High-Speed Hardware Partition Generation

JON T. BUTLER, Naval Postgraduate School
TSUTOMU SASAO, Meiji University

We demonstrate circuits that generate set and integer partitions on a set S of n objects at a rate of one per clock. Partitions are ways to group elements of a set together and have been extensively studied by researchers in algorithm design and theory. We offer two versions of a hardware set partition generator. In the first, partitions are produced in lexicographical order in response to successive clock pulses. In the second, an index input determines the set partition produced. Such circuits are useful in the hardware implementation of the optimum distribution of tasks to processors. We show circuits for integer partitions as well. Our circuits are combinational. For large n, they can have a large delay. However, one can easily pipeline them to produce one partition per clock period. We show (1) analytical and (2) experimental time/complexity results that quantify the efficiency of our designs. For example, our results show that a hardware set partition generator running on a 100MHz FPGA produces partitions at a rate that is approximately 10 times the rate of a software implementation on a processor running at 2.26GHz.

Categories and Subject Descriptors: B.2.4 [**Arithmetic and Logic Structure**]: High-Speed Arithmetic; B.6.1 [**Design Styles**]: Parallel Circuits; B.7.1 [**Integrated Circuits**]: Algorithms Implemented in Hardware

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Reconfigurable computer, set partition, integer partition, index to partition generator, combinatorial objects, partition tree, partition diagram

ACM Reference Format:
Jon T. Butler and Tsutomu Sasao. 2014. High-speed hardware partition generation. ACM Trans. Reconfig. Technol. Syst. 7, 4, Article 28 (December 2014), 17 pages.
DOI: http://dx.doi.org/10.1145/2629472

1. INTRODUCTION

The enumeration of integer and set partitions by *software* has long been a fertile area of research [Reingold et al. 1977; Semba 1984]. However, in spite of its promise, enumeration by *hardware* has received comparatively little attention. A naive approach to generating *general* partitions is slow. For example, a circuit implementation of a generator of four-element set partitions would require an 8-bit output, where the block in which each element is located is represented as 00, 01, 10, and 11. One can then generate all 8-bit binary tuples and test whether each is a set partition. This sieve

This research is supported by a Grant-in-Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS). This article is an extended version of Butler and Sasao [2013b].
Authors' addresses: J. T. Butler, Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, California 93943-5121, USA; email: jon_butler@msn.com; T. Sasao, Department of Computer Science, Meiji University; 1-1-1 Higashi-Mita, Tama-ku, Kawasaki, Kanagawa, Japan, 214-8571; email: sasao@cs.meiji.ac.jp.

DOI: http://dx.doi.org/10.1145/2629472

approach is slow because there are many fewer partitions than there are binary n-tuples. We seek a circuit in which a partition emerges at every clock pulse.

Partitions are important combinatorial objects. Set partitions on n elements enumerate the equivalence relations on n elements. For example, the set partition $\{\{3, 1\}, \{2\}, \{0\}\}$ places elements 3 and 1 in the same block and elements 2 and 0 in separate blocks. Each block represents all elements related by the equivalence relation. The number of partitions can be large, and the use of hardware offers a high-speed alternative to software.

Our effort to enumerate integer and set partitions is part of a larger project to answer a fundamental question: *Which of the combinatorial objects can be enumerated by a simple circuit at a rate of one per clock?* Combinatorial objects include combinations, permutations, set partitions, integer partitions, codewords, bent Boolean functions, and monotone Boolean functions. The term "one per clock" suggests a rate that is independent of clock frequency and refers to a circuit designed to generate individual partitions in a single clock period. We have shown a circuit that can sieve for bent Boolean functions [Shafer et al. 2010], and we have shown that, given a polynomial-sized circuit that can sieve, there exists a (possibly exponential-sized) circuit that can *generate* the same set of objects at a rate of one per clock [Butler and Sasao 2013a]. However, this leaves open the question of whether a simple circuit exists that can generate bent Boolean functions at a rate of one per clock. It also invites the question of whether other combinatorial objects can be generated at a rate of one per clock. We have shown that combinations [Butler and Sasao 2011] and permutations [Butler and Sasao 2012] can be generated at a rate of one per clock. In this article, we show that set and integer partitions can be generated at a rate of one per clock.

The ability to generate set partitions has important practical applications. Hankin and West [2007] show how partitions are used to solve optimization problems in bioinformatics, forensic science, and scheduling. For example, set partitions can be used to specify the ways tasks are allocated to processors, from which one seeks the partition that corresponds to the shortest computation time. This last application especially requires high-speed enumeration of partitions. In multistate distribution *systems* (packet, water, gas, etc.) [Nagayama et al. 2012], the overall quality of service is dependent on attributes of the components, as measured by variables. There is a need to quickly enumerate partitions of the variables used in decision diagrams that model the system. Since models of such systems are huge, as are the data structures, decision diagrams are used in the model. In 1997, Bousquet-Mélou and Erikson [1997a, 1997b] developed the theory of "Lecture Hall Partitions," which describes all possible ways an n-row lecture hall can be configured so that every seat has a view of the lecturer. Enumerating such partitions allows one to analyze all possible choices before committing to construction, including combinations of such partitions.

An integer partition is a way to write an integer n as a sum of parts (positive integers), where the order of parts is unimportant. For example, $2 + 1 + 1$ is an integer partition on 4. Recent research in computational molecular biology has shown the importance of integer partitions in quantifying the role of genes in determining global characteristics of species. For example, Woodruff [2006] and Chen et al. [2008] have identified the importance of solving the minimum common integer partition (MCIP) problem in DNA fingerprint assembly. This problem requires the enumeration of partitions at high speed, since so many partitions must be considered. We know of no hardware specifically designed for the MCIP problem. However, the enormous amount of logic available on FPGAs for highly parallel applications has inspired study into their use as accelerators in computational biology applications [Terasic 2011].

The availability of large programmable logic circuits has allowed computations to be performed in hardware that previously could only be done in software, but at a

much higher rate. There are many papers on programs and algorithms for enumerating partitions [Gosper 1972; Kawano and Nakano 2005; McKay 1965; Reingold et al. 1977; Oommen and Ng 1990; Semba 1984; Zoghbi and Stojmenovič 1998], including parallel algorithms [Stojmenovič 1990]. While there are many papers on programs and algorithms for enumerating partitions, we know of only one that uses an FPGA to enumerate partitions. Lavenier and Saouter [1998] reported the use of FPGAs to enumerate two-part integer partitions to calculate the number of Goldbach partitions on even integers. This was intended to provide insight into the Goldbach conjecture: every even integer n greater than 3 is the sum of two prime integers.

This article presents hardware that enumerates general set and integer partitions. In Section 2, we discuss the generation of *set* partitions, showing circuits and experimental results. In Section 3, we show the design of an *integer* partition circuit and experimental results. Finally, in Section 4, we give concluding remarks.

2. SET PARTITIONS

2.1. Introduction

Definition 2.1. Given an n-set $S = \{0, 1, \ldots, n-1\}$, $\{S_0, S_1, \ldots, S_{n-1}\}$ is a **set partition** of S iff (1) $S_i \subseteq S$, (2) $S_i \bigcap S_j = \emptyset$ for $i \neq j$, and (3) $\bigcup_{i=0}^{n-1} S_i = S$.

