Configuration Bitstream Reduction for SRAM-based FPGAs by Enumerating LUT Input Permutations

Ameer Abdelhadi and Guy G. F. Lemieux
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, B.C., V6T 1Z4, Canada
{ameer,lemieux}@ece.ubc.ca

***Abstract*— SRAM-based Field-Programmable Gate Arrays (FPGAs) are configured from off-chip memory through a serial link. Hence, a large configuration bitstream adversely increases off-chip memory size as well as bitstream loading time. The following work proposes a novel method to reduce the number of programming bits required for look-up tables (LUT), thereby reducing overall configuration bitstream size. Alternatively, the identified redundancy may be used to hide watermarking or security data. The proposed method does not affect the critical timing paths, nor does it affect the internal architecture of the LUT. The suggested method eliminates $\lfloor \log_2(k!) \rfloor$ configuration bits out of the 2^k configuration bits required by a *k*-input LUT (*k*-LUT). Hence, a 4-LUT, 5-LUT and 6-LUT only requires 12, 26, and 55 bits, respectively, to be stored in the external configuration bitstream, representing a reduction of 25%, 18.75%, and 14% in LUT configuration bits, respectively. Note the LUTs themselves still contain the full 16, 32, and 64 bits, respectively, but the missing bits are regenerated at bitstream load time. Furthermore, traditional lossless compression methods can still be employed on top of the proposed reduction technique.**

Keywords-Reconfigurable Computing; Field-programmable Gate Array (FPGA); Bitsream Compression; LUT optimization

I. INTRODUCTION

As FPGAs continue to grow in capacity, they require an increasing number of configuration bits to program the device. The cost of configuration takes the form of on-chip configuration bits, data transmission or loading time, and off-chip non-volatile storage. In particular, the bitstream loading process is usually performed over a serial link with modest speeds. For example, it takes approximately 35 seconds to configure Altera's DE4-530 board containing a Stratix IV (4SGX530) chip through a USB 2.0 link using a Linux host.

Reducing the number of configuration bits leads directly to a reduction in off-chip memory and faster loading of the configuration bitstream. One way of reducing the bitstream size is use of lossless compression techniques. However, in addition to compression, it is also possible to remove redundancies from the bitstream that can be easily regenerated. This paper identifies one such redundancy between the LUT and interconnect configuration bits that allows the complete removal of 25% of the LUT configuration bits in a traditional 4-LUT, for example.

In addition to bitstream reduction, there may be other advantages to identifying this redundancy. For example, it may be possible to encode watermarks or encryption/security data into the bitstream instead of removing the bits. Alternatively, there may be extensions of this method to reduce the number of interconnect configuration bits, which already dominate the number of LUT configuration bits.

This work identifies intrinsic redundancies in the LUT configuration bits. Although all the $2^{(2^k)}$ *k*-input logic functions can be implemented, much fewer functions are needed in practice. As shown in Fig. 1, FPGAs often allow connecting any of the Configurable Logic Block (CLB) inputs or the Basic Logic Element (BLE) feedbacks to any LUT input. If any input permutation is allowed, reduced LUTs which require arbitrary input permutations (P-class LUTs [16][17]) can be used to reduce the required configuration bits. However, the area and delay overhead of the P-class LUTs make them impractical.

Our proposed technique exploits this input permutation redundancy to eliminate a few bits in the stored configuration bitstream. Instead, the bits are regenerated at bitstream load time. Since there are $k!$ possible orderings of the inputs, one can remove $\lfloor log2(k!) \rfloor$ bits from each LUT. By enumerating each of the $k!$ possible orderings, a circuit can be used to detect the presented input ordering and regenerate the missing $\lfloor log2(k!) \rfloor$ LUT configuration bits.

