INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2025, VOL. 71, NO. 4, PP. 1–8
Manuscript received July 11, 2025; revised October 2025. doi: 10.24425/ijet.2025.155467

A new algorithm for generating integer partitions and its parallel implementations on CPU and FPGA

Marek Nałęcz, and Gustaw Mazurek

***Abstract*—In this paper, the long-known bit representation of integer partitions was used in a novel way to develop an algorithm for generating the next partition of an integer n. This algorithm is both loopless and conditionless (and therefore has strictly constant execution time). These features lead to an efficient and highly scalable parallel implementation on a modern multi-core CPU processor or a modern FPGA chip. In the CPU case, just 30 processor clock cycles are required to generate the next partition when $n < 128$. In the latter case, a single one of the parallel instances uses only 1,595 look-up tables and 962 registers for $n < 128$. It can produce the next partition in just 6 clock cycles. Even cost-effective FPGAs can hold a few dozen such instances.**

***Keywords*—parallel combinatorial algorithms; integer partitions; field-programmable gate arrays; bitwise operations**

I. Introduction

THE unprecedented computational power of modern field-programmable gate array (FPGA) devices makes them well-suited for solving combinatorial problems, which typically exhibit enormous computational complexity. One such problem is the generation of all integer partitions of a given integer n. Integer partitions constitute a significant and interesting topic in combinatorics since the mid-eighteenth century [1, p. 395]. In this paper, an effective parallel version of the partition generation algorithm that fits well into the capabilities of modern hardware is developed and presented. The proposed algorithm uses only simple bitwise operations that can easily be implemented on both a central processing unit (CPU) and an FPGA. Hardware implementations of combinatorial algorithms on FPGAs, in the spirit of [2] or [3], are an interesting alternative to CPU or graphics processing unit (GPU) implementations.

The remainder of this paper is organized as follows. Section II provides a necessary combinatorial background: after introducing the bit representation of partitions used in the paper, it presents basic operations on partitions, including the abstract version of the partition generation algorithm and its adaptation to the bit representation — the way of parallelizing the algorithm is discussed, too. Then Section III provides details regarding embedding this algorithm into the CPU and FPGA. The sequential and parallel versions are presented on both platforms. The results, including timing, resource utilization, scalability, and energy efficiency, are briefly discussed in Section IV, along with a list of possible practical applications.

Authors are with the Faculty of Electronics and Information Technology, Warsaw University of Technology, Poland (e-mail: Marek.Nalecz@pw.edu.pl; Gustaw.Mazurek@pw.edu.pl).

II. Theory

A. Integer partitions

A partition λ of n, denoted as $\lambda = (a_1, a_2, \ldots, a_m) \vdash n$, is a descending (non-increasing) representation of n as a sum of $m \geq 0$ positive summands: $a_1 + a_2 + \ldots + a_m = n$, $a_1 \geq a_2 \geq \ldots \geq a_m > 0$ for $i = 1, \ldots, m$. In line with [4], [5], the term 'summand' is used in the present paper as less confusing, although the word 'part' is more often encountered in the literature. The number of all the (unrestricted) partitions of a given n is called the partition function and is denoted $p(n)$ following the tables in [6]. The function $p(n, m)$ is defined as the number of (restricted) partitions of n into at most m summands or (equivalently) as the number of partitions of n into summands $\leq m$ [6, p. IX].

The natural order of partitions is reverse lexicographic order [7]. In this order, the first distinct summand is greater in the preceding of two partitions [8, pp. 320–321]. The ordering of partitions allows for a bijection between the set of integers $\{r : 0 \leq r \leq p(n) - 1\}$ and the set $\{\lambda : \lambda \vdash n\}$ of all partitions of a given n using a ranking function $r = \mathrm{V}_n(\lambda)$ (defined as the number of partitions of n preceding a given one) and its inverse $\lambda = \Lambda_n(r)$, called the unranking function [9].

In array representations, a partition is represented as a vector (array) of elements (memory cells) of the same type. These representations include standard representation [10, (3.17)], [11, p. 652], [8, p. 321], part-count representation [12], [13, (1.3)], [1, 7.2.1.4-(8)], and multiplicity representation [10, p. 76], [13, (2.3)], [14, 5.3.2]. Standard representation requires $n(\lfloor \log_2 n \rfloor + 1)$ bits, and part-count representation twice as many. Multiplicity representation is slightly more compact, but it is not suitable for every application [8, p. 321]. As all array representations require significant memory, they are not well suited for massive parallelization on contemporary hardware. Therefore, the authors tend to use a more economical bit representation.

The bit representation used throughout the present paper is created as a left-to-right concatenation of the sequences of bits corresponding to the consecutive summands, from the smallest summand to the largest one. Each summand a_i is represented

as a_i bits: $(a_i - 1)$ 0s followed by a single 1. An n-bit stream obtained in this way is then left-aligned within a w-bit machine word (where $w > n$) and right-padded with $(w - n)$ 0s, constituting the binary number $\mathcal{B}(\lambda)$, which represents the partition $\lambda = (a_1, a_2, \ldots, a_m) \vdash n$ as:

$$\begin{aligned}\mathcal{B}(\lambda) &= \left(0^{a_m-1}10^{a_{m-1}-1}1\ldots10^{a_2-1}10^{a_1-1}10^{w-n}\right)_2 \\ &= \sum_{i=1}^{m} 2^{w-\sum_{j=i}^{m} a_j}. \end{aligned} \tag{1}$$

Left alignment was used because operations on the most significant bits (MSBs) are better supported by modern hardware than operations on the least significant bits.

