2012-05

Hardware index to permutation converter

Sasao, T.; Butler, Jon T.

19th Reconfigurable Architectures Workshop, May 21-22, 2012, Shanghai, China.
http://hdl.handle.net/10945/35852

Hardware Index to Permutation Converter

J. T. Butler
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA U.S.A.

T. Sasao
Department of Computer Science & Electronics
Kyushu Institute of Technology
Iizuka, Fukuoka, Japan

***Abstract*—We demonstrate a circuit that generates a permutation in response to an index. Since there are $n!$ n-element permutations, the index ranges from 0 to $n! - 1$. Such a circuit is needed in the hardware implementation of unique-permutation hash functions to specify how parallel machines interact through a shared memory. Such a circuit is also needed in cryptographic applications. The circuit is based on the factorial number system. Here, each non-negative integer is uniquely represented as $s_{n-1}(n-1)! + s_{n-2}(n-2)! + \ldots + s_1 1!$, where $1 \leq s_i \leq i$. That is, the permutation is produced by generating the digits s_i in the factorial number system representation of the index. The circuit is combinational and is easily pipelined to produce one permutation per clock period. We give experimental results that show the efficiency of our designs. For example. we show that the rate of production of permutations on the SRC-6 reconfigurable computer is 1,820 times faster than a program on a conventional microprocessor in the case of 10-element permutations. We also show an efficient reconfigurable computer implementation that produces random permutations using the Knuth shuffle algorithm. This is useful in Monte Carlo simulations. For both circuits, the complexity is $\mathrm{O}(n^2)$, and the delay is $\mathrm{O}(n)$.**

Keywords: **reconfigurable computer, index to permutation generator, random permutation generator, combinatorial objects, factorial number system, Knuth shuffle**

I. Introduction

One of the most important combinatorial objects is the *permutation*. For example, 2103 is a permutation where 0 maps to 2, 1 maps to 1, 2 maps to 0, and 3 maps to 3 (i.e., 0 and 2 are interchanged). This can be expressed as the 8-bit binary number 10 01 00 11. The circuit we seek has an input that represents the permutation's index and an output that represents the permutation itself.

One way to generate all permutations is to generate all 8-bit bit binary numbers, one per clock, discarding those that are not permutations. However, this produces permutations at a rate that is much slower than one permutation per clock. We seek a circuit that produces one permutation per clock.

The ability to generate permutations has important practical applications. Permutations are important in cryptographic algorithms [17]. Permutations are used to create diffusion, where information in the plaintext is spread out across the ciphertext. For example, there are six permutations in DES, two in Twofish and two in Serpent [18].

The ability to generate various permutations is useful in obtaining near-optimum data compression of image data obtained by satellite [1], [13]. When permutations are compressed using internal regularities, then succinct encodings of integer functions, strings and binary relations are possible [2].

Permutations can be used to reorder data streams in FPGA-based digital signal processing engines [15]. This can be used to automatically generate efficient parallel pipelined FFT architectures.

There is a need for hardware generation of permutations in the implementation of unique-permutation hash functions to specify how parallel machines interact through a shared memory [6]. Such hash functions yield the minimal possible contention, as they probe each location with the same probability regardless of which locations are currently occupied. They are especially important now that multi-core (CPU) and many-core (GPU) architectures are commonly available.

The complexity of the binary decision diagram (BDD) [3] is strongly dependent on the order in which variables are applied. For example, the BDD of the 'Achilles Heel' function has polynomial number of nodes for the optimum ordering and exponential number of nodes for the worst case ordering. Determining the optimum ordering involves the generation of typically many permutations, testing how many nodes are required for each.

Two Boolean functions are P-equivalent if they differ only by a permutation of variables. In [5], a breadth-first search technique is shown for computing the P-representative of a given function, which is the canonical form of a Boolean function under permutation of the variables. Such a classification is useful in a lookup table implementation of Boolean functions. This advance was made in the software implementation, but a faster hardware implementation requires hardware generation of permutations.

In Section 2, we discuss the factorial number system. We show how it can be used to generate permutations, and we discuss its circuit implementation. Then, in Section 3, we discuss the generation of *random* permutations in a circuit implementing the Knuth shuffle algorithm. Finally, in Section 4, we give concluding remarks. This paper can be viewed as a companion to [4] which describes the high-speed generation of combinations, as well as the generation of random combinations. Together the two 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. Indeed, much has been written about generating combinatorial objects in software (e.g. [7], [9], [10]).