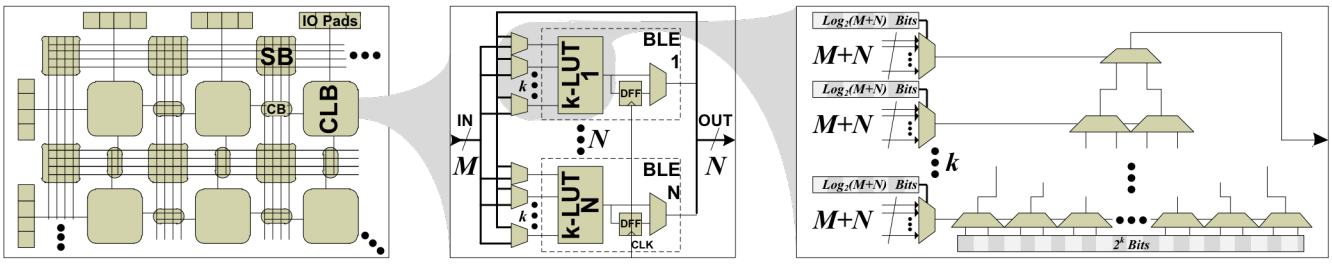

Fig. 1. Traditional architecture of a cluster/SRAM-based FPGA

For example, a 2-LUT has two different configurations for the logic function $f = a \cdot b'$. In the first configuration, the input multiplexers are configured to $(in_1 = a, in_2 = b)$ and the logic function to $f = in1 \cdot in2'$. In the second configuration, the LUT inputs are swapped, hence the input multiplexers are configured to $(in_1 = b, in_2 = a)$, with a different logic function of $f = in1' \cdot in2$. However, the input permutations do not restrict the logic function. Since the 2-LUT has two input permutations, one additional LUT configuration bit can be eliminated, say *e*. The value of *e* can depend upon the input permutation. For example, if $e = 0$ is required, the input order should be $(in_1 = a, in_2 = b)$; conversely of $e = 1$ is required, the input order should be $(in_1 = b, in_2 = a)$.[1] Hence, a tool like the router, or a bitstream-adjuster after routing, can select an appropriate input ordering to allow for the removal of the LUT configuration bit. Note that we are not forcing inputs a and b to be on specific CLB input pins, which would be extremely restrictive. Instead, we are restricting the ordering presented to the LUT, such that a appears before b. If the CLB is internally fully connected, it already allows arbitrary ordering of the LUT inputs. Generally, a *k*-LUT has $k!$ such input permutations, allowing $\lfloor log2(k!) \rfloor$ configuration bits to be removed from the bitstream and regenerated at load time.

Rather than constructing a new LUT architecture to reduce a fully functional LUT into a P-class LUT, the proposed method enumerates the input permutations and regenerates several LUT configuration bits with this enumeration. This reduces the number of LUT configuration bits while *keeping the same traditional LUT architecture* as shown in Fig. 2. Since k inputs can be permuted in $k!$ different ways, $\lfloor \log_2(k!) \rfloor$ configuration bits can be saved in each LUT. This is not bitstream compression which encodes frequently appearing patterns, this is removing a specific form information redundancy prior to compression.

The new added enumeration logic is of minimal size and can use minimum-size transistors, since it is not part of the circuit timing paths. It can even be shared by all LUTs on the device, since decompression occurs at bitstream load time.

Unfortunately, this bitstream reduction technique may not be used in some situations. For example, some high-performance FPGAs utilize the delay differences in LUT inputs for prioritizing timing paths. Also, some FPGAs have sparse connections between their CLB inputs and LUT inputs, i.e. the LUT input multiplexers are not fully connected to all of the CLB inputs. Since the proposed bitstream-reduction technique relies upon reordering LUT inputs, it may not be immediately applicable to either of these situations. However, we note that hybrid solutions are possible. For example, delay differences only matter on the critical nets, so it may be possible to apply this selectively only to non-critical nets. With sparse IIBs, removing *k!* orderings may overly restrict the routability; in this case, it may be possible to save a smaller number of bits by choosing some $k' < k$ such that routability is still good.

The rest of this paper is organized as follows. Current configuration bitstream reduction methods are reviewed in Section II. A mathematical background of indexing permutation in lexicographic order is discussed in Section III. The suggested method for enumerating LUT inputs is presented in section IV. Verification of the proposed method and results are described in Section V. Finally, this paper is concluded in Section VI.

II. Previous Work on Bitstream Reduction

Several methods have been proposed to reduce the FPGA configuration bitstream, since bitstream size adversely affects configuration memory size and configuration time. These methods can be categorized based on architecture awareness. While some methods apply general data compression methods, others exploit the internal FPGA or bitstream architecture to reduce the required bits.

General compression techniques demonstrate a high bitstream compression ratio. However, they incur high area overhead due to complex compression and decompression circuitry. These methods typically trade-off chip area and circuit complexity for an increased compression ratio.

General text compression methods, e.g. Huffman, Arithmetic, and LZ coding are adapted to bitstream compression and compared together with "don't care", readback, frame reordering, and wildcard techniques in [6]. A maximum compression factor of 4 is achieved by these methods [6]. Runlength file compression technique is used to reduce configuration bitstream by 3.6 times in [7], but bus transfer and decompression hardware overhead is required. Statistics on the Xilinx Virtex commercial FPGA family shows that less than 3% of bitstream is changed due to reconfiguration [8], hence data reuse between bitstreams of successive configurations can be used to compress the configuration bitstream [8][9].

Architecture-aware bitstream reduction techniques often suggest improving the FPGA architecture itself to reduce bits, e.g. switch boxes [10] or LUTs [11]-[17], or configuration-efficient coarse-grained architectures [18].

Several methods suggest using Universal Logic Models (ULMs) to generate optimal NPN-class LUTs [11][12]. However, ULMs with additional redundant inputs are

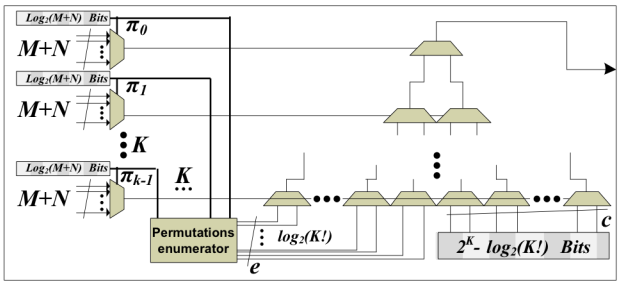

Fig. 2. Enumerating input permutations to reduce configuration bits for a SRAM-based LUT

[1] Permuting the inputs of the current LUT based upon the current LUT's own function leads to a cyclic dependence: after the new input order is determined, the LUT configuration bits must be rearranged accordingly, which will likely change the value of the bits to be removed, which may require a different input ordering. We suggest breaking this cyclic dependence using a chain: the input ordering for the next LUT is determined from the bits removed in the current LUT; this fixes the next LUT's configuration bits and enables the next removal.

impractical for SRAM-based FPGAs since the additional ULM pins require additional routing resources and steering configuration bits which often swamp the LUT bit savings.