The bit representation is very compact (it typically requires an order of magnitude fewer bits than the standard representation). Perhaps this was why it was proposed by Comét at the beginning of the digital computer era in 1953 [15, p. 22], [16, p. 144] (actually, with the reversed order of summands). The representation similar to (1), but without the rightmost 1 and the following 0s, was proposed in [10, p. 75] and [17, p. 118] as 'difference representation'.

B. Operations on partitions

The basic operations on partitions of an integer n include:

- Determining the first and last partition in reverse lexicographic order.
- Comparing any two partitions.
- Operations requiring the creation of a representation of a partition based on finding its summands one by one, such as computing the unranking function.
- Calculating the next partition in lexicographic or reverse lexicographic order.

The latter operation is crucial for efficiently generating all the partitions of a fixed integer. Using the unranking function for this purpose, although possible, is orders of magnitude slower.

The algorithm for generating the next partition in reverse lexicographic order has been described in its abstract (i.e., representation-independent) form in many sources (e.g., [10, p. 150], [17, p. 120], [18, (14.2.1)–(14.2.2), p. 230], [19, Alg. 7, p. 13], and [1, p. 391]). It can be written as Algorithm A, which is the basis of all algorithms known to the authors that generate unrestricted partitions in reverse lexicographic order. Note that starting the representation (1) from the smallest summands and aligning it to the left facilitates step 1 of this algorithm. The natural implementation of step 2 requires a loop, and steps 1 and 3 — conditional statements.

Algorithm A Generate next integer partition — abstract vers.

1: Find the smallest summand $k + 1 > 1$. If none, stop.
2: Use $k + 1$ and all of the following 1s to create as many summands of size k as possible.
3: The remainder $< k$, if any, becomes the last summand.

Let us also note that all algorithms for the standard representation contain loops, whereas all algorithms for the part-count or multiplicity representation are loopless. However, all algorithms for the multiplicity representation working in reverse lexicographic order require computing the integer division or remainder, which are not cheap operations. Finally, all partition generation algorithms known to the authors contain conditional statements; thus, they do not have a strictly constant execution time, unless appropriately padded.

The desirable features of the algorithm to easily be parallelized include: using the weakest possible parallel computation model, having the lowest memory requirements, and being as simple as possible [20].

C. Operations on partitions in bit representation

The basic operations on the partitions of an integer n are elementary in the bit representation. The first partition in the reverse lexicographic order is represented by:

$$\mathcal{B}(\Lambda_n(0)) = (0^{n-1}10^{w-n})_2 = 2^{w-n}, \tag{2}$$

whereas the last is by:

$$\mathcal{B}(\Lambda_n(p(n) - 1)) = (1^n0^{w-n})_2 = 2^w - 2^{w-n}. \tag{3}$$

The equally simple task is to compare two partitions according to the reverse lexicographic order by employing a trick described in [1, Ex. 7.1.3-45, p. 590].

The first step of Algorithm A is equivalent to finding the number t of leading 1s of $\mathcal{B}(\lambda)$. The smallest summand > 1 (if any) appears just right to this run of 1s. In this way, line 1 of Algorithm B is obtained. It applies the function $\ell(x)$, which counts leading 0s, to the negated representation (the negation is denoted by the overline). Line 2 of Algorithm B computes k, which equals the smallest summand > 1 less one, by counting 0s following the run of leading 1s. The appropriate t-bit shift ($\ll$) is used to discard these 1s, followed by invoking the $\ell(x)$ function.

Algorithm B Generate next integer partition — bitwise vers.

Require: $p = \mathcal{B}(\Lambda_n(r)), 0 \le r < p(n) - 1$ {r = rank}
Require: $\lfloor \log_2 p \rfloor < n_{\max} = w - 1$ {w = word length}
1: $t \leftarrow \ell(\overline{p})$ {count leading 1s in negated representation}
2: $k \leftarrow \ell(p \ll t)$ {count run of 0s just right to leading 1s}
3: $p \leftarrow p \oplus (\mu_{n_{\max},k} \ll (n_{\max} - t))$ {$\mu_{n_{\max},k}$ precomputed}
Ensure: $p = \mathcal{B}(\Lambda_n(r + 1))$ {next partition}

The second step of Algorithm A uses the $k+1$ summand and all the preceding t 1s. We must construct as many summands equal to k as possible. First, the summand $k + 1$ has to be shortened by one, by changing its most significant 0 to 1. This 1 will become the first bit of the new summand. The following $(k-1)$ bits (towards MSB) are included in this new summand. Up to now, these bits were 1s. For the successor partition, we must change all of them to 0s. The next bit to the left will start another summand equal to k; thus, this bit will remain 1 in the new representation. This process is repeated right-to-left for as long as possible (i.e., until we run out of the initial leading run of 1s). As a result, the bits at positions $w - t - 1 + k$, $w - t - 1 + 2k$, $w - t - 1 + 3k$, and so on will remain unchanged, and we must negate all remaining bits in

the range $w-t-1,\ldots,w-1$. If t is not divisible by k, a summand $<k$ will naturally arise on the left, implementing step 3 of Algorithm A.