TABLE I
THE FACTORIAL NUMBER SYSTEM FOR $n = 4$

N	$s_3 s_2 s_1 s_0$	Value of N	Permutation
23	3 2 1 0	$3\ 3! + 2\ 2! + 1\ 1! + 0\ 0! = 18 + 4 + 1 + 0$	3210
22	3 2 0 0	$3\ 3! + 2\ 2! + 0\ 1! + 0\ 0! = 18 + 4 + 0 + 0$	3201
21	3 1 1 0	$3\ 3! + 1\ 2! + 1\ 1! + 0\ 0! = 18 + 2 + 1 + 0$	3120
20	3 1 0 0	$3\ 3! + 1\ 2! + 0\ 1! + 0\ 0! = 18 + 2 + 0 + 0$	3102
19	3 0 1 0	$3\ 3! + 0\ 2! + 1\ 1! + 0\ 0! = 18 + 0 + 1 + 0$	3021
18	3 0 0 0	$3\ 3! + 0\ 2! + 0\ 1! + 0\ 0! = 18 + 0 + 0 + 0$	3012
17	2 2 1 0	$2\ 3! + 2\ 2! + 1\ 1! + 0\ 0! = 12 + 4 + 1 + 0$	2310
16	2 2 0 0	$2\ 3! + 2\ 2! + 0\ 1! + 0\ 0! = 12 + 4 + 0 + 0$	2301
15	2 1 1 0	$2\ 3! + 1\ 2! + 1\ 1! + 0\ 0! = 12 + 2 + 1 + 0$	2130
14	2 1 0 0	$2\ 3! + 1\ 2! + 0\ 1! + 0\ 0! = 12 + 2 + 0 + 0$	2103
13	2 0 1 0	$2\ 3! + 0\ 2! + 1\ 1! + 0\ 0! = 12 + 0 + 1 + 0$	2031
12	2 0 0 0	$2\ 3! + 0\ 2! + 0\ 1! + 0\ 0! = 12 + 0 + 0 + 0$	2013
11	1 2 1 0	$1\ 3! + 2\ 2! + 1\ 1! + 0\ 0! = 6 + 4 + 1 + 0$	1320
10	1 2 0 0	$1\ 3! + 2\ 2! + 0\ 1! + 0\ 0! = 6 + 4 + 0 + 0$	1302
9	1 1 1 0	$1\ 3! + 1\ 2! + 1\ 1! + 0\ 0! = 6 + 2 + 1 + 0$	1230
8	1 1 0 0	$1\ 3! + 1\ 2! + 0\ 1! + 0\ 0! = 6 + 2 + 0 + 0$	1203
7	1 0 1 0	$1\ 3! + 0\ 2! + 1\ 1! + 0\ 0! = 6 + 0 + 1 + 0$	1032
6	1 0 0 0	$1\ 3! + 0\ 2! + 0\ 1! + 0\ 0! = 6 + 0 + 0 + 0$	1023
5	0 2 1 0	$0\ 3! + 2\ 2! + 1\ 1! + 0\ 0! = 0 + 4 + 1 + 0$	0321
4	0 2 0 0	$0\ 3! + 2\ 2! + 0\ 1! + 0\ 0! = 0 + 4 + 0 + 0$	0312
3	0 1 1 0	$0\ 3! + 1\ 2! + 1\ 1! + 0\ 0! = 0 + 2 + 1 + 0$	0231
2	0 1 0 0	$0\ 3! + 1\ 2! + 0\ 1! + 0\ 0! = 0 + 2 + 0 + 0$	0213
1	0 0 1 0	$0\ 3! + 0\ 2! + 1\ 1! + 0\ 0! = 0 + 0 + 1 + 0$	0132
0	0 0 0 0	$0\ 3! + 0\ 2! + 0\ 1! + 0\ 0! = 0 + 0 + 0 + 0$	0123

II. THE FACTORIAL NUMBER SYSTEM

A. Introduction

In the factorial number system, an integer is represented as an ordered sequence of digits. While Knuth [8] may have been the first to use the term "factorial number system", its properties have been known as long ago as 1888 [11]. As in the standard binary number system, in the factorial number system, each number is represented by a unique vector of basis values.

