, k_1 bits of rank *1*, etc.; (2) a library of GPCs, as described in the preceding section; and (3) a flag called *mode* which takes one of three values, *ALM*, *6:2*, or *7:2*. If *mode* = *ALM*, then we are targeting an FPGA containing traditional ALMs that cannot be configured as *6:2* or *7:2* compressors; if *mode* = *6:2* or *7:2*, then we are targeting an FPGA whose logic cells can be configured as a *6:2* or *7:2* compressor, for example, Figure 10(a).

The mapping heuristic generates one level of the compressor tree at a time. A subset of the input bits is covered by GPCs and possibly *6:2* or *7:2* compressors. The output bits produced by each GPC are propagated to the next level of the compressor tree, along with the bits from the current level that are not covered. Since the rank of each GPC output bit is known, a new set of columns (array of integers) is generated for each level of the tree.

Pseudocode for the mapping heuristic is shown in Figure 11. A new level in the tree is generated until there are at most three rows of bits remaining, that is, each column of the next level has at most three input bits. A ternary CPA completes the tree. The remainder of this section focuses on the process of producing one level of the compressor tree, that is, how to cover a set of columns with GPCs. The following process is applied until no remaining (primitive) GPCs can cover any bits in the current level of the tree.

The column having the most noncovered input bits in the current level is always selected; ties are broken arbitrarily. Selecting the column with the largest number of bits tends to favor the use of GPCs with higher compression ratios and a large number of input bits. To find the best GPC for the selected

```
GPC_mapping(Integer : M, Integer : N, Array of Integers : columns, GPC Library : L,
                {ALM, 6:2, 7:2} : mode  )
{
// Local Variables
Integer : col_index
GPC: gpc1, gpc2

While( more than three rows of bits remain )
{
    Do {
        col_index ← index of largest column
        GPC : gpc1 ← forward_search(col_ index)
        GPC : gpc2 ← backward_search(col_ index)
        If ( gpc1 ≠ NULL or gpc2 ≠ NULL ) {
            Select higher priority GPC between gpc1 and gpc2
            Remove from columns the bits covered by the higher priority GPC
        }
    } While( gpc1 ≠ NULL or gpc2 ≠ NULL )

    // Extensions to handle 6:2 and 7:2 compressors
    If( mode == 6:2 ) {
        Remove contiguous sets of 6:3 counters and replace them with 6:2 compressors
    } Else If ( mode == 7:2 ) {
        Remove contiguous sets of 7:3 counters and replace them with 7:2 compressors
        Replace remaining 7:3 counters with 6:3 counters and leave one bit unmapped
    }

    Connect inputs of current-level GPCs to the outputs of preceding level GPCs
    Generate columns for the next level of the compressor tree from current level GPC
        and compressor outputs and unmapped bits.
}
```

Fig. 11. Pseudocode for GPC mapping heuristic [Parandeh-Afshar et al. 2008a] with extensions to exploit *6:2* and *7:2* compressors, where appropriate.

column, the set of primitive GPCs is searched. The first GPC that fits the base columns and its following or previous columns is selected.

If *mode = 7:2* and the column contains at least seven input bits, then a *(0, 7; 3)* GPC is always used, and a *(0, 6; 3)* GPC is always used if the column contains six input bits.

If *mode = 6:2* or *ALM* and the column contains at least six input bits, then a *(0, 6; 3)* GPC is always used.

Otherwise, the selected column contains fewer bits than the maximum input bandwidth of the largest GPC in the library; in this case, GPCs that cover bits from columns that are immediately adjacent to the selected column can be used as well.

A *forward search* looks for a GPC under the assumption that the bits in the selected column will have rank *0*. If the selected column is c, then the forward search will attempt to include bits from columns $c + 1$, $c + 2$, ..., etc. A *backward* search assumes that the bits in the selected column will be of the

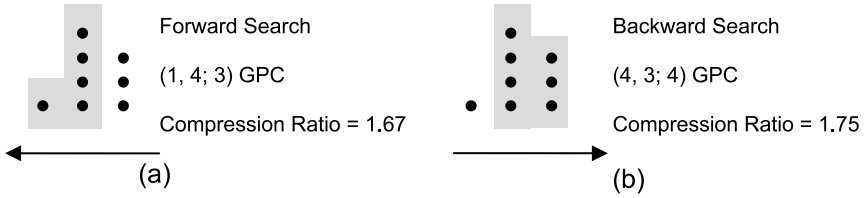


Fig. 12. Illustration of the forward (a) and backward (b) search using GPC mapping.

highest rank in the GPC that covers them. If the selected column is c, then the backward search will attempt to include bits from columns $c - 1$, $c - 2$, ..., etc. In both searches, the first GPC that fits the selected column and the additional columns are selected. Among these two GPCs, the one with the highest priority, according to the sorted order, is selected. The forward and backward searches are particularly useful when the distribution of column heights is asymmetric. This occurs quite frequently for constant multipliers, including FIR filters.

Figure 12 illustrates the forward and backward search. In Figure 12(a), a forward search finds a *(1, 4; 3)* GPC while the backward search in Figure 12(b) finds a *(4, 3; 4)* GPC. Since the compression ratios are 5/3 = 1.67 and 7/4 = 1.75, respectively, the GPC found by the backward search is selected.

After selecting a column and a GPC, the bits that have been selected are removed from the current set of columns. The output bits produced by the GPC are added to the set of columns for the next level in the tree. This process repeats—a column and GPC are selected—until either all bits at the current level have been covered, or no primitive GPC in the list can cover more than a single bit. Once the current level is completely covered, the heuristic attempts to replace some GPCs with *6:2* or *7:2* compressors.

If *mode* = *6:2*, each contiguous sequence of *(0, 6; 3)* GPCs is replaced with a contiguous sequence of *6:2* compressors, similar in principle to Figure 6. Note that this transformation reduces the number of bits in the following level; aggregated over several levels, the use of compressors rather than counters can reduce the total number of logic levels in the compressor tree.

If *mode* = *7:2*, then each contiguous sequence of *(0, 7; 3)* GPCs is replaced with a sequence of *7:2* compressors, just similar to what was done for *6:2* compressors. Each remaining *(0, 7; 3)* GPC is replaced by a *(0, 6; 3)* GPC and one unmapped bit that is propagated to the next level of the tree. The reason for doing this is that *(0, 7; 3)* GPCs do not map efficiently onto ALMs, so we replace them with a more favorable component.

Next, the current level of the compressor tree is mapped onto logic cells. GPCs are mapped onto ALMs, while *6:2* and *7:2* compressors require the logic cell to be configured to use the carry chain shown in Figure 10(a). Additionally, the outputs of the GPCs and compressors from the preceding level of the compressor tree are connected to the inputs of the GPCs and compressors in the current level. The last step is to generate the columns for the next level of the compressor tree.

5.4 The Final Carry Propagate Adder

The final carry-propagate adder (CPA) uses the carry chains that are present on modern high-performance FPGAs. In the case of the Altera Stratix II-IV series FPGAs, shared arithmetic mode permits the ALMs in the carry chains to be configured as ternary (3-input) CPAs with no additional cost over *2*-input CPAs. To exploit this device family-specific feature, the compressor tree produces three outputs, rather than two. The CPA itself is comprised of a carry-save adder (implemented in LUTs) followed by a ripple-carry adder (implemented using the carry chains).

6. EXPERIMENTAL SETUP

6.1 VPR

The publicly available *Versatile Place-and-Route (VPR)* tool [Betz and Rose 1997; Betz et al. 1999] was used to evaluate the new FPGA logic blocks proposed in Section 4. The algorithm in Section 5 was used to map each compressor tree onto the new FPGA. This determines the number of logic cells required to realize the circuit. VPR was then used to place and route the circuit; afterwards, VPR reported the critical path delay, including its decomposition into logic and routing delays, wirelength, and the minimum number of routing tracks per channel for which the design is routable.

VPR models an island-style FPGA, where each island is a cluster containing one (or more) *Basic Logic Elements (BLEs)*. Each BLE consists of a programmable LUT, a flip-flop connected to the LUT output, and multiplexer. The selection bit of the multiplexer is programmed so that it can select the LUT output for combinational logic or the flip-flop output for sequential logic. BLEs within the same cluster connect to each other by a fast local routing network. The global routing network, which is slower, connects BLEs in different clusters. The cluster in an Altera Stratix-series FPGA is called a *Logic Array Block (LAB)*, and contains several ALMs.

We used VPR version 4.30 to model logic blocks and logic clusters that resemble Altera's ALMs and LABs. Each LAB in our architecture contains four ALMs. Since VPR does not model carry chains between LABs, we model each carry chain output as being provided by an additional LUT inside the LAB, whose delay is specified appropriately. Another difference between VPR 4.30 and realistic FPGAs involves the routing network: the Stratix II-IV organizes LABs into columns with nonuniform routing in the $x-$ and $y-$ directions; the baseline VPR architecture, in contrast, has uniform routing.

We modeled a clone of the Altera ALM in VHDL, and added the extra carry chain and two multiplexers shown in Figure 10(a), along with one additional configuration bit, which is only set when the ALM is configured as a compressor; two different versions of the modified ALM were created, that respectively support configurations as *6:2* and *7:2* compressors. Using "shared arithmetic mode" (i.e., Figure 7), the ALM can be configured as a two-bit ternary adder; each LAB contains four ALMs, and can be configured as an eight-bit ternary

ripple-carry adder. The global routing network can be used to build larger ripple-carry adders.