The current trend of ULM research is to investigate functionally incomplete LUTs, namely, LUTs which eliminate rarely-used functions [13][14]. These ULMs are therefore incapable of generating all logic functions of the input variables. The usefulness of functional NPN-classes is usually investigated by statistical means, with only the most useful classes considered for LUT implementation. Although this method could save a large amount of LUT configuration bits, it incurs an increased routing and area overhead.

Another method [12][15] proposes a functional-complete ULM with no additional inputs to implement LUTs for SRAM-based FPGAs. Binary Decision Diagrams (BDDs) are employed to construct ULMs with reduced configuration bits, while considering different input permutations and negation (NP-classes). However, for ULMs with more than 3 inputs, this method suffers from increased overhead area and design complexity.

Our method is based on LUT optimization. However, it does not suffer from area overhead. Furthermore, the proposed method can be applied together with existing compression methods, hence, increasing bitstream reduction, with minimal area overhead.

III. Indexing All $k!$ Permutations in Lexicographic Order: Mathematical Background

Methods from algebraic combinatoric theory [1][2][3] are employed here to enumerate input permutations. A structural and minimal circuit is then proposed to generate a permutation enumeration code (Lehmer's code) and to convert this code into a binary representation. Rather than using factorial mixed-radix representation to convert Lehmer's code into a binary representation, which consumes high area due to factorial multiplications, an algorithmic method is proposed to convert Lehmer's code into a binary representation using only a few logic gates. Input permutation enumeration is then exploited to configure the LUT logic function.

Inputs of a fully utilized k-LUT are routed to different k inputs; hence the linear order of the chosen inputs represents a *permutation* of the finite set, $K \equiv \{0,1,\dots,k-1\}$. A permutation π is defined as a linear ordering of a set of elements K [2], and can be described notationally using a list of different k elements out of K in square brackets

$$\pi = [\pi_0 \pi_1 \dots \pi_{k-1}] | \forall_{i,j \in K}\, \pi_i \in K, \pi_i \neq \pi_j. \quad (1)$$

However, in group theory, a permutation π of a set K is described as a bijection from that set to itself [5]. Hence, the k-permutation group is defined as the *finite symmetric group* S_k,

$$S_k \equiv \{\pi | \pi: K \to K, bijectively\}. \quad (2)$$

The set of *Inversion* [1][2][4][5] $I(\pi)$ of a permutation π is defined as a set of pairs representing the places of two successive elements in a permutation, such that the values of these two elements are reversed to their place order, namely,

$$I(\pi) \equiv \left\{(i,j) \in \underline{k}^2 \;\middle|\; i < j, \pi_i > \pi_j\right\}. \quad (4)$$

The *Inversion indicator* $1_I^{(i,j)}$ is 1; if and only if (i,j) is an inversion as defined previously

$$1_I^{(i,j)}(\pi) \equiv \begin{cases} 1, & if\ (i,j) \in I(\pi) \\ 0, & otherwise \end{cases}. \quad (5)$$

The *Lehmer* code [1][5] for a place i in a permutation π is defined as the amount of inversions in π with i as the first ordered place in the inversion,

$$l_i(\pi) \equiv \left|\left\{i < j \middle| \pi_i > \pi_j\right\}\right| \equiv \sum_{j \in n} 1_I^{(i,j)}(\pi). \quad (6)$$

Hence, $l_i(\pi)$ is the number of elements that are placed after i in a permutation π, but smaller than i.

Since $1_I^{(i,j)}$ is 1, only if $i < j$, the Lehmer code could be defined as

$$l_i(\pi) = \sum_{j | i<j} 1_I^{(i,j)}(\pi). \quad (7)$$

The Lehmer code for a permutation π is defined as a list of the Lehmer code for the subsequent permutation elements,

$$L(\pi) \equiv \overline{l_0(\pi) l_1(\pi) \dots\ l_{k-1}(\pi)}, \quad (8)$$

for example, $L([57024631]) \equiv \overline{56012210}$.

Since k is known, and $k-1$ is the largest index, $l_{k-1}(\pi)$ is zero and can be dropped from the Lehmer Code. Hence, $L^+(\pi)$ is defined as

$$L^+(\pi) \equiv \overline{l_0(\pi) l_1(\pi) \dots\ l_{k-2}(\pi)}, \quad (9)$$

for example, $L^+([57024631]) \equiv \overline{5601221}$.

To reconstruct a permutation back from a Lehmer code,
π_0 is the $(l_0(\pi)+1)-th$ element of K,
π_1 is the $(l_1(\pi)+1)-th$ element of $K/\{\pi_0\}$,
π_2 is the $(l_2(\pi)+1)-th$ element of $K/\{\pi_0, \pi_1\}$,
…etc.