A set partition of a set S is the placement of elements of S into blocks. For example, there are 15 set partitions of four elements 0, 1, 2, and 3. These are $\{\{3, 2, 1, 0\}\}$ (all elements in the same block), $\{\{3, 2, 1\}, \{0\}\}$, $\{\{3, 2, 0\}, \{1\}\}$, $\{\{3, 2\}, \{1, 0\}\}$, $\{\{3, 2\}, \{1\}, \{0\}\}$, $\{\{3, 1, 0\}, \{2\}\}$, $\{\{3, 1\}, \{2, 0\}\}$, $\{\{3, 1\}, \{2\}, \{0\}\}$, $\{\{3, 0\}, \{2, 1\}\}$, $\{\{3\}, \{2, 1, 0\}\}$, $\{\{3\}, \{2, 1\}, \{0\}\}$, $\{\{3, 0\}, \{2\}, \{1\}\}$, $\{\{3\}, \{2, 0\}, \{1\}\}$, $\{\{3\}, \{2\}, \{1, 0\}\}$, and $\{\{3\}, \{2\}, \{1\}, \{0\}\}$ (all elements in separate blocks). Neither the order of the blocks nor the order of elements within each block matters. For example, partitions $\{\{3, 1\}, \{2\}, \{0\}\}$ and $\{\{0\}, \{1, 3\}, \{2\}\}$ are identical. The number of set partitions increases rapidly as the number of elements n increases and are counted by the *Bell* numbers $B(n)$. For example, for sets of size $n = 2,\ 3,\ 4,\ 5,\ 6,\ 7,$ and 8, the number of set partitions is $B(n) = 2,\ 5,\ 15,\ 52,\ 203,\ 877,$ and 4, 140. For large n, $B(n)$ is bounded above by $(\frac{0.792n}{\ln(n+1)})^n$ [Berend and Tassa 2010].

It is convenient to represent a partition in its restricted growth string form, as follows. Since a set partition is unchanged by a reordering of blocks, call the block in which $n-1$ is located block 0. Then, $n-2$ is either in the same block, block 0, or in a different block. If it is in a different block, call that block 1. Then, $n-3$ is either in block 0 or 1 or some other block. If it is in some other block, call that block 2. Continue in this way until all elements are assigned a block. For example, the partition $\{\{3, 1\}, \{2\}, \{0\}\}$ has the restricted growth string (0102). That is, 3 is (always) in block 0. This explains the leftmost 0 of the restricted growth string. Next, 2 is in a *different* block than 3. This explains the next 1. Next, 1 is in the *same* block as 3, and this explains the next 0. Finally, 0 is in its own block, and this explains the rightmost 2. From this, it can be seen that integer elements in the restricted growth string are indices to blocks in the partition. Formally:

Definition 2.2. An n-element **restricted growth string** is a sequence $(b_0 b_1 \ldots b_{n-1})$ such that $b_0 \leq b_i \leq \max_{0 \leq j < i}(b_j + 1)$, where $b_0 = 0$.

The first element of a restricted growth string is always 0, signifying that element $n-1$ is always in block 0. As one progresses down through the elements of a partition, one encounters either a new block or an old one, represented by the next larger block index (1 plus the maximum of all elements with smaller index) or a previously encountered block index, respectively.

Table I. Partitions on a Set of $n = 4$ Versus Their Index i

i	Partition	Restricted Growth String
0	{{3, 2, 1, 0}}	(0 0 0 0)
1	{{3, 2, 1}, {0}}	(0 0 0 1)
2	{{3, 2, 0}, {1}}	(0 0 1 0)
3	{{3, 2}, {1, 0}}	(0 0 1 1)
4	{{3, 2}, {1}, {0}}	(0 0 1 2)
5	{{3, 1, 0}, {2}}	(0 1 0 0)
6	{{3, 1}, {2, 0}}	(0 1 0 1)
7	{{3, 1}, {2}, {0}}	(0 1 0 2)
8	{{3, 0}, {2, 1}}	(0 1 1 0)
9	{{3}, {2, 1, 0}}	(0 1 1 1)
10	{{3}, {2, 1}, {0}}	(0 1 1 2)
11	{{3, 0}, {2}, {1}}	(0 1 2 0)
12	{{3}, {2, 0}, {1}}	(0 1 2 1)
13	{{3}, {2}, {1, 0}}	(0 1 2 2)
14	{{3}, {2}, {1}, {0}}	(0 1 2 3)

LEMMA 2.1 [ORLOV 2002]. *There is a bijection between the set of partitions of an n-set and the set of n-element restricted growth strings.*

The one-to-one relation between partitions and restricted growth strings means that we can enumerate the latter with a guarantee that we enumerate the former. Converting from one to the other requires only a combinational logic circuit. Table I shows the set of all 15 partitions on $n = 4$ elements $\{3, 2, 1, 0\}$. The first column shows the index i, where $0 \leq i \leq 14$. i indexes the set partitions according to the increasing lexicographical order of the restricted growth strings. The second column shows how the actual partition distributes the elements $\{3, 2, 1, 0\}$ into blocks. Here, commas separate blocks and elements within the same block. The third column shows the restricted growth string. Each restricted growth string begins in 0, indicating that 3 is (always) in the first (0th) block. The second element shows where element 2 is located (in the 0th or 1st block). The third element shows where element 1 is located (in the 0th, 1st, or 2nd block). The fourth element shows where element 0 is located (in the 0th, 1st, 2nd, or 3rd block). In general, the jth element can be in the 0th, 1st, $\ldots$ $(j-1)$-th block.

In order to deduce the circuit needed to produce a set partition from an index, we introduce the partition tree. Specifically, the methodology to design a hardware index to set partition converter uses a tree structure to store all partitions on a set $\{n-1, n-2, \ldots, 1, 0\}$ of n elements.

Definition 2.3. A (set) **partition tree** for n consists of three node types:

(1) the single **root** node labeled 0,
(2) **internal** nodes labeled i, for all $i \in \{0, 1, \ldots, n-2\}$,
(3) **terminal** nodes labeled i, for all $i \in \{0, 1, \ldots, n-1\}$,

and one edge type:

(1) an **edge** connects a node labeled i to a node labeled j iff (1) $0 \leq j \leq i+1$ and (2) the path from the root node to j has no more than n nodes.