The ALM clones were synthesized with *Synopsys Design Compiler* using a *90nm Artisan* standard cell library based on a *TSMC* design kit. The delays of the paths through the ALM clone were input into the VPR architecture configuration file to model the logic and carry chain delays. We estimated the size of each cell in terms of *2*-input gates; *22* additional gates were required to implement the carry chain for the *7:2* compressor, including the two multiplexors in Figure 10(a), and the extra configuration bit; this increased the area by less than *5%*. Figure 13 shows the delays of the output bits of the ALMs in a LAB; when configured as a *6*−LUT, the delay of each output is always *0.69 ns*; for other configurations, the delay depends on the position along the carry chain.

VPR generates an FPGA whose dimensions are sized specifically for each benchmark circuit. This tends to minimize the routing delays from the FPGA's input pads to the circuit inputs, and from the circuit outputs to the FPGA's output pads. The FPGA generated by VPR must have at least as many LABs as the packed circuit, and must satisfy an aspect ratio specified by the user. For example, if the user specifies an aspect ratio of 1, and the circuit requires *23* LABs, then the FPGA generated by VPR will be a *5x5* array. An aspect ratio of *1* was used throughout our experiments.

VPR uses a binary search to determine the minimum number of tracks for which a legal route can be found for each circuit. For a given placement, let t_x and t_y be the number of routing tracks used in the x and y directions, and let $t_z = max\{t_x, t_y\}$. VPR stops the binary search when it finds the minimum value of t_z for which a legal route is found.

To model routing delays, the per-unit resistance and per-unit capacitance of the wires must be specified in the VPR architecture configuration file. We selected per-unit resistance and per-unit capacitance values based on the *TSMC 90nm* CMOS technology, under the assumption that metal-6 is used for wires.

6.2 Packing

Technology mapping for FPGAs maps a circuit implemented in terms of basic gates (e.g., AND, OR, XOR, etc.) onto appropriate FPGA components: LUTs, carry chains, DSP blocks, etc. The compressor tree synthesis heuristic described in Section 5 is a form of technology mapping that is specific to compressor trees mapped onto ALMs that have been modified as shown in Figure 10(a).

Packing is the process of assigning each ALM in a technology mapped netlist to exactly one LAB. The number of ALMs assigned to the same LAB cannot exceed the maximum number of ALMs per LAB, which is an architecture-specific parameter.

Technology mapped ALMs that are connected by a carry chain must be mapped to the same LAB; otherwise, the carry chains cannot be used.

VPR 4.30 includes a packing tool called *T-VPack*; as VPR 4.30 does not support BLEs with carry chains, T-VPack cannot enforce the constraint described earlier, because it is unaware of the presence of carry chains.

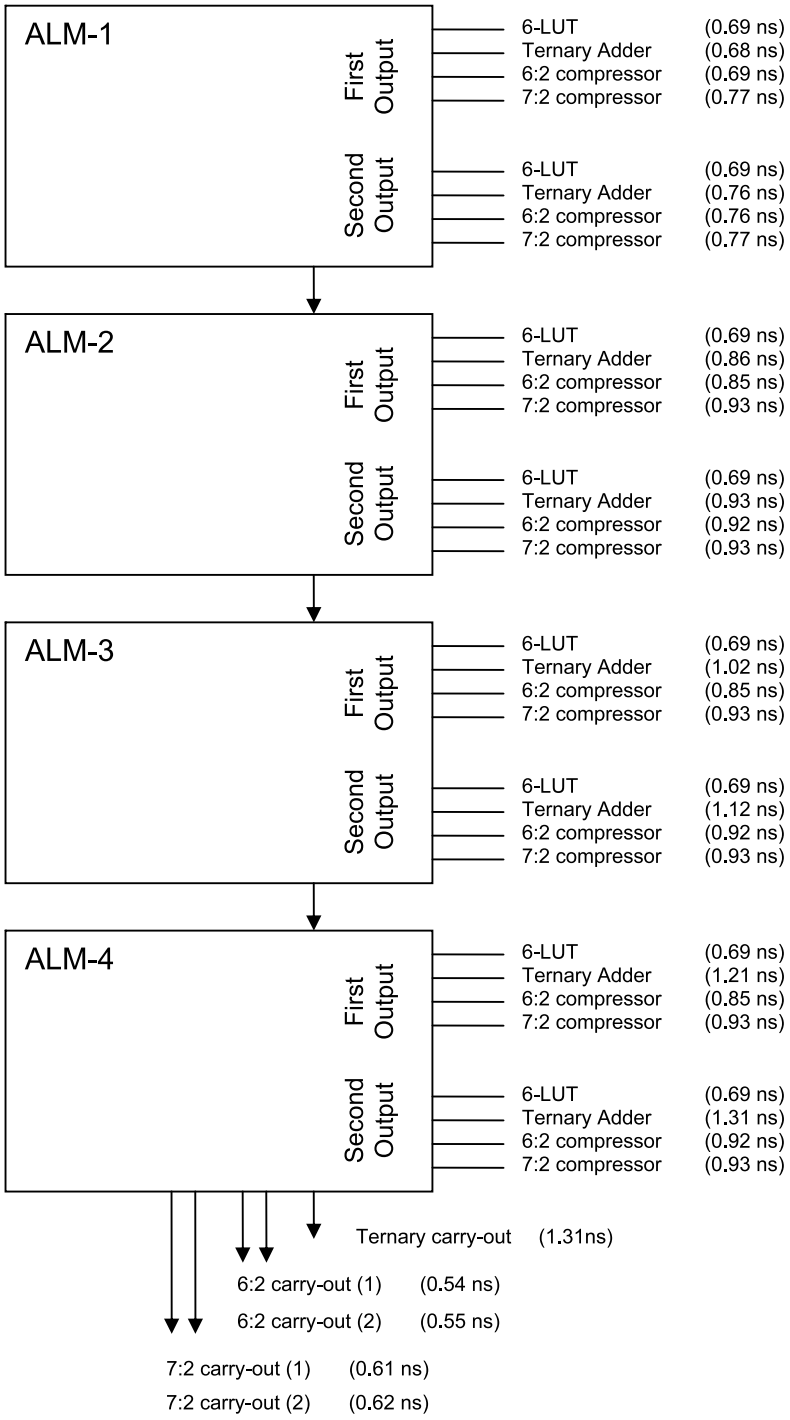


Fig. 13. Combinational delays of the ALM outputs in a LAB, including propagation delays along the carry chains.

Table I. Benchmark Summary

Benchmark	Description
m12x12, m16x16	Parallel multipliers
FIR3, FIR6	3- and 6-tap FIR filters
mac	Multiply-accumulate (a x b + c)
samul	8-bit Shift-and-add multiplier
g72x	G.721 encoder [Lee et al. 1997]
H.264 ME	H.264 motion estimation
Video Mixer	Parallel RGB-YIQ conversion mixed with an alpha blender (add2I, add2Q, add2Y, RQGQBQ, RYGYBY)

Instead of using T-VPack, we wrote our own packing software that is specific to the compressor trees produced by our mapping heuristic. The packer is greedy: it always selects the longest carry chains, and packs up to four ALMs along the chain into the same cluster. If the length of the chain is $k > 4$, then the first four ALMs in the chain are packed together, and the chain is broken after them; this yields a new chain of length $k - 4$, which is reinserted into the set of carry chains. When no carry chains remain, the remaining ALMs are packed arbitrarily. After packing, VPR performs placement and routing.

6.3 Benchmarks

A set of arithmetic, DSP, and video processing benchmarks containing compressor trees were selected for synthesis; only the compressor trees and final adders from these benchmarks were synthesized. Table I summarizes the benchmark circuits.

*m*12*x*12 and *m*16*x*16 are parallel multipliers; *FIR3* and *FIR6* are FIR filters built using the add-and-shift method [Mirzaei et al. 2006]; *mac* is a multiply-accumulator, and *samul* is an 8-bit shift-and-add multiplier. *g72x* is taken from *g721*, a *Mediabench* [Lee et al. 1997] application; *H.264 ME* is the variable block size motion estimation phase of *H.264/AVC* video encoding [Chen et al. 2006]. The remaining five circuits (*add2I*, *add2Q*, *add2Y*, *RQGQBQ*, and *RYGYBY*) are compressor trees occurring within a larger application called *Video Mixer*, which is included with *Synopsys Corp.*'s *Behavioral Optimization of Arithmetic (BOA)* tool. Video mixer, in particular, benefits significantly from the transformations of Verma et al. [2008], which expose many large compressor trees.

7. EXPERIMENTAL RESULTS

7.1 Overview of Experimental Comparison

Throughout our experiments, each compressor tree was synthesized four times. Table II summarizes the four different approaches: *Ternary*, *GPC*, *6:2+GPC*, and *7:2+GPC*. As described in Section 5, each compressor tree produces three outputs that are summed with a ternary ripple carry adder.

The *Ternary* and *GPC* synthesis methods target high-performance FPGAs that contain 6-LUTs, carry chains, and support for ternary addition; this

Table II. Description of the Four Synthesis Methodologies Used in the Experiments

Synthesis Method	Description
Ternary	Ternary Adder Tree
GPC	GPC mapping using the algorithm in Figure 11 in ALM mode.
6:2+GPC	GPC mapping using the algorithm in Figure 11 in 6:2 mode.
7:2+GPC	GPC mapping using the algorithm in Figure 11 in 7:2 mode.

includes the *Altera Stratix II-IV*, as well as the *Xilinx Virtex 5*. *6:2+GPC* and *7:2+GPC* target FPGAs containing the modified ALM in Figure 10(a).