Since there is only one way to convert from a permutation to Lehmer's code and vice versa, the Lehmer code is a bijection between π and $L^+(\pi)$ in $\mathbb{N}^{\underline{k}}$ [5], hence Lehmer code is unique, and can be used to enumerate the permutations [10] since the Lehmer code digits $l_i(\pi)$ are unrelated to each other. Furthermore, the following properties are trivially true,

$$\forall i \in \underline{k}: l_i(\pi) \leq k - i. \quad (10)$$

Hence, a Lehmer code for all permutations represents a successive sequence of numbers in the *factorial number system*, namely a lexicographic enumeration for all $k!$ permutations.

The factorial number system or *Factoradic* system is a *mixed radix numeral system* [3], where the right $i-th$ digit has a base of i, hence, should be less than i, as satisfied in the previous inequality. The $i-th$ digit has a decimal place value of $(i-1)!$, therefore the decimal value of the Lehmer's code, representing a factorial number is

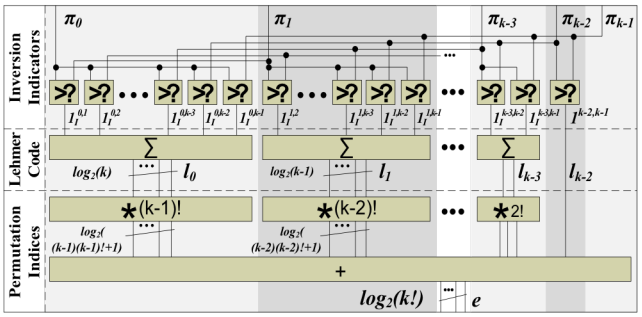

Fig. 3. Permutations fully-enumerator: (>?) are comparators, Σ's count 1's in input

$$L_D(\pi) = \sum_{i=0}^{k-2} l_i(\pi)(k-i-1) \tag{11}$$

Fig. 3 describes a full enumerator that produces binary indexes of all the $k!$ input permutations. The enumerator is combined of three stages. In the first stage, each input is compared with the successive inputs to generate inversion indicators. The second stage sums up the active inversion indicators (counts 1's in inputs) for each input to generate Lehmer's code. The final stage converts Lehmer's code into a binary representation by multiplying each Lehmer's code digit with its factorial digit place value. Finally, all factorial digits are added together to generate the permutation representative index.

IV. Enumerating LUT Input Permutations

Lehmer's code provides an enumeration of all the $k!$ input permutations in lexicographic order and can be represented as a factorial mixed-radix number in order to be converted into decimal. The conversion from the factorial mixed-radix system into a decimal requires significant calculations which include multiplication of each digit with its representative factorial number. Nevertheless, for LUT configuration purposes, only $2^{\lfloor \log_2(k!) \rfloor}$ permutations out of $k!$ permutations need to be enumerated. Furthermore, the lexicographic order is not essential. Reduced logic, which includes only a few logic gates to generate $2^{\lfloor \log_2(k!) \rfloor}$ permutations out of the $k!$ total permutations in unspecified order is proposed. The enumeration circuit is obtained by structural algorithms for grouping Lehmer's code digits then mapping these digits into a binary representation.

A. Binary enumeration of $2^{\lfloor \log_2(k!) \rfloor}$ out of $k!$ permutation in unspecified order

The previous permutation full enumerator consumes high area due to the factorial multipliers and the wide adder at the last stage. However, $\lfloor \log_2(k!) \rfloor$ of the LUT configuration bits will be replaced with a permutation enumeration vector. Hence, this enumeration vector should include all binary values, namely $2^{\lfloor \log_2(k!) \rfloor}$ binary values. Therefore, only $2^{\lfloor \log_2(k!) \rfloor}$ out of all $k!$ permutation are needed. Furthermore, the actual lexicographic order is not critical for correct LUT functionality.

```
G = groupDigits(k):
1  G = ∅;
2  for (i=k-2 ; i>=0 ; i--) {
3    totalBits += ⌊log2(k-i)⌋;
4    G ∪= { {i} }
5  }
6  while (totalBits < ⌊log2(k!)⌋) {
7    for (i=k-2 ; i>=0 ; i--)
8      if ({i}∈G AND ⌊log2(k-i)⌋<log2(k-i))
9        break;
10   for (j=i+1 ; j>=0 ; j--)
11     if ({j}∈G AND ⌊log2((k-i)*(k-j))⌋ >
                  ⌊log2(k-i)⌋+⌊log2(k-j)⌋){
12       totalBits++;
13       G /= { {i},{j} }
14       G ∪= { {i,j} }
15       break;
16     }
17 }
18 return G;
```

(a)

groupDigits(k) execution example:

K = 2 → G = { {0} }
K = 3 → G = { {1} , {0} }
K = 4 → G = { {2} , {1} , {0} }
K = 5 → G = { {3} , {2} , {1} , {0} }
K = 6 → G = { {4} , {3,0} , {2} , {1} }
K = 7 → G = { {5} , {4,1} , {3} , {2,0} , {1} }
K = 8 → G = { {6} , {5,2} , {4} , {3,1} , {1} , {0} }

(b)

Fig. 4. (a) groupDigits algorithm (b) execution example

The generated Lehmer's code numbers are independent, namely, the value of any Lehmer's code number $l_i(\pi)$ is unrelated to other numbers $l_j(\pi) \mid j \neq i$. Therefore, each Lehmer's code digit can be enumerated separately. For each Lehmer's code digit l_i, the maximum number of generated binary bits is $\lfloor \log_2(k-i) \rfloor$. Hence, enumerating each Lehmer's code digit separately could generate fewer than the $\lfloor \log_2(k!) \rfloor$ number of required bits. To overcome this problem, Lehmer's code digits can be grouped before the binary enumeration process. If two Lehmer's digits l_i and l_j are grouped together, they will supply $\lfloor \log_2((k-i)\cdot(k-j)) \rfloor$ bits, which may be larger than the sum of contribution of each bit separately.

