Hardware Index to Set Partition Converter

Jon T. Butler[1] and Tsutomu Sasao[2,⋆]

[1] Naval Postgraduate School, Monterey, CA, 93921-5121, USA
jon_butler@msn.com
[2] Kyushu Institute of Technology, Iizuka, Fukuoka, 820-8502, Japan
sasao@ieee.org

Abstract. We demonstrate, for the first time, high-speed circuits that generate partitions on a set S of n objects. We offer two versions. 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 needed in the hardware implementation of the optimum distribution of tasks to processors. Our circuits are combinational. For large n, they can have large delay. However, one can easily pipeline them to produce one set partition per clock period. We show 1) analytical and 2) experimental time/complexity results that quantify the efficiency of our designs. Our results show that a hardware partition generator running on a 100 MHz FPGA produces partitions at a rate that is approximately 10 times the rate of a software implementation on a processor running at 2.26 GHz.

1 Introduction

A partition of a set S is the placement of elements of S into blocks. For example, there are 15 partitions of four distinct 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 $\{\{0,2\},\{1,3\}\}$ are identical. The number of partitions increases rapidly as the number of elements 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 4140. Bell numbers have the property that, for large n, $B(n)$ is approximated by $(\frac{n}{\ln n})^n$ [8], p. 64.

Partitions are important combinatorial objects. For example, partitions on n elements enumerate the equivalence relations on n elements [17]. Each block represents all elements related by the equivalence relation.

⋆ This research is supported by a Grant-in-Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS).

One way to generate all partitions is to generate all binary numbers, one per clock, discarding those that are not partitions. However, only a few of these numbers are partitions. This approach produces partitions at a rate that is much slower than one partition per clock. Therefore, we seek a circuit that produces one partition per clock, where the input is an index to the partitions.

The ability to generate partitions has important practical applications. Hankin and West [7] 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. Recent research in computational molecular biology has shown the importance of partitions in understanding the role of genes in determining global characteristics of species. For example, Chen, Liu, Liu, and Jiang [4] have identified the importance of solving the minimum common integer partition (MCIP) problem in ortholog assignment and DNA fingerprint assembly. This problem requires the enumeration of partitions at high speed, since so many partitions must be considered. In multi-state distribution *systems* (packet, water, gas, etc.) [11], 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.

This paper can be viewed as a companion to [2], which describes the high-speed generation of combinations, as well as the generation of random combinations for use in reconfigurable computers. It can also be viewed as a companion to [3], which describes the high-speed generation of permutations, as well as the generation of random permutations. Together these three papers cover a subset of circuits that produce *combinatorial objects*. The advent 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. Much has been written about generating combinatorial objects in software (e.g. [6], [8] pp. 5-6). Indeed, there are many papers on programs and algorithms for enumerating partitions [6,9,10,12,14,16], including parallel algorithms [17]. However, as far as we know, there has not been a hardware enumeration of partitions. This paper addresses that deficiency.

2 Definitions

2.1 Introduction

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

For example, $\{\{3, 1\}, \{2, 0\}\}$ is a partition on the 4-set $S = \{0, 1, 2, 3\}$. 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 (0101). Formally,

Definition 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* $b_0 = 0$[1].

The first element of a restricted growth string is always 0, signifying that element $n-1$ is always in block 0. A special characteristic of a restricted growth string is that each element is between 0 and 1 plus the maximum of all lower elements.

Lemma 1. *[13] 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. Especially, a circuit exists to convert restricted growth strings into partitions. Table 1 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 (0-th) block. The second element shows where element 2 is located (in the 0-th or 1-st block). The third element shows where element 1 is located (in the 0-th, 1-st, or 2-nd block). The fourth element shows where element 0 is located (in the 0-th, 1-st, 2-nd, or 3-rd block).

In order to deduce the circuit needed to produce a set partition from an index, we introduce the set 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 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 along the path from the root node to j, there is no more nodes than n, and, for all node labels k,* $j \leq \max\{k\} + 1$.