The heuristic used to build ternary adder trees is similar in principle to the GPC mapping strategy introduced in Section 5.3; the primary difference is that the component library contains just one component: a ternary adder, that is, a *3:1* CPA. When there are several choices of input bits to add at a level of the tree, then priority is given to the widest possible CPA with the longest carry chain.

Parandeh-Afshar et al. [2008b] have already compared *Ternary* and *GPC* on Altera Stratix II FPGAs. *GPC* yielded compressor trees with faster critical path delay; however, these compressor trees required considerably more ALMs. Similar trends are observed here. The experiments presented here evaluate the benefit of extending these logic cells to be configurable as a *6:2* or *7:2* compressor. As discussed in the preceding section, we estimate that this proposed logic cell is at most *5%* larger than the *ALM* used in Stratix II-IV FPGAs. The benefits obtained using the new logic cell, as reported here, must be weighed against a uniform increase in the area of *all* logic cells in the FPGA, including a great many that will not be configured as a *6:2* or *7:2* compressor for any specific circuit.

7.2 Critical Path Delay

First, we measure the critical path delay for each benchmark and decompose it into logic delays within the compressor tree and final CPA, and routing delays. We synthesized each benchmark ten times using VPR, using a different random number seed each time. Figure 14 shows the average critical path delay of the ten runs, including a *95%* confidence interval for each benchmark and synthesis method.

The critical path delays of *Ternary* and *GPC* were comparable (i.e., overlapping confidence intervals) for most benchmarks; there are some exceptions, e.g., *g72x* where *GPC* was definitely faster than *Ternary*, and *mac* and *samul* where *Ternary* was definitively faster than *GPC*. *H.264 ME*, *mac*, and *samul* are the smallest benchmarks with the shallowest compressor trees; there is little differentiation between the results of the compressor tree synthesis methods for these three benchmarks. In most cases, the critical path delays observed for *6:2+GPC* and *7:2+GPC* were comparable; however, there are notable cases, such as *add2I*, *add2Y*, and *g72x* where there is significant differentiation in favor of one or the other. On average, *6:2+GPC* and *7:2+GPC* offer a definitive advantage over *Ternary* and *GPC*.

Figure 15 decomposes the average critical path delay into percentages due to logic (including carry chains) and routing. Typically, logic delays consumed

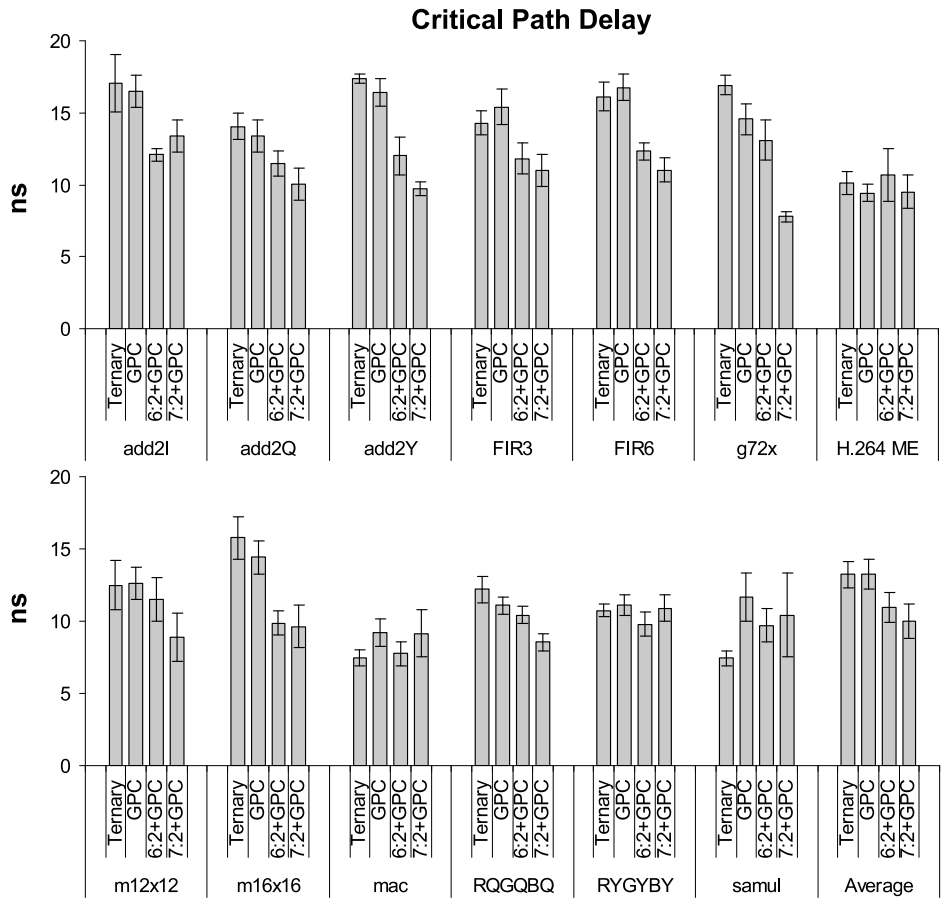


Fig. 14. The critical path delay for each benchmark and compressor tree synthesis methodology, shown with a 95% confidence interval.

30-45% of the overall delay, for each benchmark and synthesis method. For each benchmark, *Ternary* had the highest percentage of logic delay, which can be attributed to the use of carry chains at each level of the tree; taken in aggregation, the carry chains within the adder tree start from the least significant input bit to the most significant output bit of the final CPA, although the critical path is not guaranteed to include the final CPA.

In contrast, *GPC*, *6:2+GPC*, and *7:2+GPC*, include logic delays through some portion of the compressor tree, followed by some, but not all, of the final CPA. On average, the critical path of *GPC* subsumed a slightly greater percentage of routing delay than *6:2+GPC* and *7:2+GPC*, however, this trend did not occur uniformly across all benchmarks.

Among the ten runs for each benchmark and synthesis method, the standard deviation of the logic delays was always nonzero. Along any specific path through the circuit, the logic delay will always be fixed, but the routing delay differs, depending on the placement. The nonzero standard deviations of

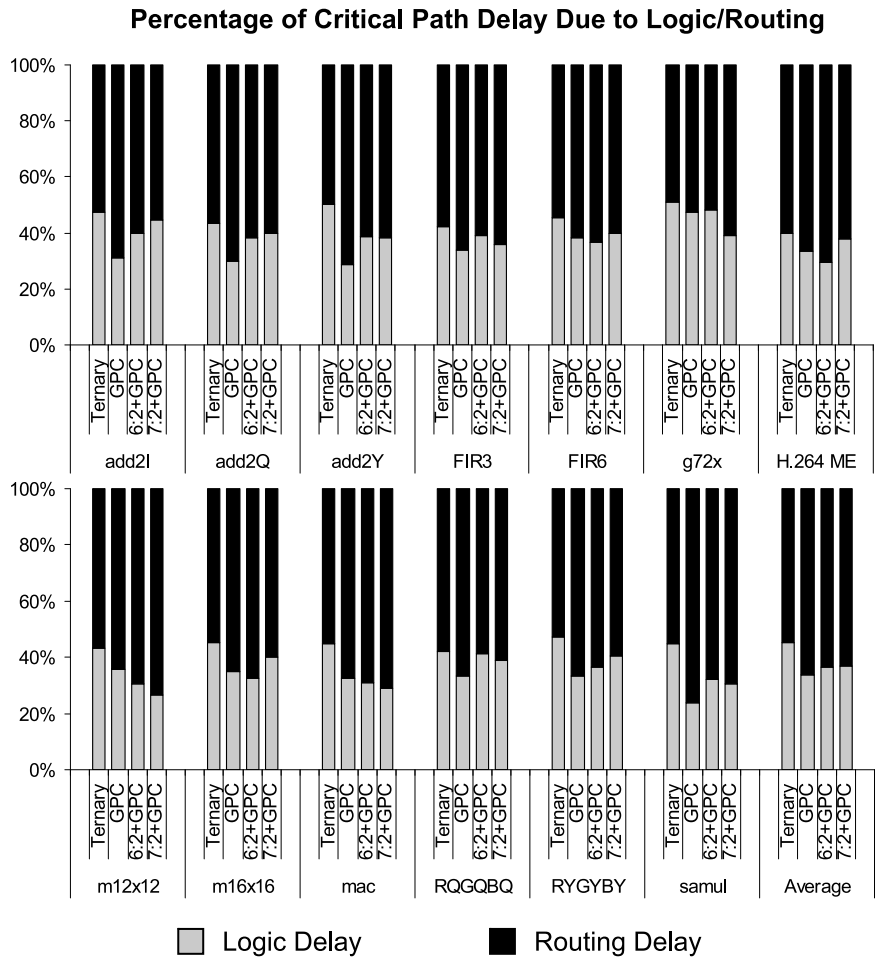


Fig. 15. On average, the percentage of critical path delay due to logic and routing for each benchmark.

the logic delays indicate that random changes in the placement, that is, variation in routing delays due to different random number seeds used by VPR, can change which paths become critical.