The *GroupDigits* algorithm described in Fig. 4 aims to find the minimal groups of Lehmer's digits such that the following equation will be satisfied

$$\sum_{g \in G} \left\lfloor log_2 \left(\prod_{i \in g} k-i \right) \right\rfloor = \lfloor log_2 k! \rfloor, \tag{12}$$

Where G is the group of all digit groups.

The proposed algorithm starts with separated Lehmer's digits, and tries incrementally to group minimal digits such that the overall bits will reach the required $\lfloor \log_2(k!) \rfloor$ bound.

```
f = mergeDigits(k,i,j):
1  for (b_{i,j}=0 ; b_{i,j}<=2^{bits(l_i)+bits(l_j)-1} ; b_{i,j}++) {
2    if (2^{bits(l_i)}-max(l_i) > 2^{bits(l_j)}-max(l_j))
3      swap(i,j);
4    l_i⟦bits(l_i)-1…0⟧ = b_{i,j}⟦bits(l_i)-1…0⟧ % max(l_i)+1;
5    if (b_{i,j}⟦bits(l_i)-1…0⟧ > max(l_i)+1) {
6      l_j⟦bits(l_j)-1…0⟧ = ⟪'1',b_{i,j}⟦bits(l_j)+bits(l_i)-2…bits(l_i)⟧⟫ % max(l_j)+1;
7      l_j⟦bits(l_j)-1⟧ = '1';
8    } else
9        l_j⟦bits(l_j)-1…0⟧ = ⟪'0', b_{i,j}⟦bits(l_j)+bits(l_i)-2…bits(l_i)⟧⟫;
10 }
11 Find mapping function f, s.t. b=f(l_i,l_j);
12 return f;
```

(a)

Notation:	
l_i	Digit i of Lehmer's code
$\max(l_i) = k-i$	Maximum value for Lehmer digit l_i
$\text{bits}(l_i) = \lceil \log_2(k-i) \rceil$	Number of bits required to represent Lehmer's digit l_i
$\langle\langle b_{n-1},..,b_0 \rangle\rangle$	Encloses a binary vector
$a[\![i]\!]$	The i-th bit of a binary vector 'a' (little Indian; $a[\![0]\!]$ is the LSB)
$a[\![j…i]\!]$	$\langle\langle a[j],a[j-1],…,a[i] \rangle\rangle \mid j<i$
$b_{i,j}$	the combined l_i and l_j, $\text{bits}(l_i)+\text{bits}(l_j)-1$ bits

(b)

Fig. 5. (a) *mergeDigits* algorithm (b) notation

B. Merging grouped digits into one binary representation

Grouped digits l_i, l_j should provide enumeration for $2^{\lfloor \log_2((k-i)\cdot(k-j)) \rfloor}$ different numbers. This enumeration is performed by mapping all of the $\lfloor \log_2((k-i)\cdot(k-j)) \rfloor$ enumeration binary numbers into l_i and l_j. The *mergeDigits* algorithm needed to generate such a mapping is described in Fig. 5. Without loss of generality, a maximum value of l_i is assumed to be closer to the maximal value that could be represented by l_i's bits, compared to l_j, namely,

$$2^{\lceil log_2(k-i) \rceil} - (k-i) < 2^{\lceil log_2(k-j) \rceil} - (k-j). \quad (13)$$

For all values of the merged binary vector b:

- l_i is mapped to the relevant LSB bits of b modulo the maximum value of l_i
- if the previous modulo operation overflows, then
  - l_j is mapped to the rest of b, except $l_j's$ MSB bit which is mapped to '1', all modulo the maximum value of l_j
  - $l_j's$ MSB bit is set to '1'
- otherwise, l_j is mapped to the rest of b, except $l_j's$ MSB bit which is mapped to '0'

For $(k,i,j) = (6,3,0)$, $(k,i,j) = (7,4,1)$ or $(k,i,j) = (8,5,2)$ the same mapping is achieved since $k-i=3$ and $k-j=6$ for all of them, as listed in Table I. For $(k,i,j) = (7,2,0)$ or $(k,i,j) = (8,3,1)$ the same mapping is achieved since $k-i=5$ and $k-j=7$ for all of them, as listed in Table II. The mapping function f can be obtained manually or by logic optimization tools.

```
enumerate(k):
1  eIndex = 0;
2  G = GroupDigits(k);
3  foreach g∈G {
4    if (|g| == 1) {
5      {i} = g;
6      for(lIndex=0 ; lIndex<⌊log2(k-i)⌋
                                  ; lIndex++)
7        e⟦eIndex++⟧ = l_i⟦lIndex⟧;
8    } else
9      {i,j} = g;
10     f = mergeDigits(k,i,j);
11     for (lIndex=0 ; lIndex<(⌊log2(k-i)⌋
                +⌊log2(k-j)⌋-1) ; lIndex++)
12       e⟦eIndex++⟧ = f(l_i,l_j)⟦lIndex⟧;
13   }
14 }
```