[1] The term restricted growth *function* is also used to describe this.

Table 1. 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)

A terminal node is simply the last node along a path from the root node. In a set partition tree, the restricted growth string of a partition is 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 1. *Fig. 1 shows the set partition tree for partitions with $n = 4$ elements. 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,*

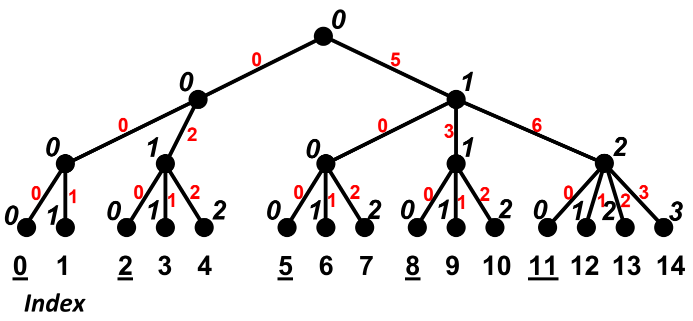

Fig. 1. Example of a Set Partition Tree for Set Partitions on Four Elements

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.* *(End of Example)*

The set partition tree is similar to a decision tree. Each node has children nodes corresponding to all possible choices at that point. Each terminal node corresponds to a partition. Fig. 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.

3 Circuit Implementations

3.1 Sequential Circuit Implementation

Fig. 2 shows a sequential circuit implementation of a set partition converter. A clock comes in at the right. At each clock pulse, this circuit produces the next set partition in increasing lexicographical order according to the restricted growth string. Specifically, it first generates $(b_0 \dots b_{n-2}b_{n-1}) = (0\dots000)$, then $(0\dots001)$, etc.. At each stage, `Counter` counts up to a maximum value allowed in a restricted growth string representation. At this point, it cycles back to 0, just as is done in a conventional counter digit. 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.

3.2 Single-Stage Combinational Circuit Implementation

Fig. 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 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 LSD or least significant digit. The threshold values in Fig. 3 are determined by the set partition tree shown in Fig. 1. They correspond to the indices associated with the 0 terminal nodes in Fig. 1. The corresponding indices are underlined in Fig. 1. There is only one stage in this implementation. As is discussed later, the number of comparators grows is approximated by $(\frac{n}{\ln(n)})^n$.