Definition 1. *In a* **factorial number system**, *integer $N < n!$ is represented as $N = s_{n-1}s_{n-2}\ldots s_2 s_1$, where*

$$N = s_{n-1}(n-1)! + s_{n-2}(n-2)! + \ldots + s_1 1! + s_0 0!, \quad (1)$$

such that $0 \leq s_i \leq i \leq n-1$.

Note that $s_0 \equiv 0$. We will however, retain s_0 as a placeholder.

Example 1. *Table I shows the representation of numbers in the factorial number system for $n = 4$, where $0_{10} \leq N \leq 23_{10}$. The leftmost column shows N in decimal, the second column shows the vector representation of N, and the third column shows the value of N as computed from the vector according to (1). The rightmost column of Table I shows the corresponding permutation.*

(End of Example)

We can make the following observations.

1) The largest value of the index N_{max} is represented by the digits $n-1\ n-2\ \ldots 2\ 1\ 0$ and is given as $N_{max} = \sum_{i=0}^{n-1} i\ i! = n! - 1$. Denote $n-1\ n-2\ \ldots 2\ 1\ 0$ as the *vector representation* of N_{max}.
2) A vector representation in which the elements are less than or equal to the maximum value $n-1\ n-2\ \ldots 2\ 1\ 0$ represents a unique number in this number system.
3) Given N, a greedy algorithm derives its vector representation, $s_{n-1}\ s_{n-2}\ \ldots\ s_1\ s_0$. For example, in Table I, the left digit is the maximum s_{n-1} such that $s_{n-1}\ (n-1)! \leq N$. Then, we form $N - s_{n-1}(n-1)!$ and repeat for s_{n-2}, etc.

B. Circuit Implementation

We propose a circuit that produces a permutation on n from an index that is an implementation of the factorial number system. It is based on the above three observations. Consider, for example, the generation of permutations for $n = 4$. There are 24 such permutations indexed from 0 to 23. Let $index$ be a 5-bit quantity that specifies which of the 24 permutations should occur at the circuit output. Each of the four elements of the permutation is an integer between 0 and 3, and is represented by two bits. Therefore, $output$ is an 8-bit binary number representing the permutation corresponding to input *index*.

Fig. 1 shows the circuit that computes the description above. $index$ comes in from the left, and $output$ exits to the right. There is an input permutation, which is typically fixed (e.g. as the identity permutation). There are $n-1$ stages. Each stage uses the running index to compute the next element of the permutation. The leftmost stage computes the first element, the next stage the next element, etc. For example, the left stage has *index* as input, and uses that to compute the first element.

