Minimal Path Delay Leading Zero Counters on Xilinx FPGAs

Gregory Morse[1](✉), Tamás Kozsik[1], and Péter Rakyta[2]

[1] Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary
morse@inf.elte.hu

[2] Department of Physics of Complex Systems, Eötvös Loránd University, Budapest, Hungary

Abstract. We present an improved efficiency Leading Zero Counter for Xilinx FPGAs which improves the path delay while maintaining the resource usage, along with generalizing the scheme to variants whose inputs are of any size. We also show how the Ultrascale architecture also allows for better Intellectual Property solutions of certain forms of this circuit with its newly introduced logic elements. We also present a detailed framework that could be the basis for a methodology to measure results of small-scale circuit designs synthesized via high-level synthesis tools. Our result shows that very high frequencies are achievable with our design, especially at sizes where common applications like floating point addition would require them. For 16, 32 and 64-bit, our real-world build results show a 6%, 14% and 19% path delay improvement respectively, enough of an improvement for large scale designs to have the possibility to operate close to the maximum FPGA supported frequency.

Keywords: FPGA · MaxCompiler · Leading Zero Counters · High-Level Synthesis · Vivado

1 Introduction

Leading Zero Counters (LZC [4] are of importance for various bit-level tasks, most notably floating point addition and subtraction [10,12,18] such as in the IEEE-754 standard. This is due to an effective subtraction when the signs are of opposite polarity with addition, or identical polarity with subtraction. Rather than the leading bit being one more, one less or identical to the original mantissa size (adjusted for its guard and round bits), which are also the three cases where rounding will occur, there also are all the cases where the result has some number of less digits up to the mantissa size plus the guard and round bit or be an all zero result. Since the exponent will need to be adjusted as well as the zero case handled, all of this information is essential. Detecting the all zero case can be merged into the logic of the leading zero detect as its intermediate information must be propagated anyway.

In fact, a traditional clever use of floating point units (FPUs) addition/subtraction unit has been using the normalization process post-subtraction with

J. Mikyška et al. (Eds.): ICCS 2023, LNCS 14075, pp. 626–640, 2023.
https://doi.org/10.1007/978-3-031-36024-4_48

custom byte-packing [2], by inserting an integer in the mantissa m and setting the exponent to $e = b - 1$ where b is the mantissa size of the data type (e.g. $b = 24$ for float32, $b = 53$ for float64). The floating point value is $m * 2^e$. There is always an implied 2^e added to the stored mantissa, in the IEEE standard variants. Then by subtracting the exponent part 2^{b-1}, the exponent becomes the leading zero count. For more details, see the well known C implementations of *Find the integer log base 2 of an integer with an 64-bit IEEE float* which is a prime use case [2]. As an example, a double-precision number would set the 12-bits representing the exponent to $1023 + 53 - 1 = 0x433$ where 1023 is the exponent bias and the 52-bits of mantissa to the value to determine the LZC. By subtracting 2^{53-1}, it effectively subtracts out the implied one at the 53rd position, and requires re-normalizing, which places the leading zeros as the exponent. Merely subtracting the bias or 1023 from the 12-bits of the exponent, yields the LZC.

More formally, we define the LZC-n for bit-vector $X_{n..1}$ as an ordered pair (V, C) where

$$V = \bigwedge_{K=n}^{1} \overline{X_k} = \overline{X_n} \wedge \overline{X_{n-1}} \wedge \cdots \wedge \overline{X_1} \tag{1}$$

is the all-zero signal and

$$z(i,j) = \left(\bigvee_{k=n-2^{i+j}}^{n-2^{i+j+1}} X_k \right) \vee \left(\bigwedge_{k=n-2^{i+j+1}}^{n-2^{i+j+2}} \overline{X_k} \wedge z(i,j+2) \right) \tag{2}$$

$$C = \mathop{\Big|\Big|}_{i=0}^{\lceil \log_2 n \rceil - 1} \left(V \vee \left(\bigwedge_{k=n}^{n-2^i} \overline{X_k} \wedge z(i,0) \right) \right) \tag{3}$$

represents the leading zero count as a bit-string (which is built via the concatenation operator $||$) in Boolean algebra as an infinite recurring relationship (where $\vee$ and $\wedge$ are logical OR and logical AND respectively). In our notation, a bar above represents a logical negation and $\lceil x \rceil$ is the ceiling operation of rounding x up to the nearest integer. In the special case that X contains all zeros, then V and all bits of C are set to 1. In some contexts, such as one-hot decoding, these artifacts may be unnecessary. However as will be seen, their intermediate computation is convenient regardless.

Although traditionally a focus on power is prevalent, we have chosen to focus on performance, then area and only minimize power if it does not effect performance or area. As higher area allows more concurrency and thus more performance, our justification for high-performance computing (HPC) is due to work in the area of Quantum simulation. But an investigation into the latest offerings for HPC in Ultrascale and Vivado is thus forthcoming.