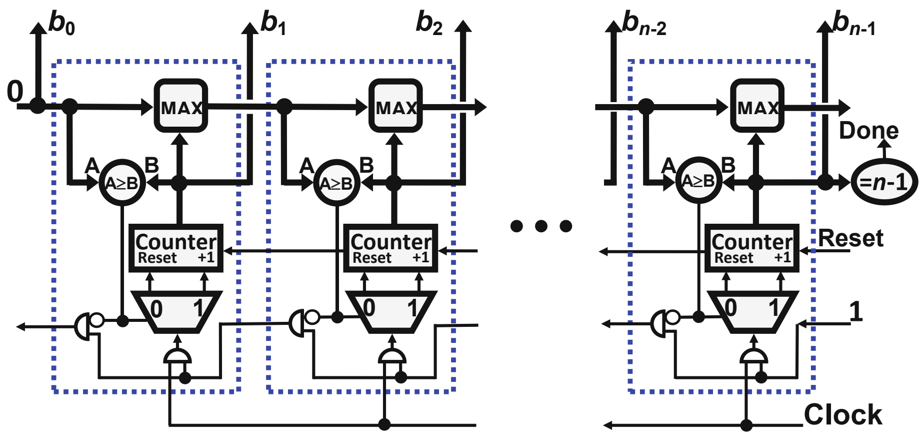

Fig. 2. Sequential Set Partition Generator

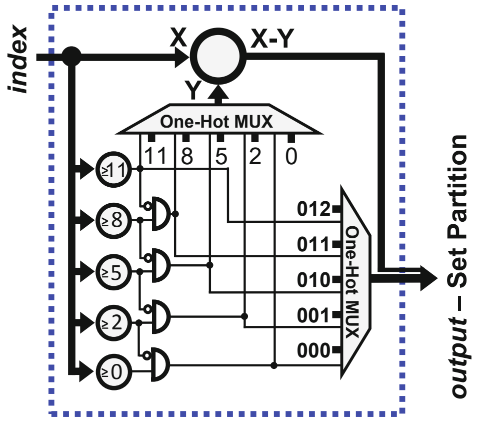

Fig. 3. Single-Stage Index to Set Partition Circuit for $n = 4$

3.3 Multi-stage Combinational Circuit Implementation

Fig. 4 shows the multi-stage index to set partition converter. Here, the index comes in on the left and is modified as it passes through the stages. 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 Fig. 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 Fig. 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. 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.

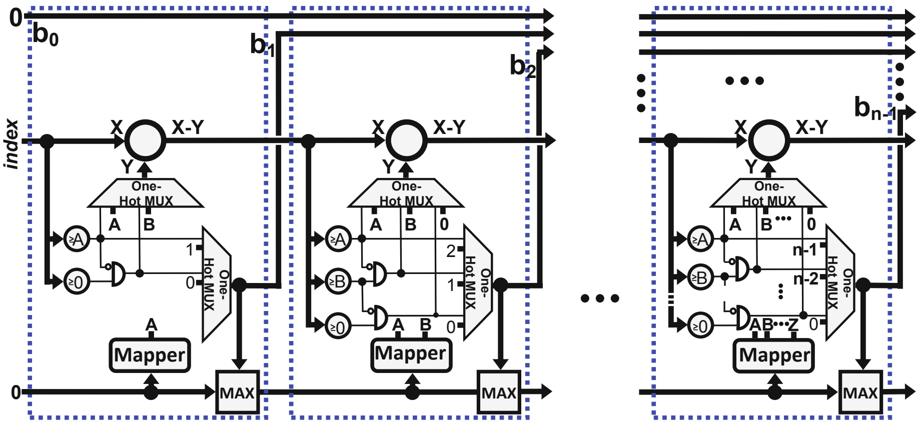

Fig. 4. Multi-Stage Index to Set Partition Circuit

Note that, in a multi-stage 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 the more compact for general n. This is addressed in the next section.

3.4 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 comparator it contains. Note that, in making this assumption, we assume that the delay and circuit complexity for comparators and multipliers remains constant as n varies.

Lemma 2. *The number of comparators C_i used in a set partition generator is*

1) sequential (Fig. 2): $C_1 = O(n)$,
2) single-stage (Fig. 3): $C_2 = O\left(\left(\frac{n}{\ln(n)}\right)^n\right)$, *and*
3) multi-stage (Fig. 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.

In the case of the set partition tree, 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 [1], we have $B(n-1) < \left(\frac{0.792(n-1)}{\ln(n)}\right)^{n-1}$. Thus,

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

In the case of the set partition tree, the first (leftmost) block has 2 comparators. The next block has 3, the next 4, etc.. 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

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

∎

It is clear from Lemma 2 that the multi-stage index to set partition converter has many fewer comparators than the single-stage converter, especially in the case of set partitions on many elements. Thus, the case for $n = 4$ discussed at the end of Section 3.3 is an aberration. We can also compare the circuits on the basis of their delay.

Lemma 3. *The delay D_i in a set partition generator is*

1) sequential (Fig. 2): $D_1 = O(n)$,
2) single-stage (Fig. 3): $D_2 = O(1)$, and
3) multi-stage (Fig. 4): $D_3 = O(n)$.

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 set partition tree, there is exactly one stage, and the delay is independent of n. Thus, this circuit has delay $O(1)$. In the case of the compact set partition tree, the index must propagate through $n-2$ stages. Thus, the delay is $O(n)$. ∎

Note that, in these calculations, we considered the multi-stage index to set partition converter to be *combinational.* When n is large, this circuit has 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 multi-stage circuit with the combinational circuit of the single-stage circuit. As a result, the time comparisons will be (significantly) different from the derived delay.