In ASIC technologies, the arrival time of each compressor tree output bit at the CPA can be predicted from synthesis [Oklobdzija and Villeger 1995]; the nonzero standard deviations for logic delays in our results indicate that for FPGA design flows, variations in routing delays make this process inexact. Hence, methods to simultaneously optimize the compressor tree output delay profile and the final CPA design for FPGAs, that is, those that are analogous to Oklobdzija and Villeger's, are unpredictable prior to placement and routing due to variations in routing delay. Moreover, due to the specific features of FPGA logic architectures, including carry chains, the hybrid final

CPAs proposed by Oklobdzija and Villeger may be an inappropriate choice for FPGAs. For this reason, we chose to implement the final CPAs using ripple-carry chains. Alternative CPAs, such as carry-select adders, will require more ALMs. Although they are superior in terms of logic delay, they may be inferior when the additional costs of routing are taken into account. A detailed investigation into final CPA design for compressor trees in FPGA technologies is left open for future work.

7.3 Critical Path Analysis

For *GPC*, *6:2+GPC*, and *7:2+GPC*, the logic delay includes a few layers of logic blocks (LB layers) in the compressor tree, followed by some number of bits in the CPA. As the specific critical path varies from run to run, we focus on individual runs. In particular, we select the minimum critical path among the ten runs for each benchmark and synthesis method for an analysis of the components that contribute to logic delay. This analysis does not make sense for *Ternary*, because the critical paths in the adder tree may include long carry chain delays at each level of the tree, not just the final CPA.

Figures 16 and 17 decompose the critical path delay into logic delays within the compressor tree and CPA, and routing delays, for *GPC*, *6:2+GPC*, and *7:2+GPC*; they also annotate the compressor tree logic delay with the number of LB layers, and the final CPA logic delay with the number of final CPA bits on the critical path.

6:2 and *7:2* compressors can lead to different phenomena that impact critical path delay. For *add2I*, for example, the use of *6:2* and *7:2* compressors reduces the LB layers on the critical path in the compressor tree, but increases the number of bits on the critical path in the CPA. In this case, the logic delay of *6:2+GPC* exceeds that of *GPC*, and the critical path delay of *7:2+GPC* exceeds that of *6:2+GPC*; the overall delays of *6:2+GPC* and *7:2+GPC*, however, are equalized when routing delays are considered.

For *add2Q*, the critical path of *6:2+GPC* goes through the same number of LB layers in the compressor tree as *GPC*, and three more bits in the final CPA. *6:2+GPC* incurs a greater logic delay than *GPC*, but a smaller critical path delay due to routing. *7:2+GPC*, reduces the number of LB layers in the compressor tree from three to one, but incurs a greater delay through the CPA than the *GPC* or *6:2+GPC*. The logic delay of *7:2+GPC*, is comparable to that of *GPC*, while its routing delay is comparable to *6:2+GPC*.

add2Y is similar to *add2I*. Here, the use of *6:2* and *7:2* compressors reduce the number of LB layers on the critical path in the compressor tree from four (*GPC*) to three (*6:2+GPC*) to two (*7:2+GPC*). The number of CPA bits on the critical path is reduced from twelve (*6:2+GPC*) to eleven (*7:2+GPC*), while routing delays are comparable.

For *FIR3*, the critical path of *6:2+GPC* passes through the same number of LB layers in the compressor tree as *GPC*, but four fewer bits of the CPA; the critical path of *7:2+GPC*, in contrast, passes through fewer LABs in the compressor tree and fewer final CPA bits than either *GPC* or *6:2+GPC*. A similar trend is observed for *m12x12*.

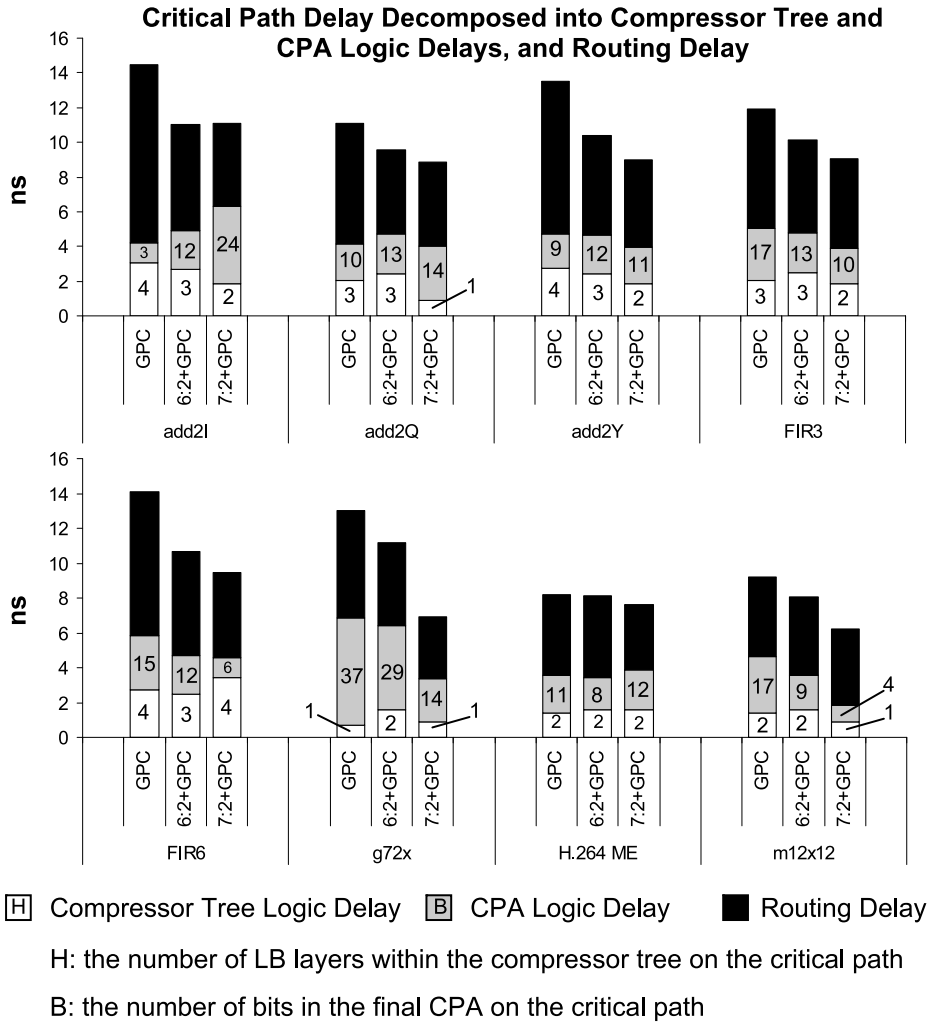


Fig. 16. The minimum critical path for each benchmark and synthesis method, decomposed into logic delays within the compressor and CPA, and routing delay.

For *FIR6* and *RQGQBQ*, the critical path of *6:2+GPC* goes through fewer LB layers in the compressor tree compared to *GPC* and *7:2+GPC*. The critical path of *6:2+GPC* goes through fewer bits of the final CPA than *GPC*, while the opposite is true for *RQGQBQ*. In both cases the critical path of *7:2+GPC* includes fewer bits in the final CPA, and significant reductions in routing delay, compared to *GPC* and *6:2+GPC*.

Among all benchmarks, *g72x* has the largest final CPA bitwidth, *39*, and would thus be the most likely to benefit from techniques that can synthesize faster CPAs than ripple-carry adders. Its critical path includes one LB layer in the compressor tree, but *37* bits of the CPA. *6:2+GPC* goes through two LB layers in the compressor tree, but just *29* bits of the final CPA. *7:2+GPC*, in

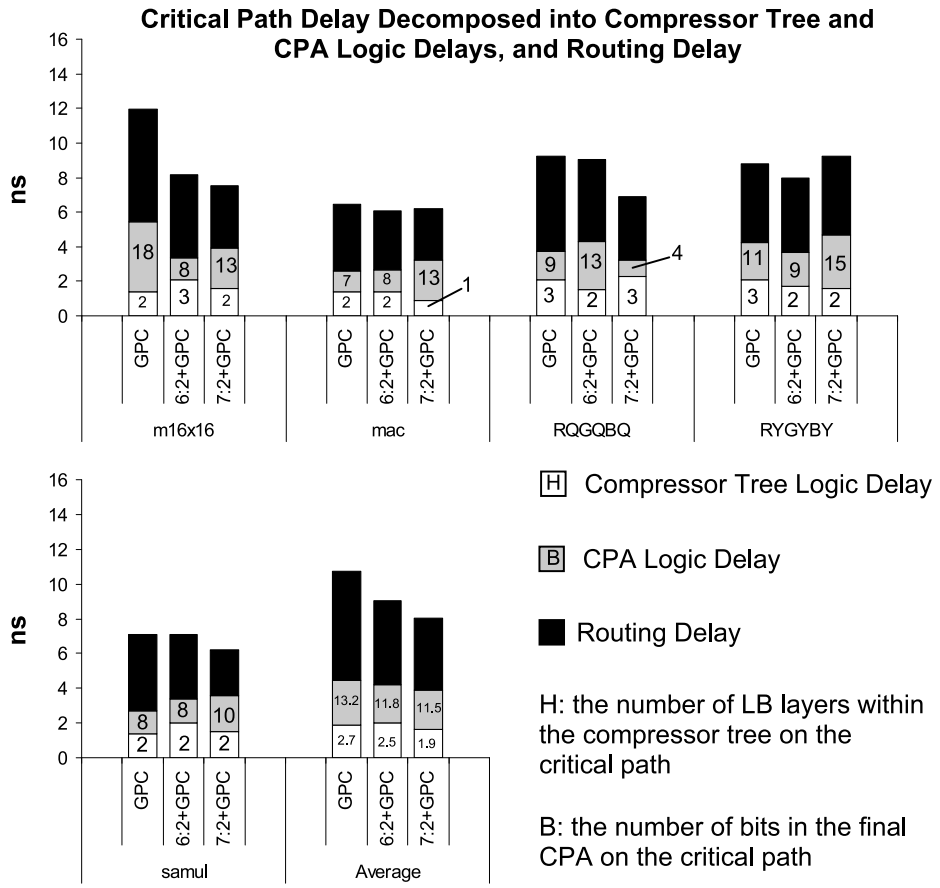


Fig. 17. The minimum critical path for each benchmark and synthesis method, decomposed into logic delays within the compressor and CPA, and routing delay.

contrast, achieves a far superior reduction in logic delay, as its critical path goes through just one LAB in the compressor tree, and fourteen bits in the final CPA, and also benefits from the smallest routing delay as well.