Changing selected bits from 0 to 1 and in the opposite way simultaneously is most conveniently done by the XOR operation $\oplus$ with an appropriate magic mask $\mu_{t,k}=2^{w-t-1}\mu_k$ (where μ_k is given below by (5)), as in line 3 of Algorithm B:

$$\mathcal{B}(\Lambda_n(r+1))=\mathcal{B}(\Lambda_n(r))\oplus\mu_{t,k}. \tag{4}$$

In conformance with the description in the previous paragraph, the generic mask μ_k should be applied from bit position $w-t-1$ to the left. The mask μ_k should contain a 1 and then a repeating pattern consisting of $(k-1)$ 1s and a 0. Such a mask is an infinite and periodic 2-adic fraction (cf. [1, p. 141]):

$$\mu_k=(\ldots\underline{01^{k-1}}\,\underline{01^{k-1}}\,1)_2=-\frac{2^k}{2^k-1}, \tag{5}$$

where underscores denote the repeating pattern of bits. The masks (5) can easily be tabulated after truncating them to $w=n_{\max}+1$ bits, and denoting as $\mu_{n_{\max},k}$ for $k=1,\ldots,n_{\max}$.

Algorithm B is the most compact algorithm for generating the next partition known to the authors. It is loopless and does not contain conditions. The table of precomputed masks $\mu_{n_{\max},k}$ is sufficiently small (2 kB for $n_{\max}=127$) to fit within a CPU L1 cache or to be implemented in an FPGA using only look-up tables (LUTs). Function $\ell(x)$ used twice in Algorithm B (lines 1 and 2) counts the leading 0s of a binary number x. It is equivalent to the fast LZCNT operation from the BMI1 instruction set of the x86-64 processor architecture. Other operations required to compute the next partition include only subtraction, multi-bit-shifting, and exclusive OR. The simplicity of these operations has been emphasized by Comét [15, pp. 22–23], [16, p. 144].

D. Parallelization

Algorithm B, which determines how to generate the single next partition, can easily be applied to generate all partitions of a fixed integer n. It is sufficient to initialize the variable p, representing the partition, with the representation of the first partition (2) and repeat the algorithm until the last partition (3) is reached. Note that in this way, we generate only $p(n)-1$ partitions using Algorithm B (skipping the first one, which is the algorithm input).

This procedure for generating all partitions of integer n can easily be parallelized as in [11, p. 644] or [20, p. 128] into M completely independent processes. Let us assume $M<p(n)$ and define $M+1$ ranks r_k: $0=r_1<r_2<\ldots<r_M<r_{M+1}=p(n)-1$ that divide the range of all partition ranks $0,\ldots,p(n)-1$ into M approximately equal chunks:

$$r_k=(k-1)\underbrace{\left\lfloor\frac{p(n)-1}{M}\right\rfloor}_{\text{quotient } q}+\min(k-1,\underbrace{(p(n)-1)\mod M}_{\text{remainder } d}), \tag{6}$$

where $k=1,\ldots,M+1$. The ranks r_k and r_{k+1} define the first and the last partition belonging to the k-th chunk as $\Lambda_n(r_k)$ and $\Lambda_n(r_{k+1})$, respectively. Note that a given chunk's final partition is also the next chunk's initial partition.

The initial and final values of the variable p of Algorithm B are determined by calculating the unranking function Λ_n, and all intermediate values are generated using Algorithm B. We noted above that calculating the unranking function can be orders of magnitude slower than calculating the next partition. In practice, however, this is irrelevant because the number of chunks $M\ll p(n)$, so the expensive unranking operation affects only a tiny fraction of the data and has no noticeable effect on the execution time of the parallel program. Furthermore, the present study used the fastest unranking algorithm known to the authors, given in [21] and based on the efficient tabulation of the $p(n,m)$ function.

Assuming the ranks (6) as the boundaries of the individual chunks generated concurrently, the first d chunks contain $q+1$ partitions each, while the remaining $M-d$ chunks contain q partitions each. As a result, the sizes of the individual chunks differ by at most one, allowing the work to be distributed as evenly as possible between the chunks. Then, assume that the number of chunks M is either a multiple of the number of available processing elements (PEs), or is much greater than the number of PEs. In that case, we obtain an optimal load balance of the individual PEs.

III. Experiment

A. CPU implementation

Algorithm B was coded in C23 language and then tested on an 11-th Gen Intel Core i7-11800H CPU processor running the Windows 11 operating system. The GCC 14.2.0 compiler was used for the tests presented below; almost identical results were obtained using the Intel oneAPI 2025.0.0 compiler. The CPU core clock frequency was fixed at 4.05 GHz to make the timing results more reliable. The observed variability in execution time for different runs of the same program did not exceed a fraction of a percent.

First, the sequential version of Algorithm B was tested to investigate the resource requirements as a function of the partitioned integer n and the length of the partition representation $w=n_{\max}+1$. Since the CPU processor has a fixed architecture and dedicates a single core to the execution of the single-threaded algorithm, the only relevant resource is the program execution time, measured in processor clock cycles.