Our contribution is thus a more general framework which uses careful and precise integration of a more modular framework, which yields a better result. Expert re-synthesis of integrated units of a modular design, can unsurprisingly yield a better state-of-the-art result. The exact ideas and optimizations used are important in a broader range of circuits.

1.1 Efficient Logic Block Usage on Xilinx FPGAs

This discussion is in part intended to draw attention to the complexity and multi-layered and even ambiguous approach the tools take to configuring the more advanced FPGA features. Various settings being utilized at various stages inclusively or exclusively with in some cases profound effects on the output make Vivado an expert tool on a per platform or even per module basis. It intends to draw attention to the reader all attributes and settings which are relevant or worth of consideration in this specific context.

Xilinx FPGAs have long utilized Look-Up Table (LUT) 6's with 6 input signals and one output signal [1] providing a realization of a generic 6 input binary function $f(x_0, x_1, x_2, x_3, x_4, x_5)$ where $x_k \in \{0,1\} \forall k \in \{0,\dots,5\}$, thus appropriate to realize high performance signal processing implementations [8]. 4 LUT-6 appear in a single slice in the architecture. A LUT-6 can also be thought of as a 4-to-1 multiplexer where 4 inputs are selected from based on the other 2 inputs as

$$f(x_0, x_1, x_2, x_3, x_4, x_5) = \begin{cases} x_0 & \text{if } \overline{x_4} \wedge \overline{x_5} \\ x_1 & \text{if } \overline{x_4} \wedge x_5 \\ x_2 & \text{if } x_4 \wedge \overline{x_5} \\ x_3 & \text{otherwise} \end{cases} \tag{4}$$

where the number encoded by the 2 binary inputs can be used to govern one of the remaining 4 inputs onto the output. However modern Xilinx 7-series and Ultrascale FPGAs also offer additional features. The *LUT N:M* (LUTNM) VHDL feature allows combining logical LUTs into a single physical LUT with multiple outputs (explicitly specified by VHDL property *LUTNM* and its hierarchical counterpart *HLUTNM*). This is ubiquitous as its presence is in both the logic slices (*SLICEL*) and the memory slices (*SLICEM*) the latter of which can be repurposed as Shift-Register LUTs (SRLs). It will occur by inference during synthesis if the no LUT combining option (*-no_lc*) is absent as well as during placement when no physical synthesis in placer (*-no_psip*) is absent [15]. Since the LUT-6 are actually implemented as an element called a *LUT6-2* as seen in Fig. 1, the flexibility of $N \leq 5$ common inputs mapped to $M = 2$ outputs has allowed increased efficiency involved with common design patterns. In practice this means a realization of two 5-input LUTs with common inputs and having 2 separate outputs [13], as the 6-th input is used to multiplex between the 2 outputs. Therefore, any LUT-2 and LUT-3 without common inputs can be easily combined into a single instance of LUT6-2. Likewise two LUT-3 which share a single common input would also be combine-able. In 1, x_5 is the 6th input used for multiplexing, while $x_0, \dots, x_4$ are the 5-common inputs to the two LUT-5s. As can two LUT-5 with 5 common inputs, etc. Formally we have the two LUT-5s with outputs $f_2(x_0, \dots, x_4), f_3(x_0, \dots, x_4)$ (see Fig. 1) but the latter output is replaced with

$$f_1(x_0, \dots, x_5) = \begin{cases} f_3(x_0, \dots, x_4) & \text{if } x_5 \\ f_2(x_0, \dots, x_4) & \text{otherwise} \end{cases}. \tag{5}$$

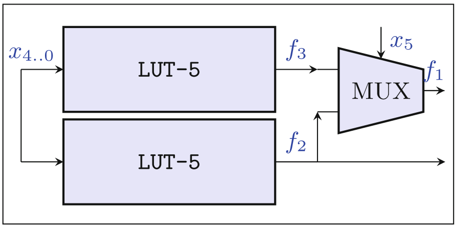

Fig. 1. LUT6-2 in Xilinx Ultrascale Configurable Logic Blocks (CLBs).