3.5 Experimental Data

In the analysis above, we used the number of comparators as a measure for the complexity. In this section, we use actual FPGA resources. We synthesized the three circuits discussed above on the Altera Stratix IV EP4SE530F43C3NES FPGA. Table 2 shows the resource usage for the sequential version.

Table 2. 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 vs # inputs 7-	6-	5-	4-	3-	Est. # 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	1923	10	477	659	298	479	1146(0%)
128	1.12×10^{158}	526	896	11.0	91.278	3847	13	727	1666	427	1014	2134(1%)

From Table 2, for all values of $n \leq 16$, the achieved frequency exceeds 100 MHz. Thus, the sequential partition generator produces one partition per clock period for all $n \leq 16$, where the clock period is 10 ns.. To compare this rate to a software implementation of a sequential partition generator, we adapted Orlov's [13] program and ran it on an Intel®Core$^{\text{TM}}$2 Duo P8400 processor running at 2.26 GHz. For 8 and 16 element partitions, we achieve a rate of partitions of one per 94 ns. and 156 ns., respectively. This represents a 9.4 and 15.6 times speed-up realized by the hardware version over the software version.

The first column in Table 2 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 the synthesis tool, 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 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 twelfth columns show the number of the various lookup tables (LUTs) that were used. The thirteenth column shows the number of packed ALMs used in the realization. The columns that represent *general* characteristics, including the number of combinational functions and the number of packed ALMs, show an approximate doubling of resources used with each doubling of n. This suggests a linear relationship between these resources and n. It correlates with the observed linear relationship between the number of comparators measure used in the previous section and n. Because of the large number of partitions, for moderate n (e.g., $n = 32$), it will be too time consuming to enumerate all partitions at typical FPGA clock frequencies (e.g., 100 MHz). 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, if the index is random and the partitions are used in Monte Carlo simulations.

Table 3 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 Fig. 3. As discussed, this has short delay paths. This is indicated by the frequency, 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 3 by the near 5-fold increase in the number of combinational logic circuits and by the nearly 5-fold increase in the number of 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 that the values of n achieved for the sequential and multi-stage circuits. In comparing the delay of the single-stage combinational logic index to the set partition converter with the multi-stage circuit below, it is important to recall that, unlike the multi-stage circuit, the single-stage circuit is not pipelined. Thus, the multi-stage circuit will achieve a higher clock speed. However, its latency will be larger.

Table 3. Frequency/resources used to realize the single stage index to set partition converter on the Altera Stratix IV EP4SE530F43C3NES FPGA

n	# Set Partitions	In # Bits	Out # Bits	Freq. (MHz)	Delay ns.	# Comb Fnc	# LUTs vs # inputs 7-	6-	5-	4-	3-	Est. # of Packed ALMs
4	15	4	8	406.3	2.461	5	0	0	0	5	-	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	4100	32	1416	1223	652	777	2628(1%)

Table 4 shows the FPGA resources and frequency achieved on the Altera Stratix IV EP4SE530F43C3NES FPGA by the multi-stage index to set partition circuit shown in Fig. 4. This uses fewer resources than the single-stage circuit in Fig. 3, but its latency is greater. In the design of the multi-stage 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 $n - 1$ clock periods.

Table 4. Frequency/resources used to realize the multi-stage index to set partition converter on the Altera Stratix IV EP4SE530F43C3NES FPGA

n	# Set Partitions	In # Bits	Out # Bits	Freq. (MHz)	Delay ns.	# Comb Fnc	# LUTs vs # inputs					Est. # of Packed ALMs
							7-	6-	5-	4-	3-	
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	3842	35	718	1524	1130	435	2339(0%)
32	1.28×10^{26}	88	160	55.6	17.973	38305	87	3671	19768	8016	6763	21206(9%)

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 multi-stage 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.