The representation lengths w equal to 32, 64, 128, and 256 bits were analyzed. For each of them (except $w=256$), the tests included the integer $n=n_{\max}=w-1$ and several smaller integers n for which the number of partitions $p(n)$ exceeded for the first time the consecutive powers of 10 (in the range from 10^2 to 10^{12}). The largest value tested in the sequential mode was $n=185$ —- the program execution time exceeded 5.5 hours in this case. To ensure good time measurement accuracy for smaller values of n, the generation of all partitions was repeated multiple times, so that the actual program execution time did not drop below 1 s. The results are presented in Fig. 1, showing the net program execution time

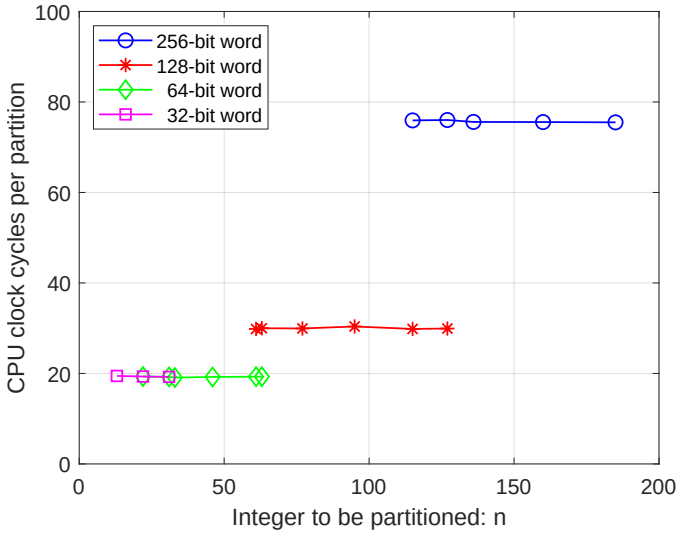

Fig. 1. CPU user time per partition as a function of n for different word lengths w.

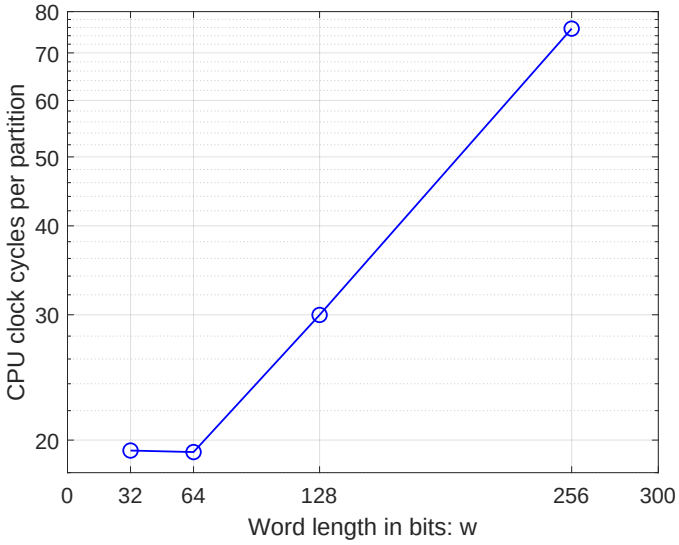

Fig. 2. CPU user time per partition as a function word length w.

(defined as the user time given by the system `time` command) divided by the number of generated partitions.

Fig. 1 shows that the generation time of a single partition does not depend on n for a fixed $n_{\max} = w - 1$. Minor deviations visible in the plot result from the inaccuracy of the measurement of the execution time. Although the tested program was the only user application running on the computer during the measurements, dozens of unavoidable operating system processes were running in the background. The second observation concerns the identical generation times of a single partition for $w = 32$ and $w = 64$. This results from the fact that although the CPU processor actually has a dual 32/64-bit architecture, the test application was compiled for 64-bit mode even with $w = 32$ (as this variant was faster).

Since the number of clock cycles per partition depends only on the word length w, this relationship is presented in Fig. 2. As mentioned above, the partition generation times for $w = 32$ and $w = 64$ are identical. Then, the function grows slightly faster than linearly —- a relationship close to linear is obtained using a logarithmic scale on the vertical axis. The number of clock cycles c can be approximated by the formula $c \approx 12\,\mathrm{e}^{w/140}$ for $w \in \{64, 128, 256\}$. However, this relationship should not be given much importance — it does not result from the essence of the problem or the algorithm but only from the properties of the multiple precision arithmetic library used by the compiler, which extends the capabilities of the natively 64-bit CPU.

Further studies concerned the parallel version of Algorithm B, obtained in the manner described in Subsection II-D. The problem was decomposed into M completely independent chunks. Since the hardware platform has eight physical cores, each capable of running two concurrent threads, the study was carried out for M changing from 1 to 16, then with a step of 16 up to 128. Each time, the speedup obtained thanks to parallelization was determined and estimated as the ratio of

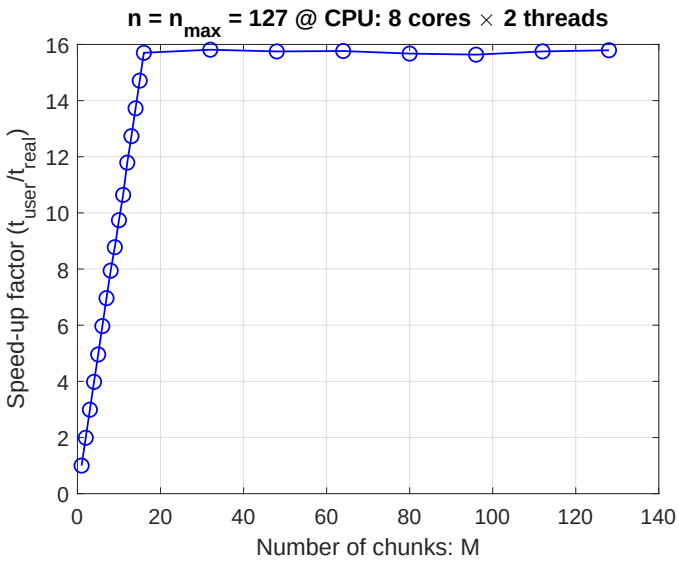

Fig. 3. CPU user speed-up factor as a function of the number M of concurrently scheduled chunks.

user time to real execution time (both values were returned by the system `time` command). The results are shown in Fig. 3.