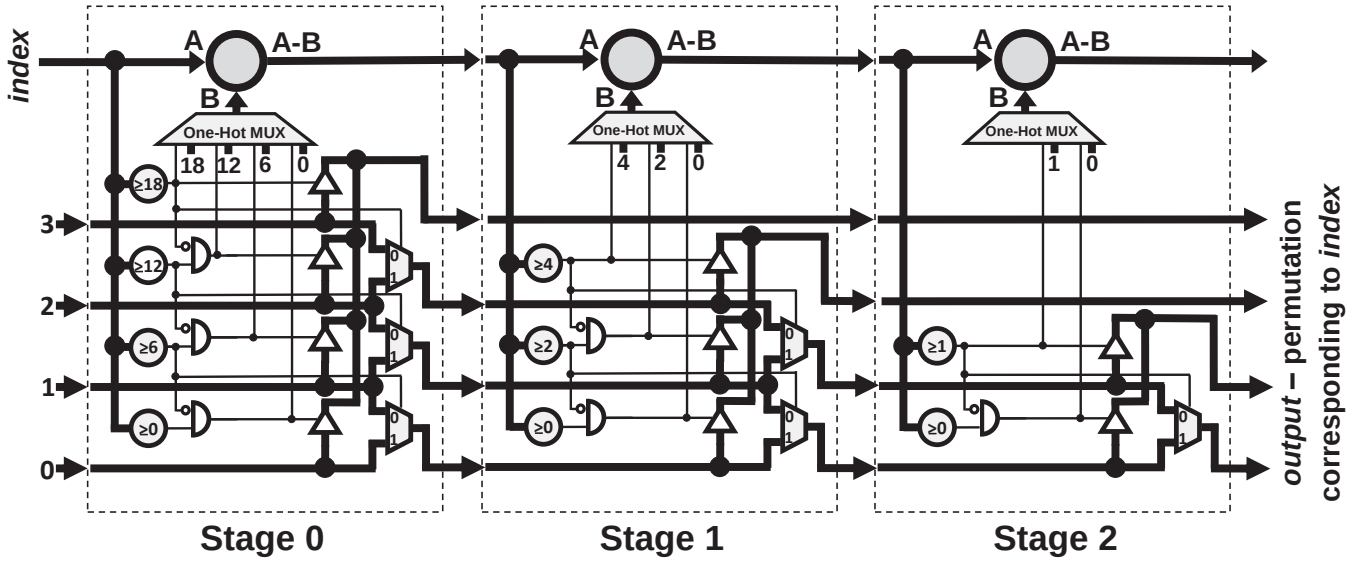

Fig. 1. Index to Permutation Circuit for 4-Element Permutations.

It passes this first element on to the next stage along with the remaining unassigned elements. This next stage computes the next element and passes the top two elements onto the third stage along with the remaining unassigned elements. At this point, there are two unassigned elements. The right stage then outputs these two elements interchanged or not depending on the next digit in the running index. Note that this circuit can be implemented as an LUT cascade [16].

At each stage of the LUT cascade, there are inputs and outputs that carry a partially completed $output$. Also, there are inputs and outputs that carry $index$ reduced by the values contributed by higher order digits. The rightmost stage produces a 0 value at it $index$ output, since there are no digits to the right.

The circuit shown in Fig. 1 is combinatorial. Note that this is easily pipelined. Pipeline registers can simply be inserted between stages. Doing so, causes the latency to be $n-1$. Note that, after the first codeword emerges, a codeword emerges at each clock period.

C. Results

A Verilog program was written to implement the index to permutation converter described above. It runs on an SRC-6 reconfigurable computer from SRC Computers and uses the Xilinx Virtex2p (Virtex2 Pro) XC2VP100 FPGA with Package FF1696 and Speed Grade -5. Table II compares the rate of processing permutations on this FPGA with the rate of a typical microprocessor. In this case, we chose the Intel Xeon microprocessor that is one of two microprocessors on the SRC-6. A C program was written that implements the factorial number system and produces, in sequence, the corresponding permutations. Table II shows that the circuit described by the Verilog code executes significantly faster on the SRC-6 than the C code on the microprocessor. For example, for 10-element permutations, the SRC-6 computes at a rate that is 1,820 faster than the Xeon microprocessor.

TABLE II
COMPARISON OF THE COMPUTATION TIMES FOR A SINGLE PERMUTATION ON THE SRC-6 RECONFIGURABLE COMPUTER VERSUS A XEON MICROPROCESSOR

n	SRC-6 time (ns)	Xeon time (ns)	# iterations
2	10	195	2,000,000
3	10	523	2,000,000
4	10	1,230	2,000,000
5	10	2,446	2,000,000
6	10	4,154	500,000
7	10	6,571	25,000
8	10	9,571	5000
9	10	13,401	250
10	10	18,203	25

For the SRC-6, the 10 ns time shown in the column labeled "SRC-6 time" is due to the fact that a single permutation is computed from the index in one clock period of a 100 MHz clock. The column labeled "Xeon time" represents processor time of 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 and divided the time durations by the number of iterations. The number of iterations is shown in the column labeled "# iterations". In the circuit described by the Verilog code, each permutation was represented by a single word. Here, each word has $n\lceil \log_2(n) \rceil$ bits, which is 36 for $n = 9$, for example. Although a word width of 36 is easily accommodated in an FPGA, it is not in a microprocessor. Therefore, in the C code version, we chose each permutation to be represented as an array of `int`s.

D. Complexity of Implementation

In order to compare how the logic resources depend on n, we synthesized the index to permutation circuit on the Altera

Stratix IV EP4SE530F43C3NES FPGA (this is the FPGA used on the SRC-7, an upgrade to the SRC-6). Table III shows the resource usage. For example, even though the SRC-6's clock frequency is much lower than the microprocessor's, its rate of generation is considerably faster.

The first column in Table III shows n. The second column shows the frequency achieved. The third through the seventh columns show how LUTs are used in the implementation, while the eighth column shows the number of packed ALMs used. The ninth column shows the number of registers used. It is clear from this table that relatively few resources are used.