(a)

Notation:	
e	Binary vector of $\lfloor \log_2(k!) \rfloor$ bits; contains final enumeration

(b)

Fig. 6. (a) *enumerate(k)* algorithm (b) notation

TABLE II: MERGEDIGITS FOR (K,I,J)=(6,3,0), (7,4,1) OR (8,5,2)

$b_{i,j}[\![]$					$l_i[\![]$			$l_j[\![]$		
4	3	2	1	0	2	1	0	2	1	0
0	0	0	0	0	→ 0	0	0	0	0	0
0	0	0	0	1	→ 0	0	0	0	0	1
0	0	0	1	0	→ 0	0	0	0	1	0
0	0	0	1	1	→ 0	0	0	0	1	1
0	0	1	0	0	→ 0	0	0	1	0	0
0	0	1	0	1	→ 0	0	0	1	0	1
0	0	1	1	0	→ 0	0	0	1	1	0
0	0	1	1	1	→ 1	0	0	0	0	0
0	1	0	0	0	→ 0	0	1	0	0	0
0	1	0	0	1	→ 0	0	1	0	0	1
0	1	0	1	0	→ 0	0	1	0	1	0
0	1	0	1	1	→ 0	0	1	0	1	1
0	1	1	0	0	→ 0	0	1	1	0	0
0	1	1	0	1	→ 0	0	1	1	0	1
0	1	1	1	0	→ 0	0	1	1	1	0
0	1	1	1	1	→ 1	0	0	0	0	1
1	0	0	0	0	→ 0	1	0	0	0	0
1	0	0	0	1	→ 0	1	0	0	0	1
1	0	0	1	0	→ 0	1	0	0	1	0
1	0	0	1	1	→ 0	1	0	0	1	1
1	0	1	0	0	→ 0	1	0	1	0	0
1	0	1	0	1	→ 0	1	0	1	0	1
1	0	1	1	0	→ 0	1	0	1	1	0
1	0	1	1	1	→ 1	0	0	0	1	0
1	1	0	0	0	→ 0	1	1	0	0	0
1	1	0	0	1	→ 0	1	1	0	0	1
1	1	0	1	0	→ 0	1	1	0	1	0
1	1	0	1	1	→ 0	1	1	0	1	1
1	1	1	0	0	→ 0	1	1	1	0	0
1	1	1	0	1	→ 0	1	1	1	0	1
1	1	1	1	0	→ 0	1	1	1	1	0
1	1	1	1	1	→ 1	0	0	0	1	1

Mapping function f:

$b_{i,j}[\![0]\!]=l_i[\![0]\!] \vee l_j[\![2]\!]$

$b_{i,j}[\![1]\!]=l_i[\![1]\!] \vee l_j[\![2]\!]$

$b_{i,j}[\![2]\!]= l_j[\![0]\!]$

$b_{i,j}[\![3]\!]=(l_i[\![1]\!] \vee l_j[\![2]\!]) \wedge l_i[\![0]\!]$

TABLE I: MERGEDIGITS FOR (K,I,J)=(7,2,0) OR (8,3,1)

$b_{i,j}[\![]$					$l_j[\![]$			$l_i[\![]$	
3	2	1	0		2	1	0	1	0
0	0	0	0	→	0	0	0	0	0
0	0	0	1	→	0	0	0	0	1
0	0	1	0	→	0	0	0	1	0
0	0	1	1	→	1	0	0	0	0
0	1	0	0	→	0	0	1	0	0
0	1	0	1	→	0	0	1	0	1
0	1	1	0	→	0	0	1	1	0
0	1	1	1	→	1	0	1	0	0
1	0	0	0	→	0	1	0	0	0
1	0	0	1	→	0	1	0	0	1
1	0	1	0	→	0	1	0	1	0
1	0	1	1	→	1	0	0	0	1
1	1	0	0	→	0	1	1	0	0
1	1	0	1	→	0	1	1	0	1
1	1	1	0	→	0	1	1	1	0
1	1	1	1	→	1	0	1	0	1

Mapping function f:

$b_{i,j}[\![0]\!]=l_i[\![0]\!] \vee l_j[\![2]\!]$

$b_{i,j}[\![1]\!]=l_j[\![1]\!] \vee l_i[\![2]\!]$

$b_{i,j}[\![2]\!]=l_j[\![2]\!] \vee l_i[\![2]\!]$

$b_{i,j}[\![3]\!]=(l_i[\![0]\!] \vee l_i[\![2]\!]) \wedge l_j[\![0]\!]$

$b_{i,j}[\![4]\!]=(l_i[\![1]\!] \vee l_i[\![2]\!]) \wedge l_j[\![0]\!]$

C. *Generating final enumeration vector*

Mapping to the final enumeration vector is based on the digits grouping procedure, achieved by *groupDigits(k)* and shown in Fig. 6. If a Lehmer's digit l_i is grouped with another Lehmer's digit l_j, these digits will be merged together to achieve a mapping function *f=mergeDigits(k,i,j)*, then $\lfloor \log_2((k-i)\cdot(k-j)) \rfloor$ bits will be mapped through f to the final enumeration vector. Examples of the generated enumeration circuits and implementation details for k=3 to 8 are depicted in Fig. 7 and Fig. 8, respectively.