This physical realization turns out to be very convenient for logic circuits like LZCs. For clarity, notation LUT$-k$ where $2 \leq k \leq 6$ represents a logical LUT, while $LUT6-2$ represents the physical element. The second feature of modern Xilinx FPGAs is the existence of two extra multiplexing units in all of the slices called a *MUXF7* and *MUXF8* as shown in Fig. 2. The figure shows its name usage context but these elements can also function as a generic multiplexer of two LUTs or two MUXF7s). On the left, is the optimal CLB arrangement of a 7-bit multiplexer with the MUXF7 preceding the output, while on the right is the optimal CLB arrangement of an 8-bit multiplexer, from which these units receive their name. Inputs and outputs are indicated by arrows. These elements allow further multiplexing of the outputs of the LUT6-2s. Conceptually, *MUXF7* and *MUXF8* allow 2 LUT-6s to be turned into an 8 to 1 multiplexer or 16 to 1 multiplexer, respectively [3]. However, they need not be strictly used for this specific purpose. At this point we notice the presence of a *MUXF9* unit in the design, however, it is not inferred during the synthesis process and the direct utilization of this element might cause undesirable path delays (unless under specific circumstances), so we do not consider this element in this work. *MUXF7* and *MUXF8* are explicitly specified via the *MUXF_MAPPING* VHDL property, and can be converted to LUT-3s via the *-muxf_remap* option in the design optimization phase. (The 3-input *MUXF7* can be also considered as a 3-input LUT having a configuration space of size $2^3 = 8$. [15]) A limitation of the outlined design is that the LUT6-2 pairs in a slice connect to one of the two multiplexed inputs of these multiplexer units, but not to the selector signal (see x_6/x_7 on the left/right side of Fig. 2.). This limitation makes the *MUXF7* and *MUXF8* units furthermore only optimal in specific contexts, depending upon the routing intricacies. Xilinx provides the Vivado Design Suite [16] to perform synthesis and implementation for its various FPGA models. The synthesis translates the VHDL into an internal model format of LUT, MUXF7, MUXF8 and optionally LUTNM units. Post-synthesis, the optional *optimize design* stage is typically invoked to start the implementation process. It can for example, reduce logic which is unnecessarily cascaded. The *-remap* option (and its related *-aggressive_remap* and *-resynth_remap*) could have a significant effect on the final circuit, often adding additional LUTs which would reduce path delay. Nevertheless, this option for simple measurements can be disabled. The physical optimizations, power optimizations and routing remain to achieve an implementation whose bitstream can be utilized.

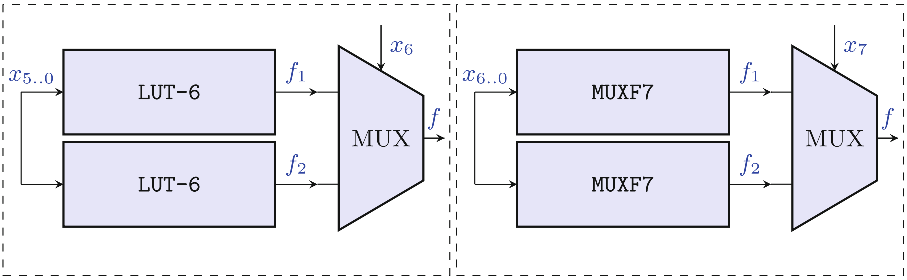

Fig. 2. MUXF7 and MUXF8 in Xilinx CLBs when used as a 7 and 8 bit multiplexer.

2 High-Level-Synthesis (HLS) Tools Background

For this project, rather than writing VHSIC Hardware Description Language (VHDL, where VHSIC is Very High-Speed Integrated Circuit), we used the MaxCompiler Data-Flow Engine (DFE) framework to provide a high-level Java description describing the calculations in a high-level extended variant of the Java programming language.

MaxCompiler uses a kernel design methodology, where each kernel is buffered by pulling input from a data stream and pushing output onto a data stream. It provides IP to integrate the PCIExpress subsystem for input and output seamlessly as data streams which can feed into the various kernels. MaxCompiler ultimately invokes Vivado to synthesize and implement the design. MaxCompiler has been used for large-scale research involving trigger algorithms [11], Complex FIR Filters [14] and even stencil computing [5] and most recently for quantum computer simulation [9]. But the idea of creating state-of-the-art circuits at a small scale has not been investigated. Although MaxCompiler supports at a kernel-level IP integrated solutions, it does not offer such functionality at a circuit level within a kernel.

The main feature needed to implementation efficient designs is part of the aptly named optimization library in MaxCompiler. The pipelining register disable and pipelining register enable features which effect a stack of states describing how operations pipeline which are constructed during any Java code in the current scope. Manually pipelining a signal s is easily done via explicit pipelining. By default, high level tools often pipeline every logic at the operator level which is not efficient for constructing efficient LUT-mapped circuitry. For example consider a bitwise AND: A & B will be stored in a register by default. This is equivalent to the behavior in no pipelining mode of pipeine(A & B).

There is furthermore far less research into state-of-the-art circuits which are constructed via high-level synthesis tools as opposed to those based upon handcrafted VHDL.

3 Description of the LZC Design and Implementation

There are two typical design patterns when synthesizing logic circuits, one is the cascade, sequential arrangement, and the other is a tree, or parallel arrangement. The timing properties, and amount of resources are very much dependent upon this choice, and there is usually a point at which performance versus resources becomes a critical deciding factor. The interesting case, is when one aspect can be improved without worsening the other.