H.264ME, *mac*, and *samul* are the smallest benchmarks evaluated here. As shown in Figure 14, *Ternary* actually achieves the best critical path delays, while Figure 15 shows that this is because routing delays account for a smaller fraction of the overall critical path delay for *Ternary* than *GPC*, *6:2+GPC*, and *7:2+GPC*. The critical path of *7:2+GPC* goes through more bits in the final CPA than *GPC* and *6:2+GPC*, and this tends to dominate the logic delay. For *H.264 ME* and *samul*, *7:2+GPC* benefits from a lower routing delay than *6:2+GPC* and *GPC*; for *mac*, the routing delays are comparable.

For *m16x16*, the critical path of *6:2+GPC* goes through more LB layers in the compressor tree than *GPC* and *7:2+GPC*; however, its logic delay advantage occurs because its critical path includes fewer bits in the final CPA. *7:2+GPC* achieves the smallest critical path delay due to reduced routing delay.

For *RYGYBY*, *7:2+GPC* has a larger critical path delay than *GPC* or *6:2+GPC*. Compared to *GPC*, its critical path goes through fewer LB layers in the compressor tree, but more bits in the final CPA, giving it a larger logic delay. Compared to *6:2+GPC*, the critical path of *7:2+GPC* goes through the same number of LB layers in the compressor tree, but more bits in the final CPA. In all three cases, the routing delays are comparable. *6:2+GPC* has the minimum critical path, as its critical path goes through fewer LB layers in the compressor tree than *GPC*, and the smallest number of bits in the CPA.

On average, the critical path of *GPC* goes through 2.7 LB layers in the compressor tree and 13.2 bits in the CPA; the critical path of *6:2+GPC* goes through 2.5 LB layers in the compressor tree and 11.8 bits of the final CPA; and *7:2+GPC* goes through 1.9 LABs in the compressor tree and 11.5 bits in the final CPA. On average, the routing delay of *GPC* is *6.3 ns*, the routing delay of *6:2+GPC* is 4.9ns, and the routing delay of *7:2+GPC* is 4.2ns. Altogether, the reductions in routing delay tend to have a greater impact on critical path delay that differences in logic delay.

To summarize, the benefits of the *6:2* and *7:2* compressors, in terms of logic delay, vary from benchmark to benchmark; there is no uniform or universal answer. The logic delay of the compressor tree may increase or decrease compared to other methods; the same is also true for the delay through the CPA. At no point, however, do both the compressor tree and CPA logic delays increase for *6:2+GPC* and *7:2+GPC* over *GPC*. *6:2+GPC* and *7:2+GPC* also retain advantages in terms of routing delay compared to *GPC*; the reasons for these advantages will be exposed in the following two subsections.

7.4 Area Utilization

Figure 18 shows the area (number of LABs) required for each benchmark. In general, *Ternary* achieve the smallest area, because ALMs in shared arithmetic mode have a compression ratio of *3*, whereas, *6*-input, *3*-output GPCs have a compression ratio of *2*, while requiring two ALMs. Although *6:2* and *7:2* compressors have compression ratios of 3 and 3.5 respectively, the use of GPCs in addition to the compressors causes *6:2+GPC* and *7:2+GPC* to use more ALMs than *Ternary*. *GPC*, consequently, requires the most area uniformly across the benchmark suite.

For *add2Q*, *FIR3*, and *H.264ME*, *6:2+GPC* required fewer ALMs than *7:2+GPC*. This occurs because the mapping heuristic in Figure 11 converts a *7:3* counter to a *6:3* counter and one unmapped input bit when a chain of *7:2* counters cannot be found. A more complicated mapping heuristic that backtracks when this occurs, and tries to cover the unmapped bit with a GPC at the current level of the compressor tree, rather than at a later level, is likely to achieve comparable results to *6:2+GPC*.

Figures 16 and 17 showed that *GPC* tends to have larger routing delays than *6:2+GPC* and *7:2+GPC*. Now, Figure 18 shows that *GPC* tends to require more LABs than *6:2+GPC* and *7:2+GPC* as well. This suggests that the primary cause for *GPC*'s higher routing delay is due to area utilization. Each *GPC*

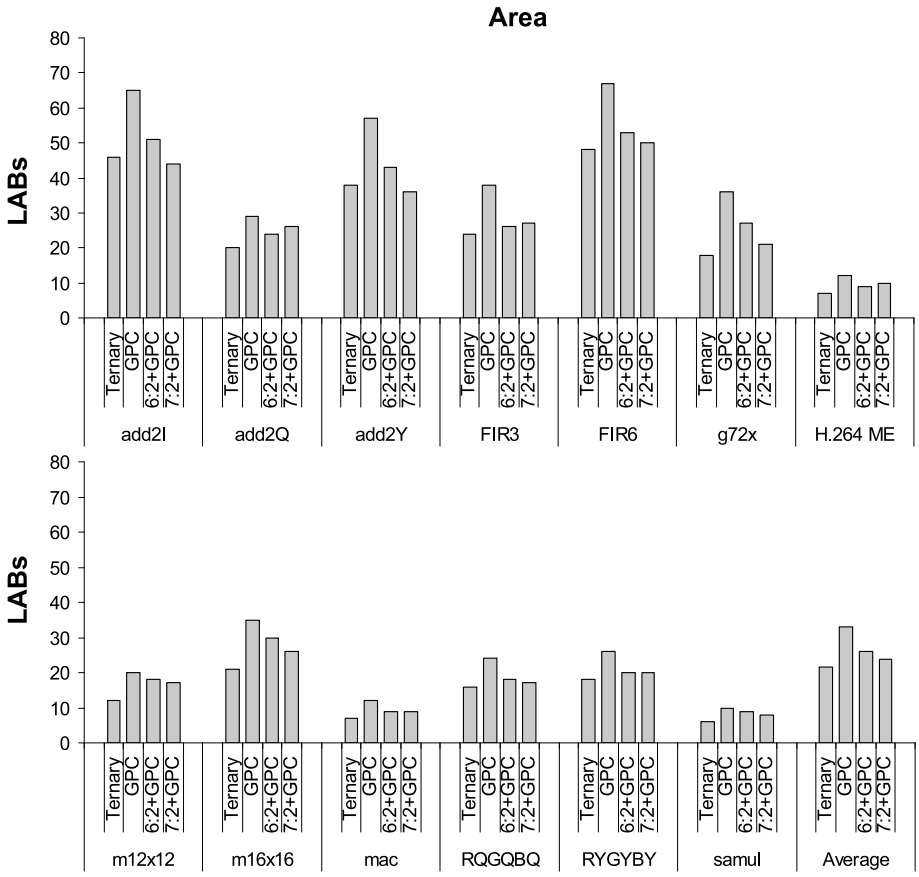


Fig. 18. The area (LABs) required for each benchmark and compressor tree synthesis method.

requires two ALMs, while each *6:2* or *7:2* compressor requires just one. Consequently, the use of compressors instead of GPCs tends to reduce the number of ALMs used in a design. This, in turn, leads to a tighter placement, which tends to reduce wirelength. As each wire crosses through fewer switch and connection boxes, routing delays tend to reduce as well.

Lastly, each wire that connects to a GPC has a higher fanout than a wire connecting to a *6:2* or *7:2* compressor, as multiple ALMs are required to implement the GPC. This is another reason for the wirelength reduction shown in the next section.

7.5 Wirelength and Routability

This section compares and contrasts *Ternary*, *GPC*, *6:2+GPC*, and *7:2+GPC* in terms of wirelength and routability. Our VPR architecture configuration contains a mixture of segments of different lengths. In the architecture we studied, *90%* of wires have span two LABs, while the remaining *10%* span four LABs; buffered routing switches were always used.