D. *Mapping LUT's logic functions to input permutations*

To map $2^{(2^k)}$ logic functions of a k-LUT to their respective permutations, a list of all input permutations is generated iteratively. The enumeration of each permutation is calculated and attached to the relevant permutation in the permutations list. The proposed permutation list should contain $2^{\lfloor \log_2(k!) \rfloor}$ entries. Each entry contains a k-numbers permutation, and is indexed by the permutation enumeration. Assuming that a permutation number is one byte, the mapping list size is $k \cdot 2^{\lfloor \log_2(k!) \rfloor} bytes$. For instance, an 8-LUT mapping list requires 256 KB. Another possible method is backtracking the required enumeration through the enumeration logic to detect the relevant permutation.

Table III gives a logic function-to-permutations mapping list and Fig. 9 shows an implementation for a 2-LUT.

V. IMPLEMENTATION RESULTS

A single enumerator can decompress the bitstream for the entire FPGA device. The proposed design has been implemented in Verilog and synthesized using Synopsys Design Compiler with the TSMC 65nm standard cell library. The implementation area overhead is given in Table IV; only 1000 transistors are required for 6-LUT architectures.

The implementation has been verified by generating output enumeration for *all* input permutations. The output enumeration vector covers all possible binary combinations. Gate-level simulation (GLS) on the synthesized netlist was also done to verify the correctness of the logic.

Relevant C code, Verilog files, and synthesis scripts are located on the authors' website [19].

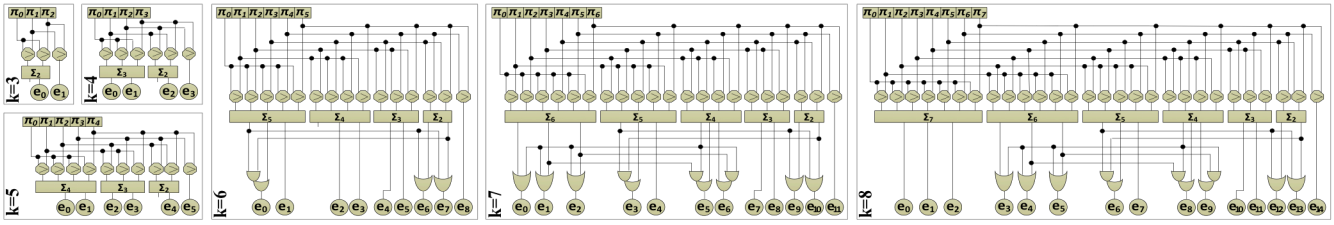

Fig. 7. Enumeration circuit for k=3…8; Σ's can be implemented by custom logic as depicted in Fig. 8.

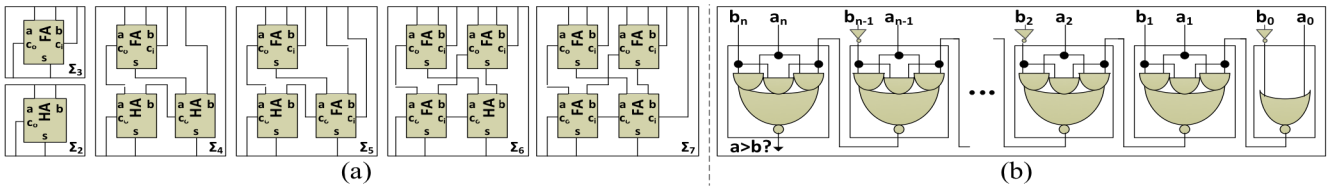

Fig. 8. Custom implementation of (a) $\Sigma_{2..7}$ (b) low-area ripple comparator using CMOS majority gates

TABLE III: MAPPING 2-LUT LOGIC FUNCTIONS TO INPUT PERMUTATIONS

f \ a,b	00	01	10	11	function	index e_0	c_2	c_1	c_0	value permutation
f_0	0	0	0	0	0	0	0	0	0	(a,b)
f_1	0	0	0	1	a∧b	0	0	0	1	(a,b)
f_2	0	0	1	0	a∧¬b	0	0	1	0	(a,b)
f_3	0	0	1	1	a	0	0	1	1	(a,b)
f_4	0	1	0	0	¬a∧b	0	1	0	0	(a,b)
f_5	0	1	0	1	b	0	1	0	1	(a,b)
f_6	0	1	1	0	a⊕b	0	1	1	0	(a,b)
f_7	0	1	1	1	a∨b	0	1	1	1	(a,b)
f_8	1	0	0	0	¬(a∨b)	1	0	0	0	(b,a)
f_9	1	0	0	1	¬(a⊕b)	1	0	0	1	(b,a)
f_{10}	1	0	1	0	¬b	1	0	1	0	(b,a)
f_{11}	1	0	1	1	a∨¬b	1	0	1	1	(b,a)
f_{12}	1	1	0	0	¬a	1	1	0	0	(b,a)
f_{13}	1	1	0	1	¬a∨b	1	1	0	1	(b,a)
f_{14}	1	1	1	0	¬(a∧b)	1	1	1	0	(b,a)
f_{15}	1	1	1	1	1	1	1	1	1	(b,a)