Another way to compute complexity is to count the number of comparators. There are $n-1$ stages; the first has n comparators, the second has $n-1$, $\cdots$, and the last has 2 comparators, for a total of $n+n-1+\cdots+2+1-1 = \frac{(n+1)n}{2}-1 = \frac{n^2+n-2}{2} = \mathrm{O}(n^2)$ comparators. Since there are $n-1$ stages, the delay is $\mathrm{O}(n)$.

III. Random Permutation Generator

A. Introduction

The generation of *random* permutations is an important topic, e.g. in Monte Carlo simulations. They are also useful in the assessment of sorting algorithms. For example, compared to other sorting algorithms, the Insertion Sort is known to be efficient when the list is "almost sorted", and inefficient when the list is "almost unsorted". Oommen and Ng [14] show how to generate random permutations with various distribution functions. For example, one can build a distribution that produces, with greater frequency, almost sorted permutations. In this paper, we consider only uniformly distributed random permutations. We consider two ways to generate random

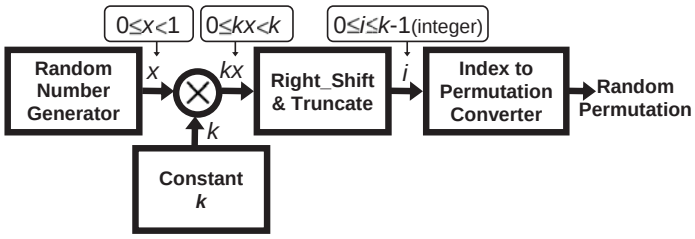

Fig. 2. Block Diagram of a Random Permutation Generator.

permutations. In the first, the index is a random integer. Fig. 2 shows how a random number generator, which generates a (pseudo) random number between $00\ldots0$ and $11\ldots1$ can be used to construct a (pseudo) random integer. The output of the random number generator can be viewed as a number x, such that $0 \le x < 1$ (Let there be a *virtual* binary point to the left of the most significant bit; all generated numbers are less than 1). Multiplying this by integer k yields a value y, such that $0 \le y < k$. We choose k appropriately. For example, in the circuit that converts an index from 0 to 23 to an $n = 4$ permutation, we set k to 24. Thus, the input kx to the Right_Shift & Truncate block of Fig. 2 has the property $0 \le kx < k$. With $k = 24$, the truncation of kx means that i, the output of the Right_Shift & Truncate block, is an integer with the property $0 \le i \le 23$. Note that the multiplier in Fig. 2 multiplies a *constant* value times a random variable. This can be implemented as a shift-and-add multiplier with little delay. The shift-and-add multiplier is much faster than the multiplier typically found in an FPGA.

Note that the degree to which the random integers are uniformly distributed depends on m, the number of bits used in the random number generator. For example, in the $n = 4$ permutation generator, $m = 5$ is too small, because only 31 random numbers are distributed across 24 random integers. This assumes that the random number generator is an LFSR, in which case 31 random numbers (x in Fig. 2) would be generated (the LFSR random number generator generates all 32 5-bit numbers except 00000). According to the pigeonhole principle [12], at least one random integer will be generated by at least one more random number than some other random integer. In the circuit shown in Fig. 2, seven of the random integers are generated from two random numbers, while 17 are generated from one random number. As a result, seven random integers are generated with a probability that is twice that of 17 other integers. Choosing a larger m reduces the difference in probabilities of two random integers. For example, for $m = 32$, the difference reduces to 5.6×10^{-7}%.

A disadvantage of the random integer generator shown in Fig. 2 is the large size of the index. For example, for permutations on $n = 64$ elements, 296 bits are required to represent the index. Alternatively, we consider another way to implement a random permutation generator.

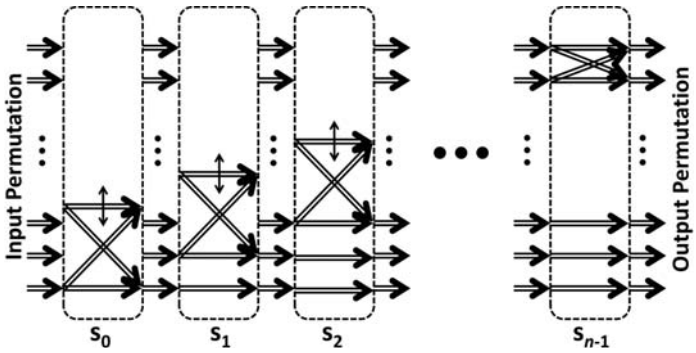

Fig. 3. Block Diagram of a Knuth Shuffle Random Permutation Generator.