A terminal node is simply the last node along a path from the root node. From the definition of an edge, all paths from the root node to a terminal node have n nodes (and $n-1$ edges). In a partition tree, the restricted growth string of a set partition is

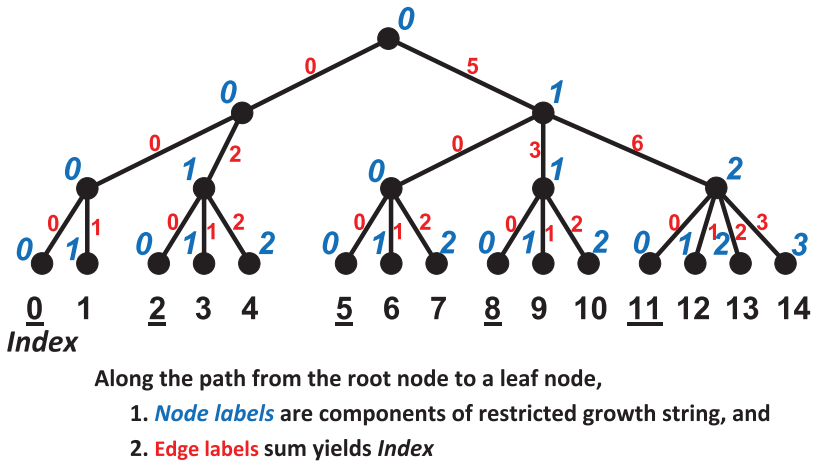

Fig. 1. Example of a partition tree for set partitions on $n = 4$ elements.

represented by the labels of edges along a path from the root node to a terminal node. Each node in a path specifies a block in which the corresponding element is located.

Example 2.1. Figure 1 shows the partition tree for partitions with $n = 4$ elements. Note that all paths from the root node to a terminal node have four nodes and three edges. Following the leftmost path from the root node to a terminal node yields the node labels (0000). This restricted growth string specifies that all elements, 3, 2, 1, and 0, belong to block 0. That is, this is the partition in which all elements are in a single block. Following the rightmost path yields the node labels (0123). This restricted growth string specifies the partition in which all elements are in different blocks. Following the path with node labels (0102) yields a partition with 3 and 1 in the same block and 2 and 0 each in separate blocks with just one element.

The partition tree is similar to a decision tree [Bryant 1986]. Each node has child nodes corresponding to all possible choices at that point. Like the decision tree, the partition is recursive. Although the partition tree and the decision tree are similar, their use in the design process is quite different.

Each terminal node corresponds to a partition. Figure 1 shows, as (additional) terminal labels, the index of the partition. There are 15 partitions in this example, labeled 0, 1, . . . , and 14. Note that edges are labeled by the part that each contributes to the index. For example, the edge from the root node to the node labeled 1 has weight 5. This is because the indices on the right side of the tree corresponding to the latter node all have index 5 or greater. It follows that the index associated with each node can be obtained by summing the weights in edges associated with the path from the root node to the corresponding terminal node. This is an important observation, since it provides the basis for a circuit design of a set partition generator.

2.2. Circuit Implementations

2.2.1. Introduction. In this section, we consider three circuit implementations of the set partition generator: (1) a sequential circuit implementation, which produces set partitions in lexicographical order; (2) a single-stage circuit implementation, which accepts an index and produces the set partition corresponding to that index; and (3) a multistage circuit implementation, which also accepts an index and produces the set partition corresponding to that index. We show that the sequential circuit is

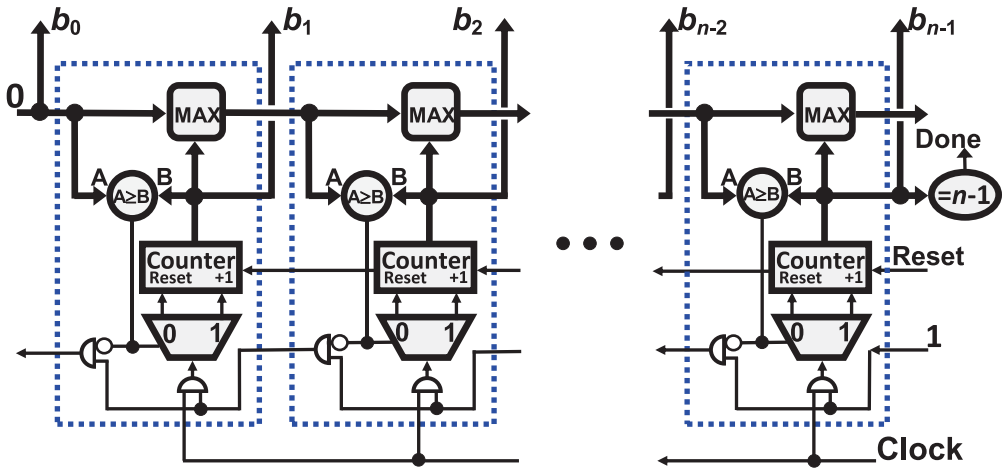

Fig. 2. Sequential set partition generator.

the simplest. However, it can only produce the set partitions in lexicographical order. The single stage and multistage circuits convert an index to the set partition with that index. Thus, they are useful in Monte Carlo simulations. Both produce a set partition at a rate of one per clock. However, the latency of the multistage circuit is larger, $n-2$, versus 1 for the single-stage circuit. But there is a significant penalty in the complexity of the single stage circuit, whose complexity we show as $O((\frac{n}{\ln(n)})^n)$ versus $O(n^2)$ for the multistage circuit.

2.2.2. Sequential Circuit Implementation. Figure 2 shows a sequential circuit implementation of a set partition converter. When Reset is asserted, the Counters of all stages are set to 0. The Clock comes in at the right. This passes to the MUX of the rightmost stage through an AND gate because the other AND gate input is 1. Because of values on the comparator inputs in this stage, the clock pulse is passed to the count input of this stage's Counter, causing it to increase by 1 (i.e., the right element of the restricted growth string is increased by 1). The comparator of this rightmost stage produces a 1 if and only if the max of all elements of the restricted growth string to the left of the current element are the same or greater than this stage's element. Otherwise, a 0 is produced, in which case the current element is strictly greater; the current element has reached its max value. A 0 result at the comparator output "switches" the MUX so that the clock is passed to the Reset input of the Counter. As a result, the next clock causes the current element to return to 0. Stages to the left of the rightmost stage operate in a similar manner. This circuit is similar to an ordinary up counter, except for the different ranges across the various digits. Overall, this circuit produces the next set partition in increasing lexicographical order according to the restricted growth string. Specifically, it first generates $(b_0 \dots b_{n-2} b_{n-1}) = (0 \dots 000)$, then $(0 \dots 001)$, and so forth. The count finishes when b_{n-1} is $n-1$. At this point, Done is asserted. This could be used externally, or it could stop the clock, preventing the circuit from receiving further clock pulses. With respect to delay, this circuit is similar to a ripple carry adder. Here, the critical path goes from right to left through the series of AND gates that have one inverted input.

2.2.3. Single-Stage Combinational Circuit Implementation. Figure 3 shows the single-stage index to set partition circuit for partitions of size $n = 4$. The index comes in on the left and is tested by five comparators. These test the range of the index and determine the

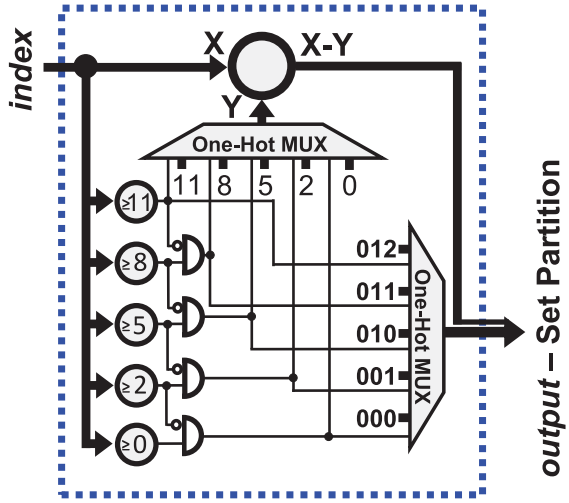

Fig. 3. Single-stage index to set partition circuit for $n = 4$.