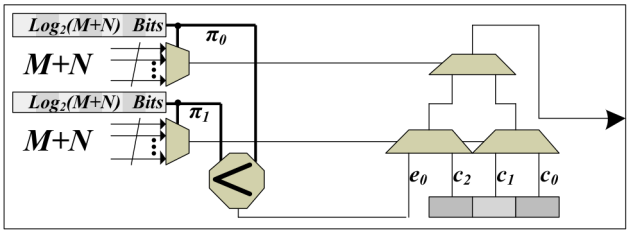

Fig. 9. 2-LUT with input permutation enumerator

VI. CONCLUSIONS

A method for removing LUT configuration bits is presented in this paper. By reducing the number of LUT bits stored in the configuration bitstream, off-chip memory size and bitstream loading time can be reduced. Minimal silicon area is required for the decoder. The technique works by removing information redundancy about the LUT input ordering stored collectively in both the LUT configuration bits and the internal CLB interconnect bits. The proposed LUT input enumerator is synthesized into a gate-level netlist and logically verified. The proposed method can be employed together with existing bitstream compression methods to achieve a maximal compression ratio.

Future improvements of the suggested method should be considered. Generalization of the enumeration method for CLBs with specific routing constraints, e.g. CLBs with sparse connection or partially fixed routing, should be considered. Furthermore, since the majority of the configuration bits are dedicated for routing, extending the technique for routing configuration bits may be helpful. Alternatively, other uses for the information redundancy can be explored, such as watermarking.

TABLE IV: CELL AREA AND TRANSISTOR COUNT FOR PROPOSED CIRCUIT

k	3	4	5	6	7	8
Area (μm)	63.7	129.2	220.0	333.0	501.5	702.4
# transistor	198	404	694	1058	1576	2208

REFERENCES

[1] D. H. Lehmer, "Teaching combinatorial tricks to a computer," *Proc. of Symp. in Applied Math.*, vol. 10: Combinatorial Analysis, Amer. Math. Society, pp. 179-193, 1960.

[2] M. Bona, *Combinatorics of Permutations*. Chapman and Hall, 2004.

[3] D. E. Knuth, *The art of computer programming, Volume 2: Seminumerical Algorithms,* 3rd Ed., Addison-Wesley, 1997.

[4] D. E. Knuth, *The art of computer programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison-Wesley, 1998.

[5] A. Kerber, *Algebraic Combinatorics Via Finite Group Actions*, B.I. Wissenschaftsverlag, 1991.

[6] Z. Li and S. Hauck, "Configuration compression for Virtex FPGAs," *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 147–159, 2001.

[7] S. Hauck and W. Wilson, "Runlength compression techniques for FPGA configurations," *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pp.286-287, 1999.

[8] I. Kennedy, "Exploiting Redundancy to Speedup Reconfiguration of an FPGA," *Field-Programmable Logic and Applications (FPL)*, pp. 262–271, 2003.

[9] J. H. Pan, T. Mitra, and W.-F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 766–773, 2004.

[10] W. Chong, M. Hariyama, and M. Kameyama,"Novel switch-block architecture using reconfigurable context memory for multi-context FPGAs", *International Workshop on Applied Reconfigurable Computing (ARC)*, pp.99–102, 2005.

[11] F. P. Preparata and D. E. Muller, "Generation of near-optimal universal Boolean functions," *Journal Computer and System Sciences*, vol. 4, no. 2, pp. 93–102, 1970

[12] X. Chen and X. Wu, "Derivation of universal logic modules, for n≥3, by algebraic means," *IEEE Computers and Digital Techniques*, vol. 128, pp. 205–211, 1981.

[13] Y. Okamoto, Y. Ichinomiya, M. Amagasaki, M. Iida, and T. Sueyoshi, "COGRE: A Configuration Memory Reduced Reconfigurable Logic Cell Architecture for Area Minimization," *Field Programmable Logic and Applications (FPL)*, pp. 304–309, 2010.

[14] J. Meyer and F. Kocan, "Sharing of SRAM tables among NPN-equivalent LUTs in SRAM-based FPGAs," *IEEE Transactions on Very Large Scale Integrated Syst.*, vol. 15, no. 2, pp. 182–195, 2007.

[15] Z. Zilic and Z. G. Vranesic, "Using Decision Diagrams to Design ULMs for FPGAs," *IEEE Transactions on Computers*, vol. 47, no. 9, pp. 971–982, 1998.

[16] C.-C. Lin, M. Marek-Sadowska, and D. Gatlin, "Universal logic gate for FPGA design," *IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD)*, pp. 164–168, 1994.

[17] S. Thakur and D. F. Wong, "On Designing ULM-Based FPGA Logic Modules," *ACM/SIGDA FPGA*, pp. 3–9, 1995.

[18] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective", *Proceedings of Design, Automation and Test in Europe (DATE)*, pp. 642–649, 2001.

[19] http://www.ece.ubc.ca/~lemieux/downloads