In the second approach, the *Knuth shuffle* circuit is used. It starts with any specified permutation, π_0, for example the identity permutation. It proceeds in $n-1$ steps. In the first step, the first (left) element of π_0 is interchanged with *any* other element including itself. These n choices occur randomly with uniform probability. In this way, the left element of π_1 is chosen as any element (and with equal probability). Next, the second element from the left is interchanged with any element to its right including itself. In this way, the second element of π_2 is chosen as any element except the left element. In a similar manner, all other elements are chosen until finally, the right two elements are either interchanged or left unchanged with equal probability.

TABLE III
FREQUENCY AND RESOURCES USED TO REALIZE THE FACTORIAL NUMBER SYSTEM IMPLEMENTATION OF AN INDEX TO PERMUTATION CONVERTER ON THE ALTERA STRATIX IV EP4SE530F43C3NES FPGA.

n	Freq. (MHz)	# of LUTs of Various Inputs 7-	6-	5-	4-	3-	Est. # of Packed ALMs	Total # of Registers
2	392.8	0	0	0	0	-	1(0%)	1(0%)
3	406.3	0	0	0	2	5	6(0%)	11(0%)
4	406.3	0	1	4	10	11	19(0%)	36(0%)
5	406.3	0	0	11	25	23	41(0%)	69(0%)
6	259.0	0	1	34	48	36	79(0%)	117(0%)
7	267.6	0	7	43	84	48	123(0%)	174(0%)
8	229.8	0	12	80	113	86	186(0%)	242(0%)
9	197.7	0	12	110	173	100	260(0%)	332(0%)
10	180.5	1	25	176	211	121	355(0%)	436(0%)
16	153.0	10	36	253	563	3,652	2,331(0%)	1,029(0%)
32	106.1	28	506	1,073	3,391	34,804	20,953(9%)	5,349(1%)

B. Circuit Implementation

The circuit that implements the Knuth shuffle is shown in Fig. 3. It is implemented as a cascade of n stages, where each stage performs the interchange discussed above. At each stage, an element is interchanged with itself or any of the elements to its right. For example, the leftmost stage, s_0 chooses from n elements the first or left element. The choices are made from among the n elements of the applied permutation, labeled "Input Permutation". Then, the second stage chooses from among the remaining elements, etc. Each choice is random. In stage s_0, there are n choices, in stage s_1, there are $n-1$ choices, etc. In Fig. 3 there is a random integer generator for each choice.

C. Results

Fig. 4 shows the output of the Knuth shuffle circuit in producing $2^{20} = 1,048,576$ random 4-element permutations. This circuit has three stages with one random number generator per stage. The input permutation is the identity permutation, while the output represents a random permutation. We choose to represent each element as a 2-bit binary number as follows: $3 = 11_2$, $2 = 10_2$, $1 = 01_2$, and $0 = 00_2$. For example, the bottom bar represents 43,399 occurrences of the permutation 0123, and the second bar up represents 43,897 occurrences of the permutation 0132. It is clear that the distribution is uniform. The vertical axis represents the value of an 8-bit binary number, 0 through 255. Of the 256 possible output values, only 24 represent permutations. For example, 00 01 10 11 and 00 01 11 10 represent 0123 and 0132, respectively. Thus, this bar chart has 24 bars, one for each 8-bit binary number that corresponds to a permutation. The permutations are shown along the right side. The vertical axis on the left hand side represents the binary value of the permutations. For example, permutations 0123 and 0132 correspond to $00\ 01\ 10\ 11 = 27_{10}$ and $00\ 01\ 11\ 10 = 30_{10}$, respectively.