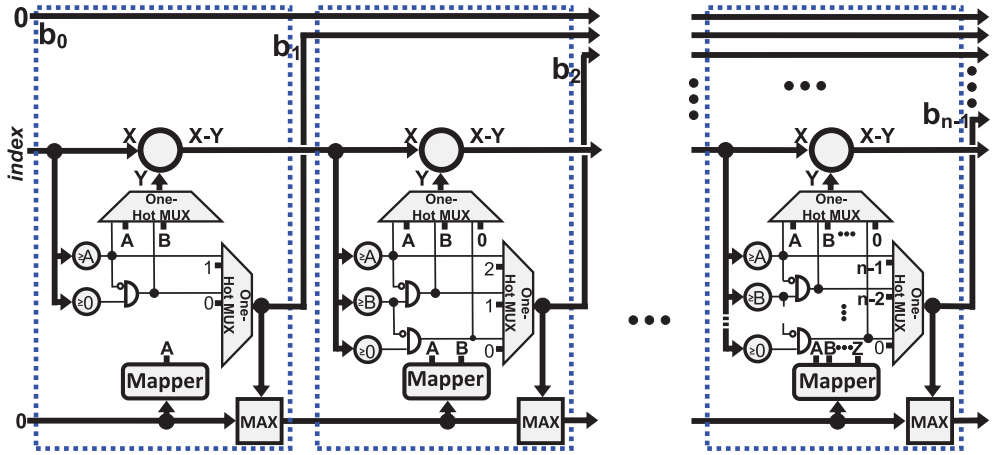

Fig. 4. Multistage index to set partition circuit.

first three elements of the restricted growth string. There are five possibilities, 000, 001, 010, 011, and 012. One of these five is applied to the one-hot MUX that drives the output. Also, the threshold is subtracted from the incoming index and the result applied to the output as the least significant digit. The threshold values in Figure 3 are determined by the partition tree shown in Figure 1. They correspond to the indices assigned to the 0 terminal nodes in Figure 1. The corresponding indices are shown in Figure 1 as underlined values assigned to terminal nodes. There is only one stage in this implementation.

2.2.4. Multistage Combinational Circuit Implementation. Figure 4 shows the multistage index to set partition converter. Here, the index comes in on the left. It then passes left to right through the stages. Within each stage, it is reduced (possibly by 0) and then passed onto the next stage, which is the first stage of a smaller partition tree. The reader may recall our earlier comment regarding the recursive nature of the set partition tree. At each stage, an element in the restricted growth string of the set partition is computed.

For example, in the left stage, b_1 is determined. From the partition tree in Figure 1, it can be seen that, if the index is 4 or less, b_1 is 0. Conversely, if the index is 5 or greater, b_1 is 1. It follows that the threshold A in Figure 4 is 5. Also, if the index is 5 or more, 5 is subtracted from the index and is passed to the next stage. Recall that the thresholds against which the index is compared vary according to the maximum value in the restricted growth string computed so far. The maximum value is computed in the MAX gate in the lower right-hand corner of each stage. In the leftmost stage, the output value of **MAX** is 0 or 1. This is passed to the next stage, which uses it to determine the two threshold values A and B. The critical path in the multistage index to set partition circuit extends from the index input on the left through the comparators, AND gates, MUXes, and subtractors of the various stages to the b_{n-1} output, which is the rightmost digit of the restricted growth string. Note that the subtractor and MAX gate of the rightmost stage are not used but are retained for consistency.

Note that, in a multistage index to set partition converter for $n = 4$, there are nine comparators. The single-stage index to set partition converter has five. This raises the question of which circuit is more compact for general n. This is addressed in the next section.

2.2.5. Circuit Complexity and Delay. Note that all three circuits use comparators. Further, the complexity of the other parts of the circuit is proportional to the number of comparators. For example, the number of AND gates is nearly the same as the number of comparators, and the one-hot MUX circuits have about as many inputs as the number of comparators. Therefore, it will be convenient to measure the circuit's complexity by the number of comparators it contains. In making this assumption, we neglect the increase in circuit complexity and delay in comparators and multiplexors that occur when n increases.

LEMMA 2.2. *The number of comparators C_i used in a set partition generator is*

1) *sequential (Figure 2): $C_1 = O(n)$,*
2) *single stage (Figure 3): $C_2 = O((\frac{n}{\ln(n)})^n)$, and*
3) *multistage (Figure 4): $C_3 = O(n^2)$.*

PROOF. In the case of the sequential set partition generator, each stage has one comparator, and there are $n - 1$ stages. Thus, $C_1 = O(n)$.

In the case of the single-stage set partition generator, the number of comparators is just the number of set partitions on $n - 1$, which is $C_2 = B(n - 1)$, where $B(n - 1)$ is the $n - 1$-th Bell number. From Berend and Tassa [2010], we have $B(n-1) < (\frac{0.792(n-1)}{\ln(n)})^{n-1}$. Thus,

$$C_2 = O\left(\left(\frac{n}{\ln(n)}\right)^n\right). \tag{1}$$

In the case of the multistage set partition generator, the first (leftmost) block has two comparators. The next block has three, the next four, and so forth. There are a total of $n - 2$ blocks. Thus, $C_3 = \sum_{i=2}^{n-2} i = \frac{n(n+1)}{2} - 2n + 4$, and we can write

$$C_3 = O(n^2). \quad \square \tag{2}$$

It is clear from Lemma 2.2 that the multistage index to set partition converter has many fewer comparators than the single-stage converter in the case of set partitions on many elements. Thus, the case for $n = 4$ discussed at the end of the previous section is not representative for large n. Recall for that specific case, the multistage convertor had *more* comparators (nine) than the single-stage convertor (five). We can also compare the circuits on the basis of their delay. In our analysis, we consider the

Table II. Frequency/Resources Used to Realize the Sequential Set Partition Generator on the Altera Stratix IV EP4SE530F43C3NES FPGA

n	# Set Partitions	In # Bits	Out # Bits	Freq. (MHz)	Delay (ns)	# Comb Fnc	# LUTs With Various # of Inputs 7-	6-	5-	4-	3-	Estimated # of Packed ALMs
5	52	6	15	236.2	4.234	42	1	1	12	8	20	22(0%)
6	203	8	18	172.6	5.793	60	2	5	19	13	21	34(0%)
7	877	10	21	156.4	6.393	58	3	1	15	9	30	31(0%)
8	4,140	13	24	130.8	7.643	63	3	3	13	11	33	35(0%)
16	1.05×10^{10}	34	64	122.1	8.190	245	5	28	87	61	64	135(0%)
32	1.28×10^{26}	88	160	53.0	18.863	741	3	231	221	94	192	459(0%)
64	1.72×10^{65}	217	384	25.9	38.584	1,923	10	477	659	298	479	1,146(0%)
128	1.12×10^{158}	526	896	11.0	91.278	3,847	13	727	1,666	427	1,014	2,134(1%)

delay of a circuit to be the number of stages through which the signal must pass. As in the case of complexity, we neglect the increase in circuit delay in comparators and multiplexors that occurs when n increases.