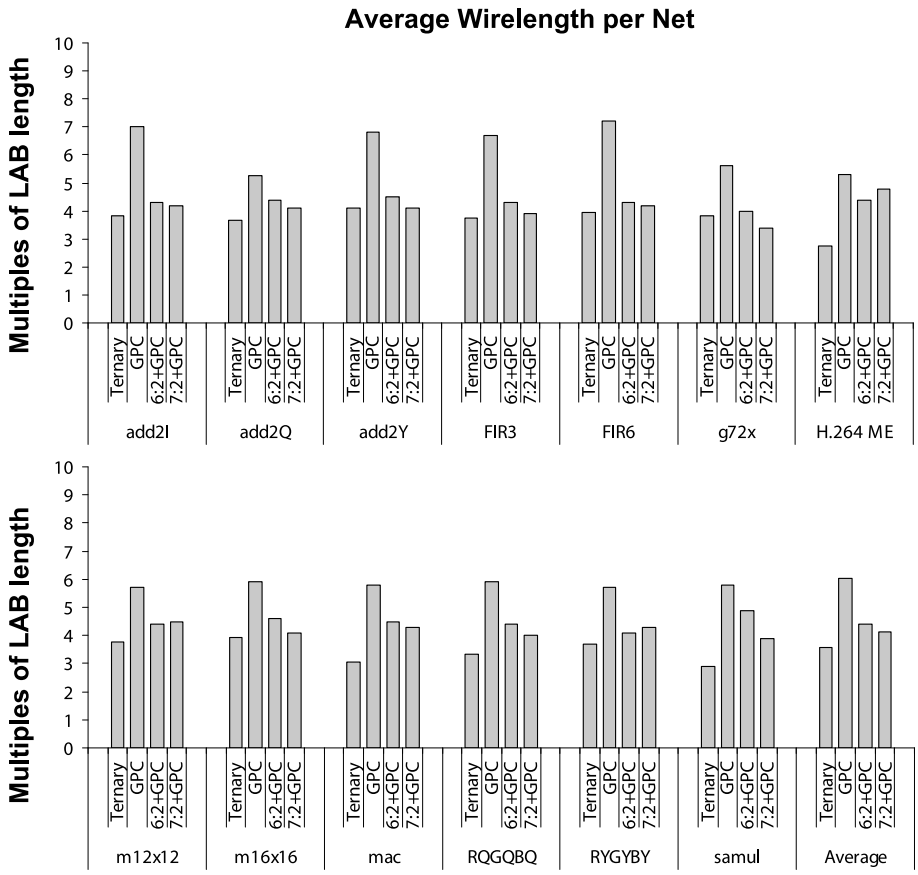


Fig. 19. Average wirelength per net for each benchmark and compressor tree synthesis method.

Figure 19 reports the average wirelength per net for each benchmark and synthesis method. The wirelength reported in Figure 19 accounts for the varying lengths of the different segments.[1]

For each benchmark, the average net wirelength of *GPC* was larger than that of *Ternary*, *6:2+GPC*, and *7:2+GPC*. Figure 18 has shown that *GPC* requires more LABs than *6:2+GPC* or *7:2+GPC*; Figure 19 shows that the tighter packing achieved by *6:2+GPC* and *7:2+GPC* is able to reduce the average net wirelength, which in turn, reduces the overall critical path delay.

Admittedly, Figure 19 does not compare the wirelengths on the critical path; however, routing delay can affect which paths are critical, as discussed in Subsections 7.2 and 7.3. Thus, it is reasonable to assume that the critical path will have longer wires for *GPC* compared to *6:2+GPC* and *7:2+GPC* for each benchmark, as the average net wirelength of *GPC* tends to be longer as well.

[1]VPR also reports the number of physical segments used without accounting for the lengths of the connections; this metric is not reported here.

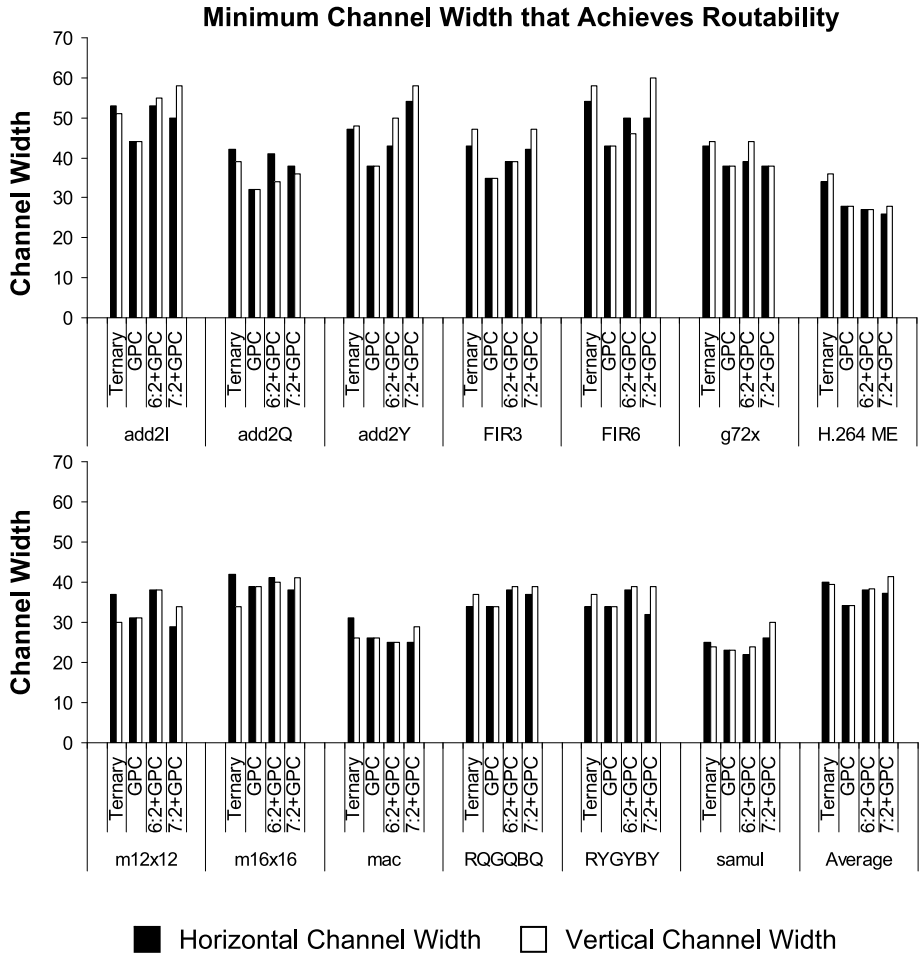


Fig. 20. The minimum channel width in the x and y directions for which each benchmark is routable for each compressor tree synthesis method.

Recall from Section 6.1 that VPR repeatedly places and routes the circuit using a binary search, stopping when it finds the minimum channel width for which it can achieve a legal route. Figure 20 reports the minimum channel width in the *x* and *y* directions found by VPR for each benchmark. *GPC* tends to achieve routability with narrower channels than *Ternary*, *6:2+GPC*, and *7:2+GPC*. As *GPC* requires more LABs than the other synthesis methods, the overall circuit is spread across a greater portion of the FPGA area. This tends to reduce congestion in the routing network, and hence, competition for routing tracks in the most congested area [DeHon 1999].

Jamieson and Rose [2006] have suggested 180 routing tracks per channel for a typical modern FPGA. Figure 20 shows that all of our benchmarks require no more than *60* routing tracks per channel, indicating that routability of the

benchmarks studied here would not be a concern, with or without the modified logic cell presented in this article.

8. ENERGY CONSUMPTION

8.1 VPR Power Model

A power model for VPR (*PVPR*) was developed by Poon et al. [2005] for island-style FPGAs; Choy and Wilton [2006] then extended the model to handle embedded IP cores, such as DSP blocks. PVPR models the power dissipated in the logic cells and the routing network separately. VPR's BLEs do not contain carry chains; consequently, we replaced the PVPR's default logic power model with a new one for the ALM-like logic cell proposed in this paper, including carry chains.

PVPR computes a cycle-by-cycle estimate of energy consumption for a VPR netlist; each cycle, an input vector is input to the system, which causes switching activity within the circuit. The cycle-by-cycle switching activity on each net is input to the power model.

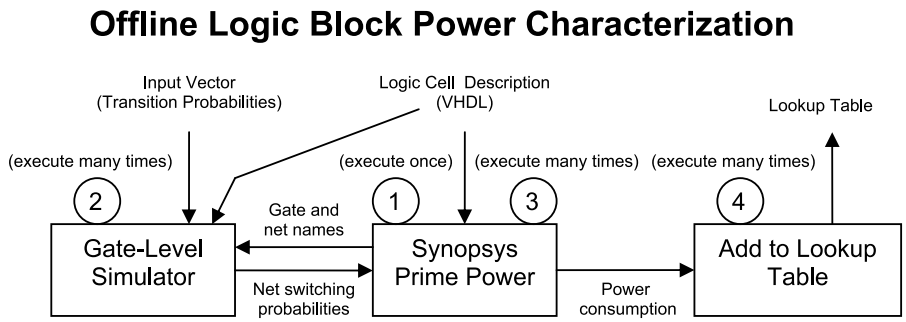
The routing power model, in addition to switching activity, requires architectural information, such as the switch box topology, number of wires per channel, length of each wire, etc.; it also requires technology-specific information about the wires, such as per-unit resistance and capacitance. PVPR's routing power model was left unmodified.

The remainder of this section focuses on the modeling of the power consumption of our logic cell. PVPR uses this information on a cycle-by-cycle basis to compute the energy dissipated by the logic cell over a period of time.

Choy and Wilton [2006] studied three offline power characterization techniques, *Constant*, *Lookup*, and *PinCap*, and compared their estimates against *Synopsys Prime Power* as a baseline. The most accurate method was *Lookup*, which achieved an average error of 6%. *Lookup* approximates the average power dissipated by each logic cell as a function of the average switching activity of all of its inputs. We have adopted the *Lookup* technique for use in our experiments.