As long as the number of chunks did not exceed the number of physical processor cores ($M \le 8$), the obtained speedup was essentially equal to M (it was within 99.3…99.9% of M). The speedup was slightly lower for M values ranging from 9 to 16 because of the concurrent execution of two threads on one physical core and the overhead of handling system applications in the background. Nevertheless, it remained at 96.7…98.3% of M, which should be considered an excellent result. For larger values of M, which are multiples of the number of virtual cores (16), the speedup settled at approximately 15.7. The results indicate the excellent scalability of the parallel version of Algorithm B with respect to the number

of processor cores. Thanks to parallelization, even on a not very powerful CPU used in the experiments, it turned out to be possible to generate all (approximately 10^{13}) partitions of integer $n = 211$ in a time slightly exceeding 5 hours. The generation of all partitions of such a large integer n on a PC platform has not been reported in the literature so far, to the best of the authors' knowledge.

During these timing experiments of the parallel version of the program, the effective power consumed by the notebook computer used in the tests (with the display turned off) was measured by the VOLTCRAFT Energy Logger 4000 as 62 W. At the same time, the HWINFO version 8.28 software showed the total CPU power (averaged over half an hour of the computations) as 40.3 W, most of which was consumed by the Intel Architecture cores (36.9 W). Consequently, only about 60% of the power consumed by the computer was devoted to the computation of the partitions.

Obtaining such promising parallelization results for Algorithm B on the CPU raises questions about parallelization results on the FPGA platform. It should scale better with respect to the word length w (due to the lack of need to use a software multiple precision library) and with the number of chunks M executed in parallel (due to the lack of overhead for handling operating system tasks), and also to provide much better energy efficiency. This approach is investigated in the following Subsection III-B.

B. FPGA implementation

The authors are unaware of previous studies on the generation of the next partition using an FPGA. Despite its title, Butler's article [2] is devoted to partition unranking, not generation; the work [22] regards set, not integer, partitions. For this reason, an attempt to implement Algorithm B in FPGA technology would be justified. An FPGA device made in 28 nm technology (XC7A100T by AMD) was chosen, available on a Digilent Nexys A7-100T development board [23]. It offers a rich amount of programmable logic resources (15,850 programmable logic slices, each with four 6-input LUTs and eight flip-flops), which encourages the prototyping of complex algorithms. Moreover, the selected board is offered as a cost-optimized evaluation platform for academic purposes.

Algorithm B was coded in VHDL for $w = 128$ as a pipeline shown in Fig. 4 (pipeline registers and clock distribution are not shown for clarity). The computation process is controlled by a simple state machine that receives input signals, counts the clock cycles required to process a complete iteration, and drives status output flags. The computation is initialized with a starting 128-bit word $p = \mathcal{B}(\Lambda_n(r))$ representing the partition (P_IN), accompanied by ND $= 1$ and LB_EN $= 0$. After completing the iteration, the VALID flag signals the appearance of a next partition representation $p = \mathcal{B}(\Lambda_n(r+1))$ at P_OUT. It is then fed back to the input (and used externally if needed), when LB_EN $= 1$, to start a new iteration. If a required final partition (given in CMP_IN) appears at P_OUT, the CMP_OUT flag is additionally activated, and the number of executed iterations is presented at the CNT output.

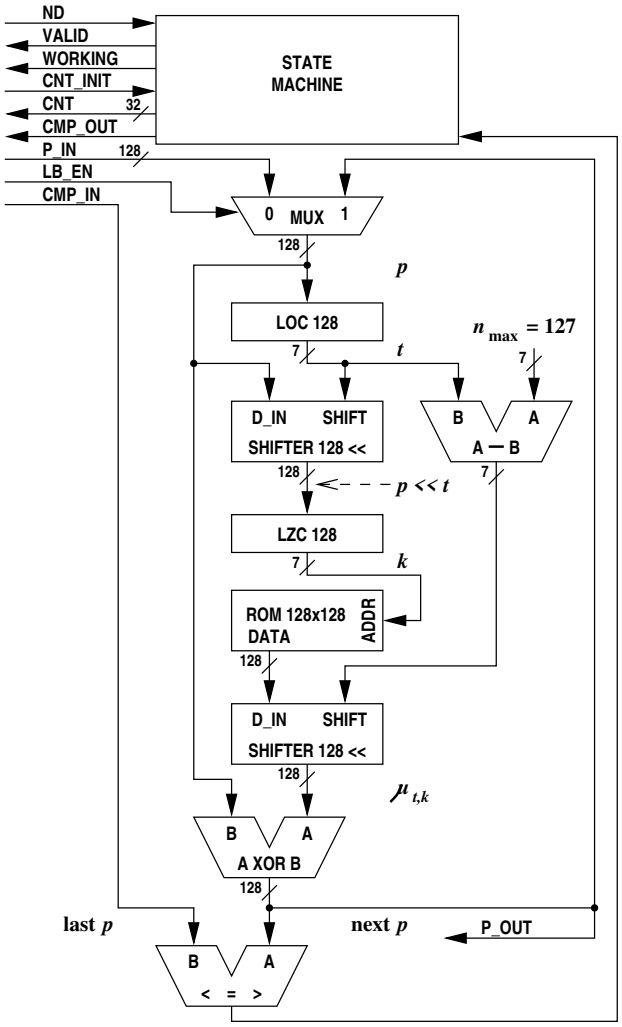

Fig. 4. Implementation of Algorithm B as Partition Process module.