LEMMA 2.3. *The delay D_i in a set partition generator is*

1) *sequential (Figure 2): $D_1 = n - 1$,*
2) *single stage (Figure 3): $D_2 = 1$, and*
3) *multistage (Figure 4): $D_3 = n - 2$.*

PROOF. In the case of the sequential set partition generator, there are $n - 1$ stages through which a signal must pass. In the case of the single-stage set partition generator, there is exactly one stage, and the delay is independent of n. In the case of the multistage set partition generator, the index must propagate through $n - 2$ stages. □

Note that, in these calculations, we considered the multistage index to set partition converter to be *combinational*. When n is large, this circuit has a large delay. In order to improve the throughput, we will create a pipelined circuit by inserting registers between stages. In the next section, we compare the experimental delay of a *pipelined* version of the multistage circuit with the combinational circuit of the single-stage circuit. As a result, the time comparisons will be different from the derived delay.

2.2.6. Experimental Data. In the previous analysis, we used the number of comparators as a measure of complexity and the number of stages as a measure of delay. This yields insight but is only an approximate measure. In this section, we use actual FPGA resources. Using the Synopsys design tool Synplify Pro, we simulated the three circuits discussed previously on the Altera Stratix IV EP4SE530F43C3NES FPGA. Table II shows the resource usage for the sequential version.

From Table II, for all values of $n \leq 16$, the achieved frequency exceeds 100MHz. Thus, the sequential partition generator produces one partition per clock period for all $n \leq 16$, where the clock period is 10ns. To compare this rate to a software implementation of a sequential partition generator, we adapted Orlov's [Orlov 2002] program and ran it on an Intel Core™2 Duo P8400 processor running at 2.26GHz. For 8- and 16-element partitions, we achieve a rate of partitions of one per 94ns and 156ns, respectively. Thus, our hardware version realizes a 9.4 and 15.6 times speedup compared to the software version.

The first column in Table II shows n, the second column shows the number of set partitions, the third column shows the number of input bits, and the fourth column shows the number of output bits. All remaining columns show circuit parameters provided by Synplify Pro. The fifth column shows the frequency specified by Synplify Pro. The corresponding delay is shown in the sixth column. The seventh column shows the

Table III. Frequency/Resources Used to Realize the Single-Stage Index to Set Partition Converter on the Altera Stratix IV EP4SE530F43C3NES FPGA

n	# Set Par-titions	In # Bits	Out # Bits	Freq. (MHz)	Delay ns.	# Comb Fnc	# LUTs With Various # of Inputs 7-	6-	5-	4-	3-	Estimated # of Pac-ked ALMs
4	15	4	8	406.3	2.461	5	0	0	0	5	0	3(0%)
5	52	6	15	406.3	2.461	22	0	3	12	4	3	12(0%)
6	203	8	18	250.8	3.988	161	3	10	83	34	31	87(0%)
7	877	10	21	113.2	8.836	882	5	54	538	183	102	469(0%)
8	4,140	13	24	100.5	9.954	4,100	32	1,416	1,223	652	777	2,628(1%)

Table IV. Frequency/Resources Used to Realize the Multistage Index to Set Partition Converter on the Altera Stratix IV EP4SE530F43C3NES FPGA

n	# Set Par-titions	In # Bits	Out # Bits	Freq. (MHz)	Delay (ns)	# Comb Fnc	# LUTs With Various # of Inputs 7-	6-	5-	4-	3-	Estimated # of Pac-ked ALMs
5	52	6	15	403.5	2.478	57	0	4	19	20	14	35(0%)
6	203	8	18	275.0	3.636	100	1	7	36	38	18	63(0%)
7	877	10	21	227.8	4.389	203	0	8	82	71	42	121(0%)
8	4,140	13	24	203.0	4.926	326	3	20	124	107	72	196(0%)
16	1.05×10^{10}	34	64	101.4	9.859	3,842	35	718	1,524	1,130	435	2,339(0%)
32	1.28×10^{26}	88	160	55.6	17.973	38,305	87	3,671	19,768	8,016	6,763	21,206(9%)

number of combinational functions used in the realization. This is an overall measure of the logic resources used; it is generated in the first step of the synthesis, prior to the technology mapping process. The eighth through 12fth columns show the number of the various lookup tables (LUTs) that were used. The 13th (rightmost) column shows the estimated number of packed ALMs used in the realization, along with the percentage this represents of the total available. Because of the large number of partitions, for moderate n (e.g., $n \geq 32$), it will be too time consuming to enumerate *all* partitions at typical FPGA clock frequencies (e.g., 100MHz). However, such designs are useful in understanding the complexity/delay of these circuits. For index to set partition generators, however, even large n is useful. For example, in Monte Carlo simulations, the index is a random number that can be any of a large set of input values.

Table III shows the FPGA resources and frequency achieved on the Altera Stratix IV EP4SE530F43C3NES FPGA by the single-stage combinational logic index to set partition converter shown in Figure 3. As discussed, this has short delay paths. This is indicated by the column labeled "Freq. (MHz)," which has a relatively shallow decline as n, the number of elements, increases. Also, as discussed, this circuit has high complexity. This can be seen in Table III by the near fivefold increase in the values in the column labeled "# Comb Fnc" (number of combinational logic circuits) and by the nearly fivefold increase in the values in the column labeled "# Estimated Number of Packed ALMs" as n increases by 1. For the single-stage circuit, it was possible to achieve an n of only 8, which is significantly smaller than the values of n achieved for the sequential and multistage circuits. In comparing the delay of the single-stage combinational logic index to the set partition converter as shown in Figure 3 with the multistage circuit as shown in Figure 4, it is important to recall that, unlike the multistage circuit, the single-stage circuit is not pipelined. Thus, the multistage circuit achieves a higher clock speed. However, its latency is larger.