Several recent LZC designs have been published. The first introduced a leading-zero anticipation framework to handle carry-lookahead operations [4]. This was improved upon to use complex gates to improve the speed [6]. There was also a design which focused on modularity rather than efficiency [7]. Vivado High-Level Synthesis itself also provides a default implementation based on the C instrinic *__builtin_clz* which are generic but not optimal, and further leave the all-zero case as undefined [16].

We offer an improvement over the method which Zahir, et al. introduced [17]. Their method used an LZC-8 consisting of 3 LUTs cascading into a LUT6-2 as a primitive for larger LZC units. By removing the cascaded LUT structure of their 8-bit LZC primitive which although necessary for a 7 or 8 bit LZC, turns out to be logic expandable and reducible into the 16-bit layer, the path delay can be significantly improved. This in turn allows for meeting timing at higher frequencies. Furthermore, by careful use of LUT combining, rather than a 3-level cascade with 10 LUTs, we achieve 6 LUTs cascading into 4 LUTs, preserving the LUT usage.

Furthermore we generalize the solution to any bit size, and not just even powers of two. Floating point implementations for various mantissa sizes can be further optimized to its precise LZC bit-optimized variant. Even at common sizes of IEEE-754 single and double precision simple implementations require LZC of 26 and 55 while an extended precision x86 long double requires 68, depending on implementation details regarding rounding strategies. In an FPGA, there is no native or built-in FPU, so this functionality cannot be replicated via the method mentioned in the introduction.

Now we note several special base cases of LZCs: (i) LZC for 1-bit is trivial and requires no LUTs just the constants, the original signal or its inversion (which count as 1 LUT only if a final signal). (ii) The 2-bit variant is similarly trivial requiring a 2-LUT but these signals can be merely propagated to the next level if part of a bigger LZC. (iii) For 3 or 4 bits, it will require 2 LUTs. (iv) For 5 or 6 bits, 3 LUTs. (v) And finally for 7 or 8 bits, 4 LUTs like in the prior method. (vi) Various occurrences where the formulae simplify down to a single signal, or even two or in some cases 3 signals may allow merging of signals into later layers, but such fine-tuned optimization should be reserved for Intellectual Property (IP)-level solutions, and we do not concern ourselves with this.

For convenience, we label a k-bit LZC unit ($k >= 2$) with LZC$-k$. The primary design issue in creating larger LZC units is to solve the odd/even parity computation of the result in a LZC-15/16 unit (i.e LZC with 15 and 16 bits) whose inner intermediate values we refer to as leading-parity-one/two/three

(*LP1*, *LP2*, *LP3*) along with one to compute the all-zero indicator V (labeled by *LP4*) and whose truth table is defined in Table 1. X6-X1 represent the 6 most significant bits in little-Endian order. Note that *LP3* is not present for LZC-4 or less while LP2 is not present for LZC-2 or less. The input value of X represents "don't care" or any value. These values are cascaded into a second stage of LUTs provided in later equations. We also provide the formulae which are straight-forward derivations from Eqs. (1) and (3), and we have left them in their form which uses the minimal possible Boolean operations:

$$LP1 = \overline{X_6} \wedge \left(X_5 \vee \left(\overline{X_4} \wedge (X_3 \vee \overline{X_2})\right)\right) \tag{6}$$

$$LP2 = \overline{X_6} \wedge \overline{X_5} \wedge \left(X_4 \vee X_3 \vee \left(\overline{X_2} \wedge \overline{X_1}\right)\right) \tag{7}$$

$$LP3 = \overline{X_6} \wedge \overline{X_5} \wedge \overline{X_4} \wedge \overline{X_3} \tag{8}$$

$$LP4 = \overline{X_6} \wedge \overline{X_5} \wedge \overline{X_4} \wedge \overline{X_3} \wedge \overline{X_2} \wedge \overline{X_1} \tag{9}$$