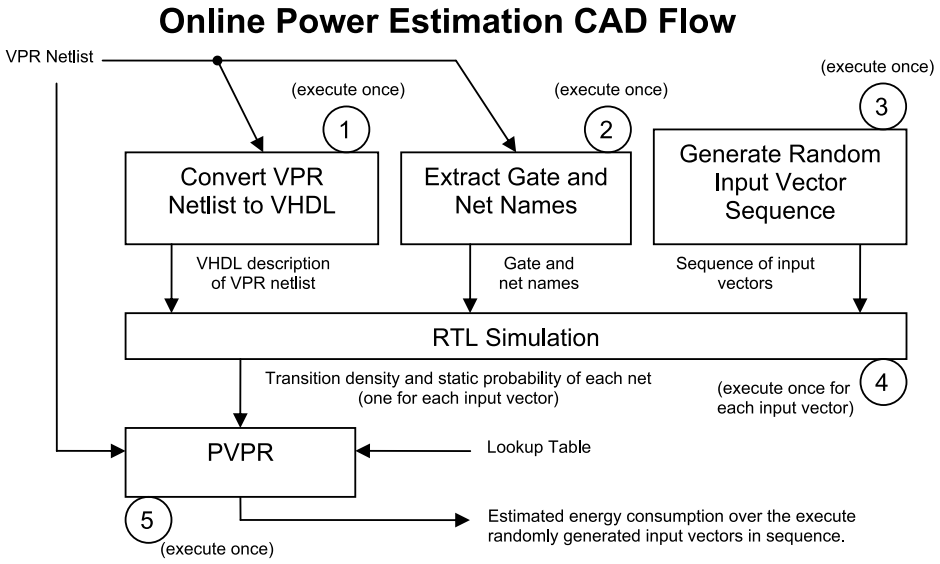
The *transition density* of a signal is the average number of transitions per unit time. The *static probability* is the probability of a signal being high at any given time. Transition density and static probability can be computed to estimate the switching activity on each net in the circuit [Najm 1994]. Lamoureux and Wilton [2006] developed an FPGA-based activity estimator based on these techniques. The time complexity of activity estimation for a k-input Boolean function is $O(2^k)$; although suitable for one LUT (e.g., $k = 4$ or 6), it becomes prohibitive for more complex LUT-based cells with carry chains and for logic clusters containing several logic cells and local routing. Instead, we use gate-level simulation for faster and more accurate activity estimation.

Figure 21(a) illustrates the offline power characterization phase. First a VHDL description of the logic cell is provided to *Synopsys Prime Power*, which assigns names to each gate and net. This information is provided to a gate-level simulator that estimates the transition activity of each net. The names



1. Synopsys Prime Power: Assign gate and net names
For as many input vectors as desired
   2. Gate-level simulation: Estimate net switching probabilities.
   3. Synopsys Prime Power: Compute power consumption
   4. Add result to Lookup Table
EndFor

(a)



1. Convert VPR netlist to VHDL
2. Extract gate and net names from VPR netlist
3. Randomly generate a sequence of input vectors
4. Apply RTL simulation to each input vector to compute the transition density and static probability of each net.
5. PVPR: Estimate energy consumption for the sequence of input vectors using the transition densities and static probabilities computed in Step 4.

(b)

Fig. 21. (a) Offline logic block power characterization and (b) online power estimation CAD flow.

are provided in the first step so that the gate-level simulator can properly annotate each net with its estimated switching probability. This information is then returned to *Synopsys Prime Power*, which estimates the total power dissipated by the circuit. In addition to the gate and net names, the gate-level simulator also requires an input vector containing the switching probability for each input bit. The gate-level simulation and power computation steps are repeated for a variety of input vectors with different switching probabilities.

The power consumption reported by *Synopsys Prime Power* is stored in a lookup table in the PVPR architecture description file. The table contains the estimated power dissipation for transition activities ranging from 0 to 1 in increments of 0.1; separate tables are instantiated depending on whether or not each output is written to its flip-flop.

Figure 21(b) shows the online power estimation flow. The input is a VPR netlist, that is, a circuit that has been synthesized onto a VPR-based FPGA architecture. The first two steps are (1) to convert the VPR netlist to VHDL and (2) to assign gate and net names. The names allow us to annotate the nets with transition densities and static probabilities for use by PVPR. The third step is to generate a random sequence of input vectors.

In the fourth step, an RTL simulator applies each input vector as a stimulus to the VHDL description of the circuit generated in the first step. RTL simulation determines the value of the signal on each wire and computes the transition density and static probability of each net. The static and density probabilities are provided to PVPR using the namespace generated in the second step. Lastly, PVPR estimates the energy consumed when the sequence of input vectors is applied to the placed-and-routed circuit using the transition densities and static probabilities computed by the fourth step. The lookup table computed during offline power characterization estimates the power consumption of the logic cells.

8.2 Results

Figure 22 shows the energy consumed for each benchmark by each synthesis methodology, decomposed into energy consumption in the logic and routing network of the FPGA. *GPC* consumed the most energy for all benchmarks, except for *H.264 ME* and *RYGYBY*.

This can be attributed to two factors: (1) *GPC* tends to have longer wires than *Ternary*, *6:2+GPC*, or *7:2+GPC*, which increases routing energy consumption; and (2) each GPC requires two ALMs, whereas each *6:2* or *7:2* compressor or two-bit ternary adder, requires one ALM; thus, there are more transistors leaking power, and more transistors where switching activity may occur. On average, the total energy consumption of *Ternary*, *6:2+GPC*, and *7:2+GPC* are approximately equal, but vary from benchmark to benchmark. On average, the percentage of total energy consumption due to the routing network ranges from *74%* (*Ternary*) to *81%* (*7:2+GPC*).

Among *Ternary*, *6:2+GPC*, and *7:2+GPC*, the average routing energy consumption is approximately equal, while the compressors do gain small advantages in terms of logic energy consumption. *GPC*, in contrast, consumes more

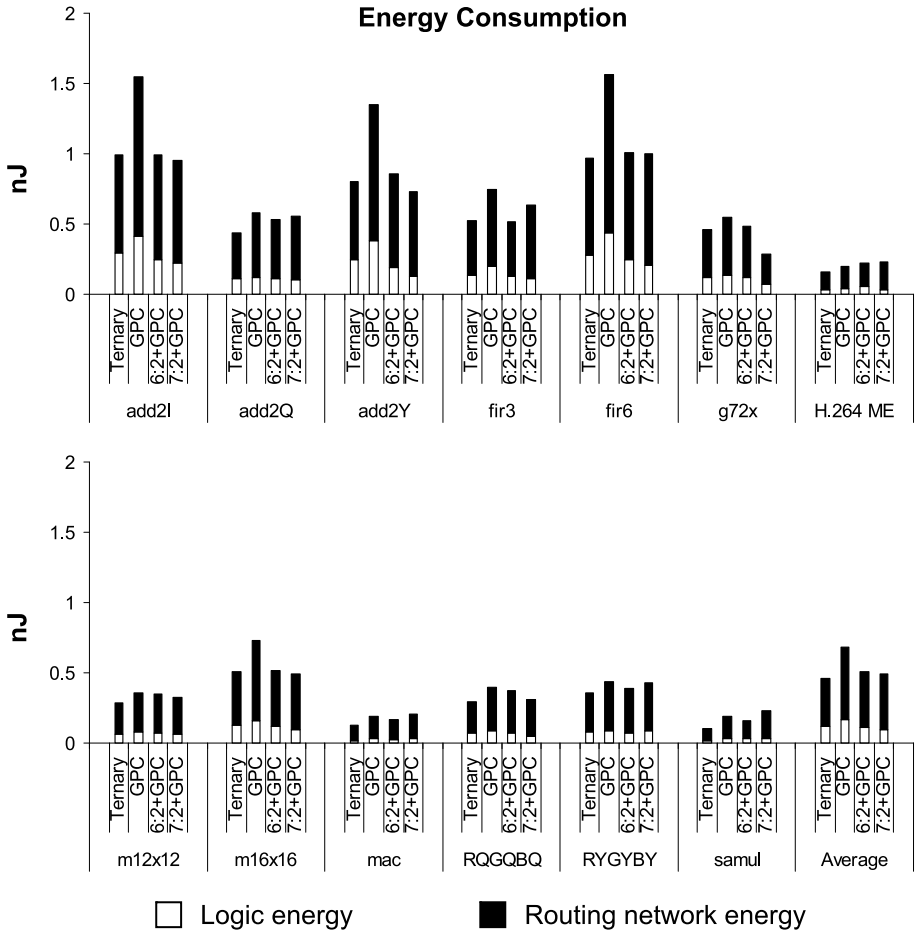


Fig. 22. Total energy consumption observed for each benchmark decomposed into logic and routing.

energy in both the logic and routing network, compared to the other three synthesis approaches.

9. CONCLUSION

This article has introduced a new FPGA logic cell and carry chain that can be configured as a *6:2* or *7:2* compressor. These logic cells can accelerate a wide variety of arithmetic applications, including those which can be transformed [Verma et al. 2008] to expose large compressor trees. In contrast, the *Altera ALM* can be configured as a ternary ripple carry adder, or two ALMs can be configured as a six-input GPC using the carry chains. Compressor tree synthesis using GPC mapping [Parandeh-Afshar et al. 2008b, 2008c] significantly reduces the critical path delay compared to synthesis using ternary adder trees; however, GPC mapping requires more ALMs and consumes considerably more energy. The logic cell proposed in this article significantly improves the situation. Used in conjunction with GPC mapping, the new logic cell offers a

moderate average reduction in critical path delay and total wirelength compared to GPC mapping using standard ALMs, while using slightly more logic resources and consuming approximately the same amount of energy as synthesis on ternary adder trees.