The processing pipeline consists of several lower-level modules. The LZC 128 module presented in Fig. 5 is responsible for counting leading zeros $\ell(\mathsf{X})$ in a 128-bit binary word X, and it consists of two (smaller by half) LZC 64 modules. Each LZC 64 module is realized again in a similar manner from two yet smaller modules LZC 32 (described in detail in [24]), and composed hierarchically from even smaller (16- and 8-bit) blocks. The combination logic that gives the zero-count result Z is implemented as an extension of the logic equations given in [24, (20)–(24)].

The LOC 128 module is a modified version of LZC 128 with negated all input bits to obtain the functionality of counting leading ones in a 128-bit word. Both bit shifters (SHIFTER 128) are well-known barrel shifters with seven layers of simple 2:1 multiplexers controlled by individual bits of the SHIFT input. The ROM block contains 128 words, 128 bits each, equal to the precomputed masks $\mu_{n_{\max},k}$ for $k = 1, \ldots, n_{\max}$, with an extra zero fill-in at address 0.

To replicate numerous instances of Algorithm B and integrate them with higher-level functionality, each instance was wrapped in the Partition Engine module, depicted in Fig. 6. This module incorporates an additional state machine,

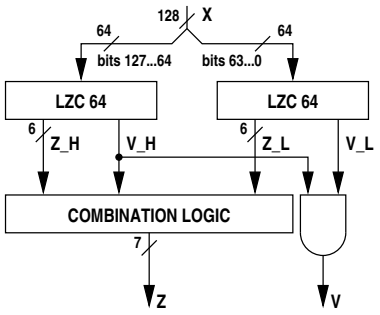

Fig. 5. LZC 128 module, implementing $\ell(\mathsf{X})$, which counts leading 0s.

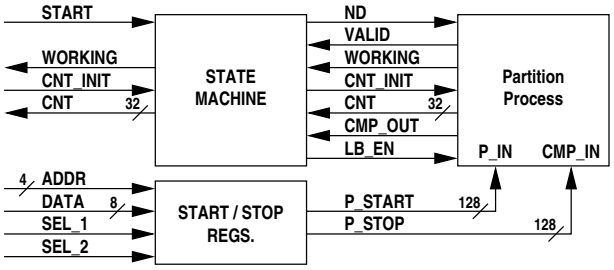

Fig. 6. Implementation of the wrapper module Partition Engine.

launched by the START input, that controls the algorithm iterations (by driving ND and LB_EN signals) until achieving the last partition, which is signaled by CMP_OUT. It also contains two 128-bit registers that supply the words representing the first (P_START) and the last (P_STOP) partition scheduled for the given instance of the algorithm. These two words are programmed during a setup stage (before the START signal) via a simple 8-bit bus (ADDR, DATA, SEL_1, SEL_2). At the output of the Partition Engine, the WORKING flag signals the activity state (i.e., when the iterations are performed), and the CNT value shows the number of executed iterations.

The top-level module shown in Fig. 7 contains M instances of Partition Engine, a Communication Controller, and one Phase Locked Loop (PLL) that doubles the input clock frequency. All the instances of the Partition Engine work in parallel and share the common START signal. There is one difference in CNT_INIT input values — the first instance of Partition Engine works with CNT_INIT $= 1$, while the others with CNT_INIT $= 0$. Such a difference avoids double-counting the last partitions, which are simultaneously used as the starting ones for the subsequent instances. All the CNT outputs from the Partition Engines are concatenated into one large ($M \times 32$-bit) shift register to enable 8-bit readouts and final summation after finishing the iterations. All WORKING flags are ORed to obtain a global flag required for counting iterations, determining the total processing time, and detecting the end of computations. The Communication Controller implements a UART interface and an additional state machine to handle a simple text-based protocol used to communicate with the host PC. The RXD and TXD pins of the UART are wired to the USB-UART bridge on the development

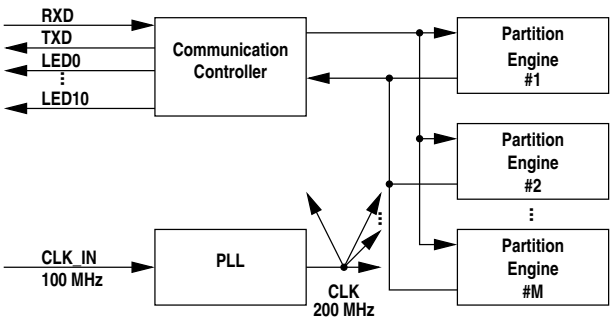

Fig. 7. The top-level schematics.

board to connect to the host PC USB port. All the modules operate synchronously with a clock from the PLL. The board is powered from the host PC via a USB cable.