A general 16-bit function takes at least a two-stage LUT cascade on a LUT-6-based architecture, which turns to be both sufficient and efficient in the case of LZC-15/16, due to the regularity and evenness of the calculation. A possible realization of a LZC-15/16 is to cascade two LZC-8 sub-units of Ref. [17]. Determining the parity with these LZC-8 sub-units would require 4 intermediate parity signals, and 3 all-zero signals. Since this approach would not properly fit the Ultrascale CLB architecture, the solution is to introduce a specialized strategy by utilizing a special signal splitting along groups of 6, 6 and 5 bit-slices of the input X for *LP1* and the all-zero checks (see Fig. 3 for details). In 3, combined LUT6s are colored dark. The high (H) 6 bits of the input X are elaborated by the upper half of the design, while the LUTs in the bottom row operates on 5-bit slices of the input X denoted by low (L) and lower low (LL) labels. The design can be adopted for LZC-9 up to LZC-14 implementations with signal reductions as described in the text. Note that LP1$_{LL}$ is computed by the LP1 truth Table 1 by setting $X_6 = 0$. The *LP1* computation need not consider the least significant bits (X_{11}, X_5, X_1) in the presence of the all-zero signal (reducing the bit-slices effectively to 5, 5 and 4 bits). This observation reduces the final computation of *LP1* to the combinations of 5 signals compared to 7 signals if maintaining the approach of Ref. [17]. To minimize the path delay for LZC-15/16, we use an LZC-8-Intermediate, i.e. a non-cascaded design of 2 LZC-8 units. The concept of the improved circuit is outlined in Fig. 3. In contrast to the LZC-8-Intermediate used by 15 and 16-bit inputs, the 9 to 14-bit input X is partitioned into two 8-bit segments (high and low) and transferred into LZC-8-High and LZC-8-Low components. LZC-8-High refer to the top half of Fig. 3 while LZC-8-Low indicate the high unit duplicated with $X_{8..1}$ substituted for $X_{16..9}$, having a symmetrical structure in this case. In order to fit the Xilinx CLB architecture, the LZC-8-High and LZC-8-Low units are utilized in the following way: 6 signals are received from LZC-8-High ($LP1_H$, $LP2_H$, $LP3_H$, $LP4_H$, X_9, X_{10} in Fig. 3) and from Low units ($LP1_H$, $LP2_H$, $LP3_H$, $LP4_H$, X_9, X_{10} in Fig. 3). In case the final LZC unit is less than 2, 4 or 6 bit-wide (corresponding to LZC-9 up to LZC-14), the lower part of the design returns 1,

Table 1. Boolean Logic Mappings used by LZC-8-Intermediate results.

X6	X5	X4	X3	X2	X1	LP3	LP2	LP1	LP4
1	X	X	X	X	X	0	0	0	0
0	1	X	X	X	X	0	0	1	0
0	0	1	X	X	X	0	1	0	0
0	0	0	1	X	X	0	1	1	0
0	0	0	0	1	X	1	0	0	0
0	0	0	0	0	1	1	0	1	0
0	0	0	0	0	0	1	1	1	1

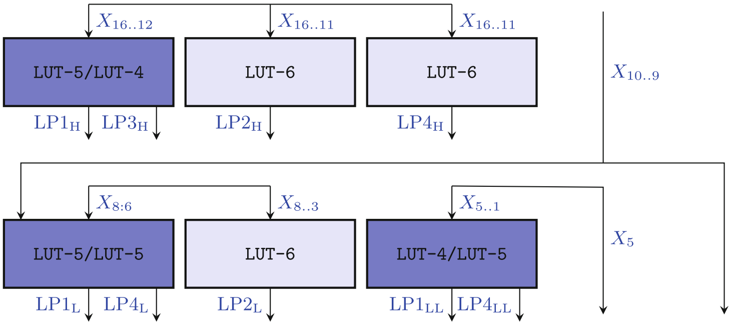

Fig. 3. Fully Parallel LZC-8-Intermediate circuit for LZC-15/16.

2 or 3 signals, respectively. Now we switch to a more generic notation, as after the base case, the units can be combined recursively to form larger units. In general, for LZC-k where $k = 8q$ or $k = 8q - 1$ ($q \geq 2$, $q \in \mathcal{Z}$), we have the following equations which define temporary logic values (signals) V_H, Z_{0_H}, Z_{1_H}, Z_{2_H}, Z_{1_L}, Z_{2_L} to be used after (where the equations for $V_L, Z_{2_L}, Z_{1_L}, Z_{0_L}$ are equivalent to the high equations for LZC-9 up to LZC-14):

$$\begin{aligned} V_H &= \mathrm{LP4}_H \wedge \overline{X_{10}} \wedge \overline{X_9}, \\ Z_{2_H} &= \mathrm{LP3}_H, \\ Z_{1_H} &= \mathrm{LP2}_H, \\ Z_{0_H} &= \left(\mathrm{LP1}_H \wedge \overline{\mathrm{LP4}_H}\right) \vee \left(\mathrm{LP4}_H \wedge \overline{X_{10}}\right), \\ Z_{2_L} &= \mathrm{LP4}_L \wedge \overline{X_5}. \\ Z_{1_L} &= \mathrm{LP2}_L. \end{aligned} \tag{10}$$

Observe that intermediary V_H specifically has 3 signals involved in its computation, which will reduce possibility for LUT combining on the second stage, but we have compensated for this with an efficient first stage. It will be combined with at most 2 or 3 further signals to compute the LZC-16 unit signals. This combined usage can be represented by on-chip multiplexers. V_L and Z_{0_H}, Z_{0_L}

are not needed in further processing as will be described shortly. However, Z_{0_H} is used in each LZC ranging from 9-bits up to 14-bits but not necessary in the LZC-15/16.