Table IV shows the FPGA resources and frequency achieved on the Altera Stratix IV EP4SE530F43C3NES FPGA by the multistage index to set partition circuit shown in Figure 4. This uses fewer resources than the single-stage circuit in Figure 3, but its

Table V. Integer Partitions on $n = 8$ Versus Their Index i

i	Partition							
0	1	1	1	1	1	1	1	1
1	2	1	1	1	1	1	1	-
2	2	2	1	1	1	1	-	-
3	2	2	2	1	1	-	-	-
4	2	2	2	2	-	-	-	-
5	3	1	1	1	1	1	-	-
6	3	2	1	1	1	-	-	-
7	3	2	2	1	-	-	-	-
8	3	3	1	1	-	-	-	-
9	3	3	2	-	-	-	-	-
10	4	1	1	1	1	-	-	-
11	4	2	1	1	-	-	-	-
12	4	2	2	-	-	-	-	-
13	4	3	1	-	-	-	-	-
14	4	4	-	-	-	-	-	-
15	5	1	1	1	-	-	-	-
16	5	2	1	-	-	-	-	-
17	5	3	-	-	-	-	-	-
18	6	1	1	-	-	-	-	-
19	6	2	-	-	-	-	-	-
20	7	1	-	-	-	-	-	-
21	8	-	-	-	-	-	-	-

latency is greater. In the design of the multistage circuit, registers were placed between each stage. As a result, the delay figures shown are reduced, approximating the delay of one stage. The first index comes out of this circuit in $n - 1$ clock periods.

The data shown comes from Verilog code that was written to implement each of the three circuit types. Synplify Pro was used to design each circuit. Further, ModelSim was used to simulate each circuit. In the case of the multistage circuit, a MATLAB program was written to produce a header file that was called from the Verilog code to provide threshold values for the comparators.

3. INTEGER PARTITIONS

3.1. Introduction

In this section, we show how to generate integer partitions at a rate of one per clock. Because there are many fewer integer partitions than set partitions, an efficient way to generate integer partitions is to precompute them, assign an index, and use a lookup table. However, we are motivated to develop a general strategy for the hardware design of circuits that produce combinatorial objects at a rate of one per clock [Butler and Sasao 2013a]. Specifically, we show that there exists a tree-like structure that represents integer partitions that is analogous to the partition tree for set partitions. Interestingly, its hardware realization is a cascade of individual blocks, similar to the circuit realization of the set partition generator.

Definition 3.4. A nonincreasing sequence of positive integers $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k)$ is an **integer partition** of n if $\sum_{i=1}^{k} \lambda_i = n$.

An integer partition can be viewed as an unordered partition of a set of identical objects. *Unordered* refers to the fact that order does not distinguish two otherwise equivalent collections of subsets. For example, Table V shows the 22 integer partitions of 8. Since

order is unimportant, we choose a lexicographical ordering, where the largest part is leftmost and the smallest rightmost. In this way, we can index the partitions by index i, as shown in Table V.

Each partition can be expressed as eight 4-bit numbers, each capable of realizing a quantity between 8 and 0 (0 is a placeholder, representing - (dash) in Table V). Fewer bits are possible. For example, the rightmost part has a maximum value of 1, which requires only one bit to represent. In this case, the application must tolerate different numbers of bits for the different parts. In this article, we choose all parts to have the same number of bits. The circuit we seek for $n = 8$, therefore, has eight 4-bit parts, for a total of 32 bits. If the end use of the circuit is a partition for printing (e.g., to ASCII), then one is likely to want the same number of bits to represent each part. If storage is limited, then the smallest needed word widths can be used. In this case, only 15 bits are needed. This follows from the observation that the eight possible parts need, from left to right, 4, 3, 2, 2, 1, 1, 1, and 1 bits to represent the maximum value within the corresponding part. Thus, a total of 15 bits is needed. Further, it has a 5-bit input, and thus, it is capable of realizing an index from 0 to 21. The circuit produces a partition as a function of the index i, where $0 \leq i < 22$, as shown in Table V.

One way to generate all integer partitions is to generate all 32-bit binary numbers, at a rate of one per clock, discarding those that are not integer partitions on n. However, only 22 of the 2^{32} 32-bit numbers or 0.0000005% are integer partitions. Therefore, this produces integer partitions at a rate that is much slower than one partition per clock. We seek a circuit that produces one integer partition per clock.

3.2. The Partition Diagram

3.2.1. Introduction. In designing the index to set partition converter, we used the set partition *tree*. In a similar way, we design a hardware index to integer partition generator using a partition *diagram*. This adapts a data structure introduced in Lin [2006] to store all partitions of an integer n.

Definition 3.5. The (integer) **partition diagram** for n consists of three types of nodes:

(1) a **root** node labeled $(-, n)$,
(2) **internal** nodes labeled (y, Y), where y corresponds to a part in a partition of m and $Y = m - y$, the remainder when y is subtracted from m, and
(3) **terminal** nodes labeled $(y, 0)$, where y is the final (and smallest) part of a partition of n,

and one type of edge:

(1) a directed **edge** connects a node (x, m) to a node (y, p) iff $m = y + p$, $x \geq y$, and $m \geq y$. No edge extends from a terminal node.

In the partition diagram for n, an integer partition is represented by a directed path from the root node to a terminal node. Each node in a path generates a part of the partition. Each directed edge goes from a source node to a sink node, where the source node expresses the current part p in a partition on n, and the sink node is the root node of a partition diagram for partitions on $n - p$. In our version of the partition diagram, the largest parts occur first at the root node on the left and the smallest part occurs at the terminal nodes on the right. A node appears at some level if its part value and remaining sum value can exist at that level. This specification aids the understanding of the design methodology. Therefore, two identically labeled nodes may exist at two or more different levels. For example, in Figure 5, the node $(1, 3)$ exists at levels 2, 3, 4, and 5.

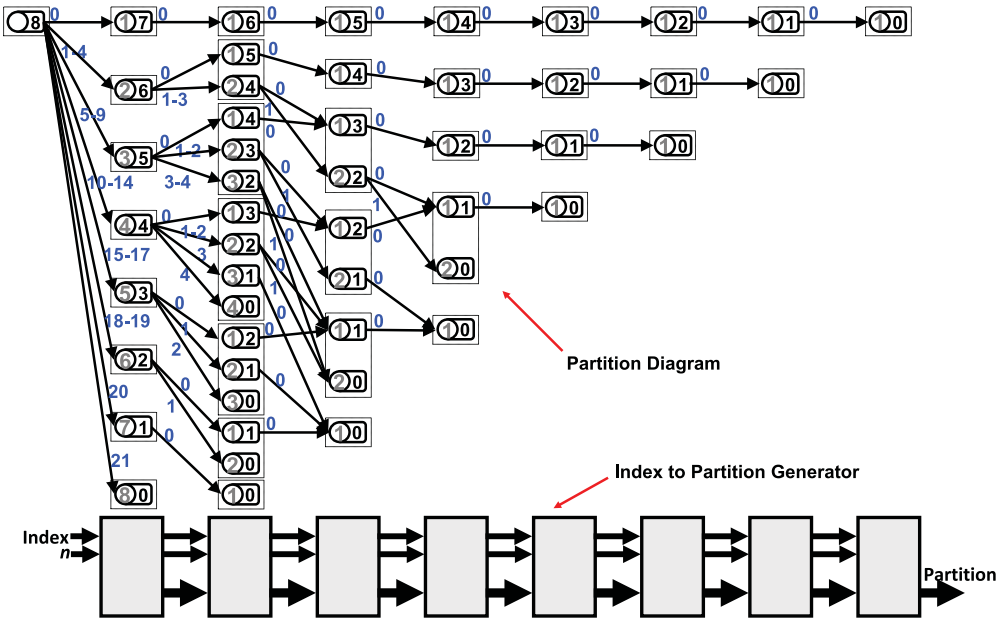

Fig. 5. Example of a partition diagram for integer partitions on $n = 8$.