The design was implemented in Vivado 2020.2 for $M = 36$ parallel instances of the Partition Engine. Preliminary estimation of the combinational path delay yielded a minimum clock period of 4.643 ns, which resulted in a 215 MHz clock frequency limit. Therefore, the design was implemented with the clock frequency of 200 MHz. More than 91% of Slice LUTs were utilized (57,985 LUTs configured as logic). Additionally, 26% of Slice Registers (33,135) were occupied. No scarce FPGA resources like block RAM and DSP slices were used. Each instance of the Partition Engine takes 1,591…1,595 LUTs and 866…962 registers, depending on the location and on the optimization results. The Communication Controller requires only 598 LUTs and 240 registers.

To check design scalability, additional configurations of the Partition Process module were designed in VHDL, synthesized and implemented for $w \in \{16, 32, 64, 256\}$, similarly to the structure shown in Fig. 4. The logic utilization for each case was then read from the post-implementation report generated in the Vivado environment. The LUT and register counts required by each instance of the Partition Engine are presented in Fig. 8 for all considered values of w. The close-to-linear relationship between the word length w and FPGA resource utilization is readily visible.

The computation of each iteration of Algorithm B takes 6 clock cycles due to pipeline implementation. The project runtime is controlled by the host PC in the MATLAB environment via a serial link. In this parallel implementation, the complete computations for $n = 127$ (generating $p(n) = 3,913,864,295$ partitions) take 652,310,727 clock cycles and last for 3.262 s, as measured by the internal cycle counter in the Communication Controller. During these computations, the development board consumes 1.74 W of supply power from the USB port (the real-time supply voltage and current were measured with the UT658DUAL USB Tester from UNI-T). It must be mentioned that the considered design is fully scalable, and the parallelization factor M is limited only by LUT availability, since the register utilization is always much lower than that of LUTs. For instance, the project could be moved to the bigger device (XC7A200T) from the Artix-7 FPGA family, which contains more than twice the available

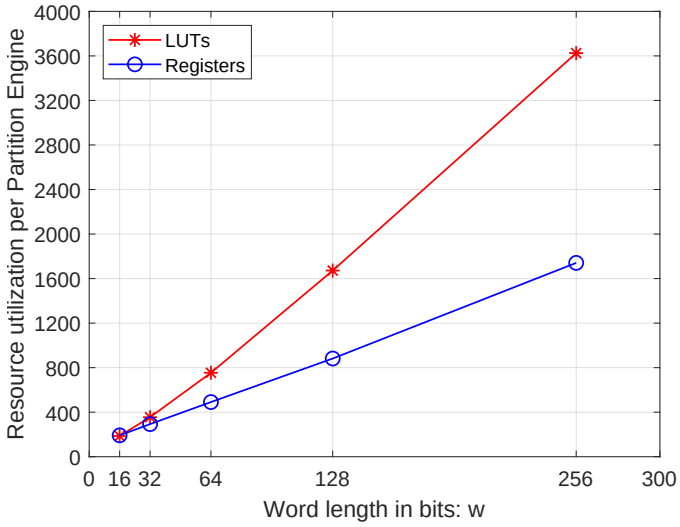

Fig. 8. FPGA resource utilization as a function word length w.

resources (134,600 LUTs). In this extended platform, up to 84 instances of the Partition Engine could be implemented (for $w = 128$), and the computation time for the same problem size of $n = 127$ would be reduced to 1.4 s, thanks to the increased parallelism.

IV. Conclusions

In this paper, a novel integer partition generation algorithm based on the bit representation was developed. Its unique features (loopless, conditionless, and pure function) lead to the efficient parallel implementations on a CPU and an FPGA, which have been described in detail. The parallel instances are independent, resulting in much better flexibility of the hardware implementation than linear systolic arrays proposed in [22]. Each instance in the FPGA implementation uses only LUTs and registers (less than 1,600 and 1,000, respectively, for $n < 128$). Hence, it is highly scalable and easily portable to different hardware platforms.

Although the single-instance FPGA implementation is four times slower than the single-threaded CPU implementation for $n_{\max} = 127$, the FPGA is a more promising platform. First, it scales better with the word length. For example, increasing the word length by 2 (to 256) in the CPU implementation results in a 2.5 times longer execution time. In contrast, in the FPGA implementation, the number of parallel instances that fit within the chip resources decreases linearly (2 times) in this case. Second, even on a cost-optimized evaluation FPGA platform used in this research, the number of parallel instances is a few times greater than the number of cores available on typical CPU processors (except the highest-end solutions). Finally, the energy advantage of the FPGA solution prevails: the energy required to compute one partition using the parallel FPGA implementation (1.45 nJ) is approximately 23 times lower than that required by the multi-threaded application on the CPU used in the experiment (33 nJ).

The presented implementations were designed only as a proof of concept. Hence, the generated partitions are just counted. However, the results prove that the fast and energy-efficient FPGA generation of combinatorial objects is feasible. The proposed approach seems thus helpful in solving open problems such as computing the average number of distinct summands of a partition [11, p. 653], the maximum number of such summands, the average number of summands [8, p. 331], or the average length of a rim representation [16, p. 144], [10, pp. 75–76]. This could be the subject of further research.

References

[1] D. E. Knuth, *The art of computer programming*. Upper Saddle River, NJ, USA: Addison-Wesley Professional, Jan. 2011, vol. 4A / Combinatorial algorithms, Part 1.