Generalizing this scheme to arbitrary LZC sizes where $8 \leq (k \mod 16) \leq 14$, one requires a simple fallback strategy where if $\mathrm{LP3}_L$ is not present, $\mathrm{LP2}_L$ is used. If $\mathrm{LP2}_L$ is not present then $\mathrm{LP1}_L$ is used while both $\mathrm{LP1}_L$ and $\mathrm{LP4}_L$ are always present. When $1 \leq (k \mod 16) \leq 7)$ the same fallback strategy is used for $\mathrm{LP3}_H$ and $\mathrm{LP2}_H$, and X_{10} and X_9 can be removed or set to zero if they are not present. Furthermore, in this uneven case when $k \mod 16 < 8$, we can assume when no low pair is present that:

$$V_L = Z_{2_L} = Z_{1_L} = Z_{0_L} = 1\ . \tag{11}$$

The correct result then just makes substitutions of the prior defined intermediate values into the standard LZC equations, and this is the point at which the cascade occurs. It is clear that all signals are functions of at most 6 Boolean signals:

$$V = V_H \wedge V_L \tag{12}$$

$$Z_3 = V_H \tag{13}$$

$$Z_2 = \left(\overline{V_H} \wedge Z_{2_H}\right) \vee \left(V_H \wedge Z_{2_L}\right) \tag{14}$$

$$Z_1 = \left(\overline{V_H} \wedge Z_{1_H}\right) \vee \left(V_H \wedge Z_{1_L}\right) \tag{15}$$

$$Z_0 = \left(\overline{V_H} \wedge Z_{0_H}\right) \vee \left(V_H \wedge Z_{0_L}\right) \tag{16}$$

For LZC-15 and LZC-16, the LUT reduction modifications require the following substitutions defining signals V and Z_0:

$$V = \mathrm{LP4}_H \wedge \mathrm{LP4}_L \wedge \mathrm{LP4}_{LL} \tag{17}$$

$$Z_0 = \mathrm{LP4}_H \wedge ((\mathrm{LP4}_L \wedge \mathrm{LP1}_{LL}) \vee \left(\overline{\mathrm{LP4}_L} \wedge \mathrm{LP1}_L\right)) \vee \left(\overline{\mathrm{LP4}_H} \wedge \mathrm{LP1}_H\right) \tag{18}$$

The multiplexer usage possibility is based on the simple relationship e.g.:

$$\left(\overline{V_H} \wedge Z_{0_H}\right) \vee \left(V_H \wedge Z_{0_L}\right) == \begin{cases} Z_{0_L} & \text{if } V_H \\ Z_{0_H} & \text{otherwise} \end{cases} \tag{19}$$

(while programmers might be more familiar with $V_H ? Z_{0_L} : Z_{0_H}$).

The same technique as the first set of equations is then generalized and repeated for the larger building blocks from LZC $-k$ with $k > 16$ and beyond, including inferring the low-signals as all true when there is an odd number of signal groups leaving one without a pairing. Since signals created by Z_1 and Z_2 as well as Z_3 and V are candidates for LUT combining into LUT6-2 units or alternatively into MUXF7 and MUXF8 units, the latter stages also yield an area minimized solution. The final result can be returned as

$$(V, C) = \left(V, \mathop{\Big|\Big|}_{k=\log_2 n-1}^{0} Z_k\right). \tag{20}$$

Direct propagation of 3 signals from the LZC-8 stage to the LZC-16 stage creates an additional constraint for the routing part of the implementation. To address this, we can also allow Vivado synthesis flexibility in optimizing the logic as it sees fit. Since synthesis strategies for performance, area and power exist separately, this is preferred approach. Otherwise we can use VHDL *KEEP* directives on LUT output signals to prevent any sort of combining at synthesis time. This is as opposed to the VHDL *DONT_TOUCH* attribute which is similar but also applied during implementation having the unfortunate side-effect of preventing LUTNM combining during placement. However, the optimize design step may no longer preserve the original plan without this attribute.

So the 16-bit LZC achieves the smaller path delay without any additional LUT cost. At most $\lceil \frac{k}{16} \rceil$ additional LUTs are used while the theoretical path delay is reduced by $\lceil \log_2 \frac{k}{8} \rceil / (\lceil \log_2 \frac{k}{8} \rceil + 1)$ percent. We also indicate that this design is suitable for older architectures like Virtex 7 or any other which support the LUT6-2 units.