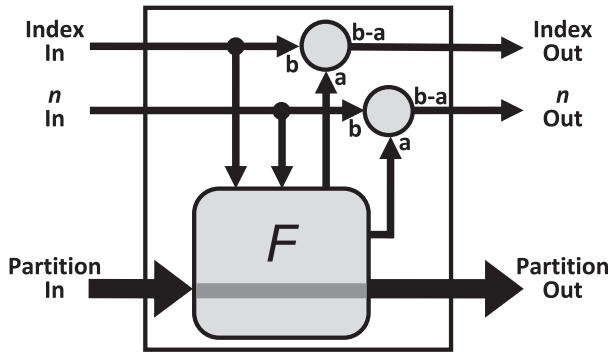

Fig. 6. One stage of the partition generator.

Example 3.2. Figure 5 shows the partition diagram for $n = 8$. On the left is a node that represents 8, while all subsequent nodes represent some part in a partition of n. For example, following the topmost paths yields the partition {1 1 1 1 1 1 1 1}. Following the bottom path yields the partition {8 – – – – – – –}.

The values of the input *index* are shown as labels of the edges from the root node to its eight daughter nodes. For example, the edge labeled 0 from the root node is the beginning of the path corresponding to the partition {1 1 1 1 1 1 1 1}. From Table V, this partition has an index of 0. From Figure 5, it can be seen that all partitions that begin with 2 have index 1 to 4, all that begin with 3 have index 5 to 9, and so forth. This can be checked by comparing Figure 5 with Table V.

3.2.2. Circuit Implementation. We propose a circuit that is an implementation of the partition diagram, which produces an integer partition on n from an index.

Table VI. Comparison of the Computation Times for a Single-Integer Partition on an Altera Stratix IV IV EP4SE530F43C3NES Versus a Xeon Microprocessor

n	# parti--tions	FPGA Freq (MHz)	FPGA time (ns)	Xeon time (ns)	Speed-up
3	3	401.0	2.5	1,067	427
4	5	401.0	2.5	1,680	672
5	7	297.4	3.4	2,743	816
6	11	249.3	4.0	4,400	1,100
7	15	249.6	4.0	7,280	1,820
8	22	136.7	7.3	11,091	1,516
9	30	133.5	7.5	17,253	2,303
10	42	115.7	8.6	25,829	2,988
11	56	111.8	8.9	38,821	4,340
12	77	93.0	10.8	55,725	5,182
13	101	84.3	11.9	80,990	6,827
14	135	70.5	14.2	113,526	8,004

Figure 6 shows one stage of the partition generator. There are three inputs, the index, n, and the partition. The internal block labeled F computes a value by which to reduce the index and subtracts that to produce the output index. It is an implementation of a part of the partition diagram, an example of which is shown in Figure 5. That is, each block in Figure 5 realizes that part of the partition diagram directly above the block. Specifically, it computes a value by which to reduce n and subtracts that to produce the output n. This reduction is precisely the new part produced by this stage, and that is also applied to the output partition. All other parts of the partition are unchanged. This is indicated by the shaded line across the internal block in Figure 6. In determining the new part from this stage, the internal block chooses that new part value to be less than or equal to the input n and less than or equal to the last part produced, as indicated by the input partition. The input index determines the new part value.

3.2.3. Results. A Verilog program was written to implement the index to integer partition converter described previously. The target technology was chosen as the Altera Stratix IV EP4SE530F43C3NES. Table VI compares the rate of processing partitions on this FPGA with the rate of an Intel Xeon microprocessor running at 2.8GHz with 512KB L2 cache and 1GB of memory. The data shown is from a C program that computes all integer partitions. It implements McKay's Partition Generator [McKay 1965], which generates integer partitions in lexicographical order beginning with $111\ldots1$. We compare this with Verilog code that runs at the frequency specified by Synplify Pro. Table VI shows the comparison, and the potential speedup as measured by the Xeon time divided by the Stratix IV time. For example, for 14-element partitions, the FPGA achieves a speedup of $8,004\times$ over the Xeon microprocessor.

Because the system's clock function provides only a crude measure of elapsed time when small time differences are computed, we repeatedly (redundantly) did the computations for many iterations on the Xeon microprocessor and divided the time durations by the number of iterations. For each n in Table VI, we chose 25,000 iterations.

3.2.4. Complexity of Implementation. Table VII shows the resource usage. This data comes from the Synopsys logic design tool, Synplify Pro, in which the FPGA chosen is the Altera Stratix IV EP4SE530F43C3NES.

The first column in Table VII shows n. The second column shows the frequency achieved. The third through seventh columns show how many LUTs are used in the implementation, while the eighth column shows the number of packed ALMs used.

Table VII. Frequency and Resources Used to Realize the Index to Integer Partition Implementation on the Altera Stratix IV EP4SE530F43C3NES FPGA

n	Freq. (MHz)	# of Comb Fnc	# LUTs With Various # of Inputs 7-	6-	5-	4-	3-	Estimated # of Packed ALMs	Total # of Registers
3	401.0	8	0	0	0	2	6	7 (0%)	13 (0%)
4	401.0	39	0	4	14	9	12	30 (0%)	33 (0%)
5	297.4	84	0	3	43	23	15	55 (0%)	51 (0%)
6	249.3	182	3	11	104	37	27	116 (0%)	77 (0%)
7	249.6	200	3	13	120	43	21	135 (0%)	95 (0%)
8	136.7	724	7	43	436	164	74	427 (0%)	167 (0%)
9	133.5	1,025	13	69	611	242	90	595 (0%)	179 (0%)
10	115.7	1,623	25	200	867	371	160	955 (0%)	196 (0%)
11	111.8	2,073	29	306	1,081	459	198	1,218 (0%)	207 (0%)
12	93.0	3,262	45	551	1,473	744	449	1,921 (0%)	247 (0%)
13	84.3	4,229	40	839	1,837	921	592	2,525 (0%)	276 (0%)
14	70.5	6,019	45	1,120	2,618	1,293	943	3,548 (1%)	319 (0%)

The ninth column shows the number of registers used. The last two columns show the percentage of available resources used. It is clear from this table that relatively few resources are used.

4. CONCLUDING REMARKS

The generation of set and integer partitions by hardware has important practical applications. The challenge is to generate these partitions at a rate of one per clock period. For set partitions, we show two ways to accomplish this. The first is a sequential circuit that generates the partitions in lexicographical order according to their restricted growth string. This circuit can produce partitions of large sets. The second circuit is an index to set partition converter. In this circuit, an up counter on the index input produces set partitions in increasing lexicographical order, while a down counter produces set partitions in decreasing lexicographical order. Also, a random number generator at the index produces random set partitions. This is useful for Monte Carlo simulations involving partitions. It is combinational but can be pipelined to produce a set partition at a rate of one per clock. An analysis of the complexity of these two circuits shows that the complexity of both grow polynomially with n, the number of elements in the partition, while the delay grows linearly with n. Also, for both circuits, we show experimental results from the Synopsys logic design tool, Synplify Pro, to show the FPGA resources used. Specifically, small to large circuits were modeled on the Altera Stratix IV EP4SE530F43C3NES FPGA. Our experimental results show that an FPGA running at 100MHz produces partitions at a rate that is about 10 times the rate of a software-implemented partition generator on a processor that runs at 2.26GHz.