As another test of the random permutation generator, we wrote Verilog code to compute the number of derangements on the SRC-6 reconfigurable computer. A *derangement* is a permutation in which *all* elements have moved from their original positions; i.e. have no *fixed points*. For example, permutation 0123 has four fixed points (0, 1, 2, and 3), 0312 has one fixed point (0), and 3210 has no fixed points (and is, therefore, a derangement). It is known that the number of derangements d_n on n elements is given as $d_n = \|n!/e\|$, where $\|a\|$ is the closest integer to a. In the generation of 1,048,576 random 4-element permutations used in Fig. 4, 392,712 of them were derangements. Therefore, we can approximate e as $e \approx 1048576/393661 = 2.665$. Repeating this for $n = 8$ and $n = 16$, yields $e \approx 16777216/6174763 = 2.7171$ and $e \approx 16777216/6171117 = 2.7187$, respectively.

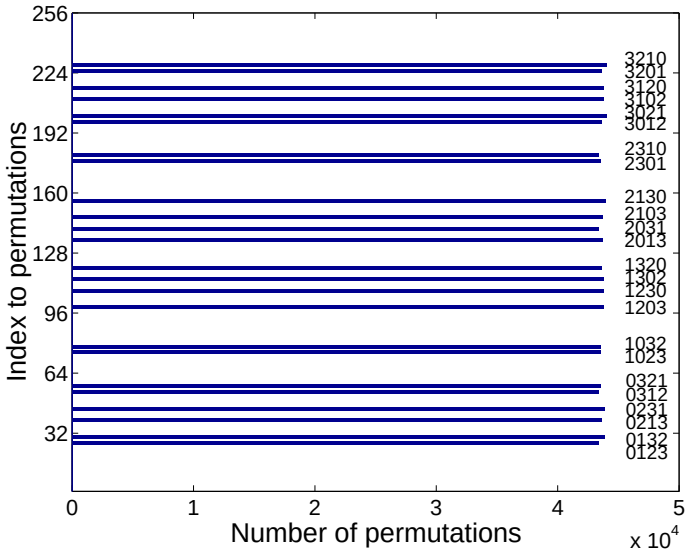

Fig. 4. Distribution of Permutations Produced by the Knuth Shuffle Random Permutation Generator

When the random permutation generator and the derangement generator were programmed to run on the SRC-6 reconfigurable computer, in both cases, only 1% of the 4-input LUTs were used, only 2% of the flip-flops were used, and only 3% of the slices were occupied. Table IV shows how much of the various resources are used in the Knuth shuffle implementation versus various n when programmed for the the Altera Stratix

TABLE IV
FREQUENCY AND RESOURCES USED TO REALIZE THE KNUTH SHUFFLE RANDOM PERMUTATION GENERATOR ON THE ALTERA STRATIX IV EP4SE530F43C3NES FPGA.

n	Freq. (MHz)	# of LUTs of Various Inputs 7-	6-	5-	4-	3-	Est. # of Packed ALMs	Total # of Registers
2	1,520.2	0	0	1	1	-	6(0%)	10(0%)
3	528.4	0	0	2	1	17	19(0%)	28(0%)
4	810.7	0	0	3	6	24	32(0%)	56(0%)
5	500.8	0	1	6	10	43	54(0%)	87(0%)
6	496.6	0	5	8	25	43	77(0%)	125(0%)
7	424.0	2	9	18	34	51	110(0%)	167(0%)
8	347.7	1	21	32	42	63	139(0%)	199(0%)
9	388.3	2	12	55	60	62	181(0%)	284(0%)
10	329.9	1	23	94	56	75	219(0%)	328(0%)
16	173.5	10	70	507	186	373	951(0%)	1,145(0%)
32	153.4	84	863	2,384	695	2,499	5,499(2%)	5,434(1%)

IV EP4SE530F43C3NES FPGA. In both cases, the number of elements includes the random number generator needed for each stage. Indeed, a 32-bit random integer generator similar to that shown in Fig. 2 was included in each stage.

If we measure the complexity of the Knuth Shuffle circuit by the number of crossover circuits, the first stage has $n-1$, the second has $n-2$, $\cdots$, and the last has 1, for a total of $n-1+n-2+\cdots+1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \mathrm{O}(n^2)$. This is identical to the complexity of the index to permutation generator. Similarly, the delay is also $\mathrm{O}(n)$.