Finally, we also present a MUXF7/MUXF8 specific IP-level solution for an LZC-8 which does not cascade LUTs. Although the circuit would not infer from high-level tools, its existence is noteworthy. Namely *LP2*, *LP3* are computed by Eqs. (7) and (8). On the other hand, *LP1*, *LP4* and V are computed by:

$$\begin{aligned} LP4 &= \begin{cases} 0 & \text{if } X_7 \\ \overline{X_6} \wedge \overline{X_5} \wedge \overline{X_4} \wedge \overline{X_3} \wedge \overline{X_2} \wedge \overline{X_1} & \text{otherwise} \end{cases}, \\ V &= \begin{cases} 0 & \text{if } X_8 \\ LP4 & \text{otherwise} \end{cases}, \\ LP1 &= \begin{cases} 0 & \text{if } X_8 \\ X_7 \vee LP1(X_{6..1}) & \text{otherwise} \end{cases} \end{aligned} \tag{21}$$

This straight-forward method uses 4 LUTs, and for *LP1* an additional single MUXF7, while for V, both a MUXF7 and a MUXF8. All the signals enter and leave the slice only one time, providing minimal routing delay, and only the delay of the MUXF7 and MUXF8 units themselves.

3.1 Demonstration of the Design on a Small-Sized Example

As an example to show the main computational stages in the outlined procedure, consider the LZC-16 of the number 2 which is “00000000 . 00000010b”. It is clear that $(V, C) = (0, 14)$. Computing the LZC-8-Intermediate values shows that $\mathrm{LP1}_H, \mathrm{LP2}_H, \mathrm{LP3}_H, \mathrm{LP4}_H, \mathrm{LP1}_L, \mathrm{LP2}_L, \mathrm{LP3}_L, \mathrm{LP4}_L$ will be 1 while $\mathrm{LP1}_{LL}, LP4_{LL}$ are both 0. This implies that V, Z_1, Z_2, Z_3 are 1 while Z_0 is 0. We can calculate C from concatenated binaries Z_i as: $C = 2^3 Z_3 + 2^2 Z_2 + 2^1 Z_1 + 2^0 Z_0 = 14$, as expected.

If we used an LZC-8-High and LZC-8-Low in this example, instead of LZC-8-Intermediate, then the values turn out to be the same, except X_1, X_2, X_8, X_9 along with $\mathrm{LP1}_H, \mathrm{LP4}_H, \mathrm{LP1}_L$ are needed to compute Z_0 since $\mathrm{LP1}_{LL}, LP4_{LL}$ are not present.

4 Results and Discussion

For our experiments and design we target the Ultrascale+ architecture and specifically Alveo U250 FPGAs. We targeted a 650MHz clock frequency, the highest that MaxCompiler can synthesize with the Mixed-Mode Clock Manager (MMCM) oscillators/frequency synthesizers on the FPGA, from the "freerun" clock which enters the chip through an IO port at 300MHz. We only use the Super Logic Region (SLR) of SLR1 as the PCIExpress ports are bound there by MaxCompiler (though the Xilinx floorplan does indicate SLR0 as the true point of entry). Our MaxCompiler version is 2021.1 which works alongside Vivado 2020.1. The Vivado implementation was based upon the versatile "Performance_ExplorePostRoutePhysOpt" strategy. We utilized a global clock buffer for the clock enable (CE) signal of the kernels which uses a BUFGCE unit on the Ultrascale device.

Comparing to high-level MaxCompiler provided machinery was determined to be inappropriate as it would require a combination of a leading one detector (via the simple two's complement property leading1detect(x)=-x&x where here a bitwise AND is used) and a one-hot decoder which generates an $O(n^2)$ VHDL algorithm, giving high area and power, and degraded performance due to presence of addition (as -x=˜ x+1), fanout and congestion. However, the one-hot decoder did not allow disabling pipeline register insertion, so we were unable to make a meaningful comparison without modifying the VHDL and we thereby opted not to compare it. However the combination of a leading-one-detector and a one-hot decoder as an alternative means of computing LZC, is convenient and perfectly fine in some scenarios.

We chose an synthesis strategy optimized for performance based on Vivado's "Flow_PerfOptimized_high", and hence will have a resulting non-optimal LUT count, but optimal path delay and ability to synthesize at higher frequencies. For purpose of comparison, we used sizes the prior algorithm supported rather than the minor differences of the optimal generalized size variants.

Our results will appear different from those of the referenced paper because they targeted Virtex 7 with a 28nm process. While here we target Ultrascale+ with a 16nm process. There is a common practical rule that assumes roughly 3 levels of cascaded LUTs can meet timing regardless of the frequency in most situations. This has been applied for barrel shifters and other various circuits that require cascading for efficiency. At 16nm, several more layers of cascading seem to be possible, but it will start to place difficult timing constraints on the signals involved and influence their allowed spatial arrangements. So for an actual implementation, for very large LZC implementations (such as LZC-68) some adjustment of lower frequency or inserting pipeline registers into the algorithm after some number of levels becomes strictly necessary.