4 Concluding Remarks

To the best of our knowledge, our circuits are the first hardware implementations of set partition generators. The generation of set partitions by hardware has important practical applications. The challenge is to generate set partitions at one per clock period. 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 is fast and 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. It is combinational, but can be pipelined to produce a set partition at one per clock. An analysis of the complexity of these two circuits show 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 that confirm these predictions. Specifically, small to large circuits were implemented on the Altera Stratix IV EP4SE530F43C3NES FPGA. Our experimental results show that an FPGA running at 100 MHz produces partitions at a rate that is about 10 times the rate of a software implemented partition generator on a processor that runs at 2.26 GHz.

Acknowledgements. We thank six referees for comments that improved this paper.

References

1. Berend, D., Tassa, T.: Improved bounds on Bell numbers and on moments of sums of random variables. Probability and Mathematical Statistics 30(2), 185–205
2. Butler, J.T., Sasao, T.: Index to Constant Weight Codeword Converter. In: Koch, A., Krishnamurthy, R., McAllister, J., Woods, R., El-Ghazawi, T. (eds.) ARC 2011. LNCS, vol. 6578, pp. 193–205. Springer, Heidelberg (2011)
3. Butler, J.T., Sasao, T.: Hardware index to permutation converter. In: 19th Reconfigurable Architectures Workshop (RAW 2012), Proc. of the 26th IEEE International Parallel and Distributed Processing Symposium, Shanghai, China, May 21-22, pp. 424–429 (2012)
4. Chen, X., Liu, L., Liu, Z., Jiang, T.: On the minimum common integer partition problem. ACM Trans. on Computational Logic V, 1–19 (2008)
5. Debnath, D., Sasao, T.: Fast Boolean matching under permutation by efficient computation of canonical form. IEICE Trans. Fundamentals (12), 3134–3140 (2004)
6. Beeler, M., Gosper, R.W., Schroeppel, R.: HAKMEM. MIT Artificial Intelligence Laboratory, Cambridge, MA, Memo AIM-239 (February 1972), `http://www.inwap.com/pdp10/hbaker/hakmem/hacks.html#item175`
7. Hankin, R.K.S., West, L.J.: Set partitions in *R*. J. of Statistical Software 23, Code Snippet 2 (December 2007), `http://www.jstatsoft.org/`
8. Knuth, D.E.: Volume 4 Generating all combinations and permutations. In: The Art of Computer Programming, Fascicle 3. Addison-Wesley ISBN: 0-321-58050-8
9. Kawano, S., Nakano, S.: Constant time generation of set partitions. IEICE Trans. Fundamentals E88-A(4), 930–934 (2005)
10. McKay, J.K.S.: Algorithm 263, Partition Generator. Communications of the ACM 8(8), 493 (1965)
11. Nagayama, S., Sasao, T., Butler, J.T.: Analysis of multi-state systems with multi-state components using EVBDDs. In: Proc. 42nd International Symposium on Multiple-Valued Logic, Victoria, Canada, May 14-16, pp. 122–127 (2012)
12. Oommen, B.J., Ng, D.T.H.: On generating random partitions with arbitrary distributions. The Computer Journal 33(4), 368–374 (1990)
13. Orlov, M.: Efficient generation of set partitions (March 2002), `http://www.cs.bgu.ac.il/~orlovm/papers/partitions.pdf`
14. Reingold, E., Nivergelt, J., Deo, N.: Combinatorial Algorithms, Theory and Practice. Prentice-Hall (1977)
15. Sasao, T.: Memory Based Logic Synthesis, 1st edn. Springer (2011) ISBN: 978-1-4419-8103-5
16. Semba, I.: An efficient algorithm for generating all partitions of the set $\{1,2,..., n\}$. Journal of Information Processing 7(1) (1984)
17. Stojmenovič, I.: An optimal algorithm for generating equivalence relations on a linear array of processors. BIT 30(3), 424–436 (1990)