REFERENCES

BETZ, V. AND ROSE, J. 1997. VPR: A new packing, placement, and routing tool for FPGA research. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. 213–222.

BETZ, V., ROSE, J., AND MARQUARDT, A. 1999. *Architecture and CAD for Deep Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA.

BRISK, P., VERMA, A. K., IENNE, P., AND PARANDEH-AFSHAR, H. 2007. Enhancing FPGA performance for arithmetic circuits. In *Proceedings of the 44th Design Automation Conference*. 404–409.

CEVRERO, A., ATHANASOPOULOS, P., PARANDEH-AFSHAR, H., VERMA, A. K., BRISK, P., GURKAYNAK, F. K., LEBLEBICI, Y., AND IENNE, P. 2008. Architectural improvements for field programmable counter arrays: Enabling efficient synthesis of fast compressor trees on FPGAs. In *Proceedings of the 16th International Symposium on Field Programmable Gate Arrays*. 181–190.

CHEN, C.-Y., CHIEN, S.-Y., HUANG, Y.-W., CHEN, T.-C., WANG, T.-C., AND CHEN, L.-G. 2006. Analysis and architecture design of variable block-size motion estimation for H.264/AVC. *IEEE Trans. Circ. Syst. 53*, 578–593.

CHEREPACHA, D. AND LEWIS, D. 1996. DP-FPGA: an FPGA architecture optimizated for datapaths. *VLSI Des. 4*, 329–343.

CHOY, N. C. K. AND WILTON, S. J. E. 2006. Activity-based power estimation and characterization of DSP and multiplier blocks in FPGAs. In *Proceedings of the IEEE International Conference on Field Programmable Technology*. 253–256.

CONG, J. AND HUANG, H. 2005. Technology mapping and architecture evaluation for k/m-macrocell-based FPGAs. *ACM Trans. Des. Automat. Electron. Syst. 10*, 3–23.

DADDA, L. 1965. Some schemes for parallel multipliers. *Alta Frequenza 34*, 349–356.

DEHON, A. 1999. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). In *Proceedings of the International Symposium on Field Programmable Gate Arrays*. 69–76.

FADAVI-ARDEKANI, J. 1993. M × N Booth encoded multiplier generator using optimized Wallace trees. *IEEE Trans. VLSI Syst. 1*, 120–125.

FREDERICK, M. T. AND SOMANI, A. K. 2006. Multi-bit carry chains for high performance reconfigurable fabrics. In *Proceedings of the 16th International Conference on Field Programmable Logic and Applications*. 1–6.

GROVER, R. S., SHANG, W., AND LI, Q. 2002. A faster distributed arithmetic architecture for FPGAs. In *Proceedings of the 10th International Symposium on FPGAs*. 31–39.

HAUCK, S., HOSLER, M. M., AND FRY, T. W. 2000. High-performance carry chains for FPGAs. *IEEE Trans. VLSI Syst. 8*, 138–147.

HU, Y., DAS, S., TRIMBERGER, S., AND HE, L. 2007. Design, synthesis, and evaluation of heterogeneous FPGA with mixed LUTs and macro-gates. In *Proceedings of the International Conference on Computer-Aided Design*. 188–193.

JAMIESON, P. AND ROSE, J. 2006. Enhancing the area of FPGAs with hard circuits using shadow clusters. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*. 1–8.

KASTNER, R., KAPLAN, A., OGRENCI-MEMIK, S., AND BOZORGZADEH, E. 2002. Instruction generation for hybrid reconfigurable systems. *ACM Trans. Des. Automat. Electro. Syst. 7*, 605–627.

KAVIANI, A., VRANISEC, D., AND BROWN, S. 1998. Computational field programmable architecture. In *Proceedings of the IEEE Custom Integrated Circuits Conference*. 261–264.

KUON, I. AND ROSE, J. 2007. Measuring the gap between FPGAs and ASICs. *IEEE Trans. Comput.-Aid. Des. 26*, 203–215.

KWON, O., NOWKA, K., AND SWARTZLANDER JR., E. E. 2002. A 16-bit by 16-bit MAC design using fast 5:3 compressor cells. *J. VLSI Sign. Process. 31*, 77–89.

LAMOUREUX, J. AND WILTON, S. J. E. 2006. Activity estimation for field programmable gate arrays. In *Proceedings of the 16th International Conference on Field Programmable Logic and Applications*. 1–8.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th International Symposium on Microarchitecture*. 330–335.

LEIJTEN-NOWAK, K. AND VAN MEERBERGEN, J. L. 2003. An FPGA architecture with enhanced datapath functionality. In *Proceedings of the 11th International Symposium on FPGAs*. 195–204.

MIRZAEI, S., HOSANGADI, A., AND KASTNER, R. 2006. FPGA implementation of high speed FIR filters using add and shift method. In *Proceedings of the International Conference on Computer Design*. 308–313.

MORA MORA, H., PASCUAL MORA, J., SANCHEZ ROMERO, J. L., AND PUJOL LOPEZ, F. 2006. Partial product reduction based on look-up tables. In *Proceedings of the International Conference on VLSI Design*. 399–404.

NAJM, F. N. 1994. A survey of power estimation techniques in VLSI circuits. *IEEE Trans. VLSI Syst. 2*, 446–455.

OKLOBDZIJA, V. G. AND VILLEGER, D. 1995. Improving multiplier design by using improved column compression tree and optimized final adder in CMOS technology. *IEEE Trans. VLSI Syst. 3*, 292–301.

PARANDEH-AFSHAR, H. BRISK, P., AND IENNE, P. 2008a. A novel FPGA logic block for improved arithmetic performance. In *Proceedings of the 16th International Symposium on Field Programmable Gate Arrays*. 171–180.

PARANDEH-AFSHAR, H., BRISK, P., AND IENNE, P. 2008b. Efficient synthesis of compressor trees on FPGAs. In *Proceedings of the Asia-South Pacific Design Automation Conference*. 138–143.

PARANDEH-AFHSAR, H., BRISK, P., AND IENNE, P. 2008c. Improving synthesis of compressor trees on FPGAs via integer linear programming. In *Proceedings of the International Conference on Design Automation and Test in Europe*. 1256–1262.

PARANDEH-AFSHAR, H. BRISK, P., AND IENNE, P. 2009. Exploiting fast carry chains of FPGAs for designing compressor trees. In *Proceedings of the 19th International Conference on Field Programmable Logic and Applications*. 242–249.

PARHAMI, B. 2000. *Computer Arithmetic, Algorithms and Hardware Designs*. Oxford University Press.

POLDRE, J. AND TAMMEMAE, K. 1999. Reconfigurable multiplier for Virtex FPGA family. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*. 359–364.

POON, K. K. W., WILTON, S. J. E., AND YAN, A. 2005. A detailed power model for field-programmable gate arrays. *ACM Trans. Des. Automat. Electro. Syst. 10*, 279–302.

SANTORO, M. AND HOROWITZ, M. 1988. A pipelined 64x64b iterative array multiplier. In *Proceedings of the IEEE Solid State Circuits Conference*. 36–37, 290.

SONG, P. J. AND DE MICHELI, G. 1991. Circuit and architecture tradeoffs for high-speed multiplication. *IEEE J. Solid-State Circ. 26*, 1184–1198.

STELLING, P. F., MARTEL, C. U., OKLOBDZIJA, V. J., AND RAVI, R. 1998. Optimal circuits for parallel multipliers. *IEEE Trans. Comput. 47*, 273–285.

STELLING, P. F. AND OKLOBDZIJA, V. J. 1996. Design strategies for optimal hybrid final adders in a parallel multiplier. *J. VLSI Signal Process. 14*, 321–331.

STENZEL, W. J., KUBITZ, W. J., AND GARCIA, G. H. 1977. A compact high-speed parallel multiplication scheme. *IEEE Trans. Comput. C-26*, 948–957.

SWARTZLANDER JR., E. E. 1973. Parallel counters. *IEEE Trans. Comput. C-22*, 1021–1024.

UM, J. AND KIM, T. 2002. Layout-aware synthesis of arithmetic circuits. In *Proceedings of the 39th Design Automation Conference*. 207–212.

VERMA, A. K., BRISK, P., AND IENNE, P. 2008. Data-flow transformations to maximise the use of carry-save representation in arithmetic circuits. *IEEE Trans. Comput.-Aid. Des. 27*, 1761–1774.

VERMA, A. K. AND IENNE, P. 2007a. Automatic synthesis of compressor trees: Reevaluating large counters. In *Proceedings of the International Conference on Design Automation and Test in Europe*. 443–448.

VERMA, A. K. AND IENNE, P. 2007b. Improving XOR-dominated circuits by exploiting dependencies between operands. In *Proceedings of the Asia-South Pacific Design Automation Conference*. 601–608.

WALLACE, C. S. 1964. A suggestion for a fast multiplier. *IEEE Trans. Elec. Comput. 13*, 14–17.

WEINBERGER, A. 1981. A 4:2 carry save adder module. *IBM Techn. Disclos. Bull. 23*.

ZUCHOWSKI, P. S., REYNOLDS, C. B., GRUPP, R. J., DAVIS, S. G., CREMEN, B., AND TROXEL, B. 2002. A hybrid ASIC and FPGA architecture. In *Proceedings of the International Conference on Computer-Aided Design*. 187–194.

Received August 2008; revised February 2009; accepted June 2009