IV. CONCLUDING REMARKS

We show two approaches to the generation of permutations

1) a factorial number system index to permutation converter
2) a Knuth shuffle random permutation generator

Both approaches can be implemented as a combinational logic circuit or as a pipeline producing a permutation at each clock. We have shown that the index to permutation converter implemented on the SRC-6 reconfigurable computer operates 1,820 times faster than a software implementation on an Intel Xeon microprocessor. We have also shown that our designs require modest resources on an Altera Stratix IV EP4SE530F43C3NES FPGA. We have shown an efficient circuit to produce random permutations, which is useful in Monte Carlo simulations. For both circuits, the complexity is $\mathrm{O}(n^2)$, and the delay is $\mathrm{O}(n)$. We believe this is the first time such circuits have appeared in the literature.

The alert reader will note that the factorial number system circuit and the Knuth shuffle circuit can also serve as a sorting network.

V. ACKNOWLEDGMENTS

This research is supported by a Grant-in-Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS) and a Knowledge Cluster Initiative (the second stage) of MEXT (Ministry of Education, Culture, Sports, Science and Technology). We appreciate the comments of four referees.

REFERENCES

[1] Z. Arnavut and S. Narumalini, "Application of permutations to lossless compression of multispectral thematic mapper images", *Opt. Eng.*, Vol. 35, No. 12, 3442–3448, December 1996.

[2] J. Barbay and G. Navarro, "Compressed representations of permutations and applications," *26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009)*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, pp. 111-122, 2009.

[3] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Computers*, Vol. C-35, No. 8, pp. 677-691, Aug. 1986.

[4] J. T. Butler and T. Sasao, "Index to constant weight codeword converter," *Proc. of the 7th Inter. Symp. on Applied Reconfigurable Computing, Proceedings Lecture Notes in Computer Science (LNCS 6576) Springer-Verlag Berlin Heidelberg, 2011*, A. Koch et al (Eds.), Belfast, N. Ireland, March 23-25, 2011, pp. 193-204,

[5] D. Debnath and T. Sasao, "Fast Boolean matching under permutation by efficient computation of canonical form," *IEICE Trans. Fundamentals*, No. 12, December 2004, pp. 3134-3140.

[6] S. Dolev, L. Lahiani, and Y. Haviv, "Unique permutation hashing," *Stabilization, Safety, and Security of Distributed Systems, Lecture Notes in Computer Science*, Volume 5873. ISBN 978-3-642-05117-3. Springer-Verlag Berlin Heidelberg, 2009, p. 777, https://www.math.bgu.ac.il/ frankel/TechRep/09-01/09-03.pdf .

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

[8] D. E. Knuth, "Volume 2 Seminumerical Algorithms," *The Art of Computer Programming*, (3rd ed.), Addison-Wesley, p. 192, ISBN 0-201-89684-2.

[9] D. E. Knuth, *The Art of Computer Programming*, "Generating all combinations and partitions," Vol. 4, **Fascicle 3**, Addison-Wesley, pp. 5-6, ISBN 0-321-58050-8.

[10] D. E. Knuth, *The Art of Computer Programming*, "Generating all combinations and partitions," Vol. 4, **Fascicle 3**, Addison-Wesley, pp. 5-6, ISBN 0-321-58050-8.

[11] C.-A. Laisant, "Sur la numération factorielle, application aux permutations," (http://www.numdam.org/item?id=BSMF_1988__16__176_0) Bulletin de la Société Mathématique de France Vol. 16, pp. 176-183.

[12] C. L. Liu, "Elements of Discrete Mathematics," *McGraw-Hill*, New York, 1977, pp. 72-76.

[13] N. D. Memon, K. Sayood, and S. S. Magliveras, "Lossless compression of multispectral image data," *IEEE Trans. Geosci. Remote Sensing*, Vol. 32, No. 2, pp. 282–289 1994.

[14] B. J. Oommen and D. T. H. Ng, "On generating random permutations with arbitrary distributions," *The Computer Journal*, Vol 33, No. 4, 1990, pp. 368-374.

[15] A. Parsons, "The symmetric group in data permutation, with applications to high-bandwidth pipelined FFT architectures," *IEEE Signal Processing Letters*, Vol. 16, No. 6, June 2009.

[16] T. Sasao, *Memory Based Logic Synthesis*, 1st Edition, Springer, 2011, ISBN 978-1-4419-8103-5.

[17] B. Schneier, *Applied cryptography*, 2nd Edition, John Wiley & Sons, Inc. 1996.

[18] J. L. Shi and R. B. Lee, "Bit permutation instructions for accelerating software cryptography," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2000)*, pp. 138-148, July 2000.