Second, we show an index to integer partition converter that can produce partitions in various orders, for example, lexicographically and randomly. As in the case of set partitions, we demonstrate that there is a data structure, in this case, the partition diagram, that allows the design of a circuit that generates an integer partition at a rate of one per clock. Both the partition tree (for set partitions) and the partition diagram (for integer partitions) are tree data structures. In the two cases, they are essential in deriving the partition-generating circuit. Experimental analysis using Synplify Pro shows that it is also efficient compared to the software generation of integer partitions. Specifically, our hardware achieves an 8,000 times speedup compared to a software implementation.

Table VIII. Comparison of the Circuits Presented in This Article

Circuit	Complexity	Delay
Set partitions		
Sequential	Simple	Medium
Single stage	High	Small
Multi state	Medium	Medium
Integer partitions		
Multi stage	Medium	Medium

Table VIII summarizes the circuits presented in this article with respect to complexity and delay.

There are many avenues for future work. Especially, there is a need for hardware generation of set partitions with restrictions on the size and number of blocks and on integer partitions with distinct parts, with odd parts, with even parts, and so forth. An interesting survey on integer partitions [Wilf 2000] provides a starting point. Another extension is to investigate the simultaneous implementation of more than one hardware generator on one or more FPGAs. It would be interesting to compare this with a parallel software generation program on a cluster and/or GPU computer.

ACKNOWLEDGMENTS

We appreciate many detailed comments by five reviewers that improved the manuscript.

REFERENCES

Daniel Berend and Tamir Tassa. 2010. Improved bounds on Bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 30, 185–205.

Mireille Bousquet-Mélou and Kimmo Eriksson. 1997a. Lecture hall partitions (1). *Ramanujan Journal*, 1, 101–111.

Mireille Bousquet-Mélou and Kimmo Eriksson. 1997b. Lecture hall partitions (2). *Ramanujan Journal*, 1, 165–185.

Randall E. Bryant. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35, 677–691.

Jon T. Butler and Tsutomu Sasao. 2011. High-speed constant weight code generators. In *Proceedings of the 7th International Symposium on Applied Reconfigurable Computing (ARC'11), Lecture Notes in Computer Science (LNCS 6576)*, A. Koch et al. (Eds.). Springer-Verlag, Berlin, 193–204.

Jon T. Butler and Tsutomu Sasao. 2012. Hardware index to permutation converter. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*. 424–429.

Jon T. Butler and Tsutomu Sasao. 2013a. Combinatorial computing - One object per clock. In *Proceedings of the 2013 Reed-Muller Workshop*. Toyama, Japan, 66–74.

Jon T. Butler and Tsutomu Sasao. 2013b. Hardware index to set partition converter. In *9th International Workshop on Applied Reconfigurable Computing (ARC2013), Proceedings Lecture Notes in Computer Science (LNCS 7806)*. Springer-Verlag, Los Angeles, CA, 72–83.

Xin Chen, Lan Liu, Zheng Liu, and Tao Jiang. 2008. On the minimum common integer partition problem. In *ACM Transactions on Computational Logic*, V, 1–9.

R. William Gosper. 1972. Item 175 in Beeler, M., Gosper, R. W., and Schroeppel, R., "HAKMEM." Cambridge, MA: MIT Artificial Intelligence Laboratory. http://www.inwap.com/pdp10/hbaker/hakmem/hacks.html\#item175 Memo AIM-239.

Robin K. S. Hankin and Luke J. West. 2007. Set partitions in R. *Journal of Statistical Software*, 23, 99–100.

Shin-Ichiro Kawano and Shin-Ichi Nakano. 2005. Constant time generation of set partitions. *IEICE Trans. Fundamentals*, E88-A, 930–934.

Dominque Lavenier and Yannick Saouter. 1998. Computing Goldbach partitions using pseudo-random bit generator operators on an FPGA systolic array. In *Proceedings of the 8th International Workshop on Field Programmable Languages FPL'98, Proceedings Lecture Notes in Computer Science (LNCS 1482)*, R. W. Hartenstein and A. Keevallik (Eds.). Tallinn, Estonia, 316–325.

Rung-Bin Lin. 2006. On the applications of partition diagrams for integer pertitioning. In *Proceedings of the 23rd Workshop on Combinatorial Mathematics and Computation Theory*, Vol. 7. Chang-Hua, Taiwan, 349–362.

John K. S. McKay. 1965. Algorithm 263, partition generator. *Communications of the ACM*, 8, 493.

Shinobu Nagayama, Tsutomu Sasao, and Jon T. Butler. 2012. Analysis of multi-state systems with multi-state components using EVBDDs. In *Proceedings of the 42nd International Symposium on Multiple-Valued Logic*. Victoria, Canada, 122–127.

B. John Oommen and David T. H. Ng. 1990. On generating random partitions with arbitrary distributions. *The Computer Journal*, 33, 368–374.

Michael Orlov. 2002. Efficient generation of set partitions. Technical Report. Retrieved from https://www.cs.bgu.ac.il/∼orlovm/papers/partitions.pdf, 1–6.

Edward M. Reingold, Jurg Nivergelt, and Narsingh Deo. 1977. *Combinatorial Algorithms, Theory and Practice*. Prentice-Hall.

Ichiro Semba. 1984. An efficient algorithm for generating all partitions of the set $\{1, 2, \ldots, n\}$. *Journal of Information Processing*, 7, 424–436.

Jennifer L. Shafer, Stuart Schneider, Jon T. Butler, and Pantelimon Stănică. 2010. Enumeration of bent Boolean functions by reconfigurable computer. In *Proceedings of the 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines*. 265–272.

Ivan Stojmenovič. 1990. An optimal algorithm for generating equivalence relations on a linear array of processors. *BIT*, 30, 424–436.

Terasic. 2011. High Speed FPGA-based Bio-computing Platform: FPGA Development Kits an Ideal Platform for Intensive Gene and Protein Calculations. Retrieved from http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=613.

Herbert S. Wilf. 2000. Lectures on Integer Partitions. Retrieved from http://www.math.upenn.edu/wilf/PIMS/PIMSLectures.pdf.

David P. Woodruff. 2006. Better approximations for the minimum common integer partition problem. In *Approximations, Randomizations, and Combinatorial Optimizations: Algorithms and Techniques*, J. Diaz et al. (Eds.). Springer-Verlag, Berlin, 248–259.

Antoine Zoghbi and Ivan Stojmenovič. 1998. Fast algorithms for generating integer partitions. *International Journal on Computer Mathematics* 70, 319–332.

Received June 2013; revised September 2013; accepted January 2014