[2] J. T. Butler and T. Sasao, "High-speed hardware partition generation," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 7, no. 4, pp. 1–17, Dec. 2014. [Online]. Available: https://doi.org/10.1145/2629472

[3] T. B. Preußer and M. R. Engelhardt, "Putting queens in carry chains, №27," *Journal of Signal Processing Systems*, vol. 88, no. 2, pp. 185–201, Aug. 2017. [Online]. Available: https://doi.org/10.1007/s11265-016-1176-8

[4] P. Erdős and J. Lehner, "The distribution of the number of summands in the partitions of a positive integer," *Duke Mathematical Journal*, vol. 8, no. 2, pp. 335–345, Jun. 1941. [Online]. Available: https://doi.org/10.1215/s0012-7094-41-00826-8

[5] L. Comtet, *Advanced combinatorics. The art of finite and infinite expansions*, Revised and enlarged ed. Dordrecht, Holland: D. Reidel Publishing Company, 1974. [Online]. Available: https://doi.org/10.1007/978-94-010-2196-8

[6] H. Gupta, C. E. Gwyther, and J. C. P. Miller, *Tables of partitions*, ser. Royal Society Mathematical Tables. Cambridge: Cambridge University Press, 1962, vol. 4.

[7] J. K. S. McKay, "Algorithm 371: Partitions in natural order," *Communications of the ACM*, vol. 13, no. 1, p. 52, Jan. 1970. [Online]. Available: https://doi.org/10.1145/361953.361980

[8] A. Zoghbi and I. Stojmenović, "Fast algorithms for genegrating integer partitions," *International Journal of Computer Mathematics*, vol. 70, no. 2, pp. 319–332, Jan. 1998. [Online]. Available: https://doi.org/10.1080/00207169808804755

[9] A. Curiel and A. Genitrini, "Lexicographic unranking algorithms for the twelvefold way," in *35th International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithms (AofA 2024)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), C. Mailler and S. Wild, Eds., vol. 302. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Jul. 2024, pp. 17:1–17:14. [Online]. Available: https://doi.org/10.4230/LIPICS.AOFA.2024.17

[10] M. B. Wells, *Elements of combinatorial computing*. Oxford, UK: Pergamon Press, 1971. [Online]. Available: https://doi.org/10.1016/c2013-0-05582-X

[11] S. G. Akl and I. Stojmenović, *Parallel computing: paradigms and applications*. London, UK: International Thomoson Computer Press, 1996, ch. Generating combinatorial objects on a linear array of processors, pp. 639–670.

[12] F. Stockmal, "Algorithm 95: Generation of partitions in part-count form," *Communications of the ACM*, vol. 5, no. 6, p. 344, Jun. 1962. [Online]. Available: https://doi.org/10.1145/367766.368163

[13] E. M. Klimko, "An algorithm for calculating indices in Faà di Bruno's formula," *BIT Numerical Mathematics*, vol. 13, no. 1, pp. 38–49, 1973. [Online]. Available: https://doi.org/10.1007/bf01933522

[14] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial algorithms: Theory and practice*. Englewood Cliffs, NJ, USA: Prentice Hall, Jun. 1977.

[15] S. Comét, "On the machine calculation of characters of the symmetric group," in *Tolfte Skandinaviska Matematikerkongressen, Comptes rendus*, Lund, Sweden, Aug., 10–15 1953, pp. 18–23.

[16] ——, "Notations for partitions," *Mathematical Tables and Other Aids to Computation*, vol. 9, no. 52, pp. 143–146, Oct. 1955. [Online]. Available: https://doi.org/10.2307/2002049

[17] E. S. Page and L. B. Wilson, *An introduction to computational combinatorics*, ser. Cambridge Computer Science Texts. London: Cambridge University Press, 1979, no. 9.

[18] G. E. Andrews, *The theory of partitions*, ser. Encyclopedia of Mathematics and its Applications. Cambridge University Press, Dec. 1984, vol. 2. [Online]. Available: https://doi.org/10.1017/cbo9780511608650

[19] D. Stanton and D. White, *Constructive combinatorics*, ser. Undergraduate Texts in Mathematics. New York, USA: Springer, 1986. [Online]. Available: https://doi.org/10.1007/978-1-4612-4968-9

[20] I. Stojmenović, "Listing combinatorial objects in parallel," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 21, no. 2, pp. 127–146, Apr. 2006. [Online]. Available: https://doi.org/10.1080/17445760500355777

[21] J. K. S. McKay, "Algorithm 263: Partition generator," *Communications of the ACM*, vol. 8, no. 8, p. 493, Aug. 1965. [Online]. Available: https://doi.org/10.1145/365474.366063

[22] Z. Kokosiński and P. Halesiak, "FPGA generators of combinatorial configurations in a linear array model," in *2008 International Symposium on Parallel and Distributed Computing*. Kraków, Poland: IEEE, Jul. 2008, pp. 223–227. [Online]. Available: https://doi.org/10.1109/ispdc.2008.48

[23] Digilent, "Nexys A7 reference manual," Online, accessed May 21, 2025. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

[24] A. Zahir, A. Ullah, P. Reviriego, and S. R. U. Hassnain, "Efficient leading zero zount (LZC) implementations for Xilinx FPGAs," *IEEE Embedded Systems Letters*, vol. 14, no. 1, pp. 35–38, Mar. 2022. [Online]. Available: https://doi.org/10.1109/les.2021.3101688