We collected the results using a hand designed Tool Command Language (TCL) script which integrates with Vivado. The input stream registers are easily found via regular expressions and by traversing output pins, nets and cells, the circuit can be recursively discovered in a depth-first search (DFS) manner. The specific slices and Basic ELements (BELs) are tracked during this traversal. Then the “get_timing_paths” command measures the worst timing path amongst the starting and terminating elements of the realized circuit (from register output pin to register input pin). Power was measured hierarchically via “report_power”. When a build succeeds, this script is automatically run based on the saved checkpoint preserved by setting MaxCompiler not to clean the build folder (“clean_build_directory” set to “false”).

The explanation for the LUT counts being high requires a deeper understanding of Vivado synthesis. Although it can merge or split apart Boolean functions, when it sees a function of more than 6 arguments, how it choose to cascade or plan them is according to its own algorithms. At the lowest-level, VHDL can describe individual LUTs, while at the highest level it describes mere logic which infers some sort of LUT structure. The Vivado design methodology makes it reasonable to trust their automatic synthesis choices as long as the design is constrained in appropriate areas. Our area of interest is primarily performance, though optimization for power is also possible. We do not expect too much beyond some reasonable heuristics as the circuit satisfiability problem (circuit-SAT) is NP-complete, only here we have 6-input gates rather than binary gates. Tools are beginning to incorporate machine learning (ML) for improved heuristic inference.

Table 2. Performance Results for various LZC sizes.

LZC	LUTs(LUTNMs	Slices	Power (mW)	Delay (ns)	Freq. (MHz)
bitwidth	/MUXF7/MUXF8)				
8 new/old [17]	4 (1)	2	10	0.808	600
16 old [17]	12 (1)	3	13	1.016	650
32 old [17]	29 (1)	7	11	1.226	650
64 old [17]	58 (2)	14	11	1.69	470
16 new	11 (3)	3	10	0.952	500
32 new	27 (5)	10	13	1.142	600
64 new	67 (1)	16	15	1.429	650
8	5 (1)	2	13	0.772	610
16	10 (4)	5	10	0.988	510
32	27 (0)	7	12	1.052	650
64	56 (0/8/0)	15	20	1.363	650

We note several things about the results in Table 2. The number of Logical LUTs introduced is LUTs plus LUTNMs. The data was gathered by compiling 2 independent but identical circuits so LUTs and slices have been ceiling divided by 2. The Zahir, et al. [17] scheme is labeled “old”, the proposed scheme synthesized with *KEEP* to preserve LUT output signals is labeled “new” while the unlabeled does not use this attribute. We refer interested readers to [17] for comparison to the other algorithms from [4,6,7,16]. First, the number of slices for two identical circuits can have overlap, but slices are not considered a useful modern metric in large scale designs as the tools are not directly optimizing for minimal slices. The power measurement in Vivado is likely inaccurate and not very detailed for specific circuits. We therefore measured the power to the whole MaxCompiler kernel core in milli-Watts, the finest granularity and resolution offered by the tool. The kernel has some registers, counters and control mechanisms which make this an impure result, however it is a fixed cost across all kernels of identical input/output sizes.

The results are all for builds at 650 MHz which although not the $f_{\max}$ of 725 MHz., is the highest frequency at which MaxCompiler can configure an MMCM unit. Such excellent performance shows that the LZC is unlikely to be a limiting factor in the designs which utilize it at least through to 64 bits. We don't consider overall efficiency metrics such as the power-delay-area product (PDAP), as our focus was on highest build speed, and path delay while other metrics requires uniform build speeds or consideration of frequency weighting. The frequency and power have a simple linear relationship.

Furthermore, at very high frequencies, deeper circuits can in some cases perform better as the path delay effects the setup (which balances the clock skew against the path delay, clock uncertainty and setup time) as well as the hold (balancing path delay against clock skew, uncertainty and hold time) slacks for the registers which ultimately latch the result signals. For example at a clock speed of 650MHz, the clock period is $\frac{10^3}{650\text{MHz}} = 1.538$ nanoseconds (while Vivado timing scores are reported in picoseconds), which although an upper bound on path delay to achieve a build at this frequency, needs to account for the setup and hold slack in full.

5 Conclusion

The shortcomings of the proposed technique is less modularity, more complicated logic, and that certain non-power of two LZC sizes may have further unique optimizations which require complicated generation algorithms or on a case-by-case basis analysis to determine or achieve. The ideas presented are enough to find such simplifications. However, if willing to implement a more complicated and general framework, our design allows higher frequency builds, and savings in LUT and routing resources and more optimal IP. Future works can use the ideas presented to improve circuits beyond the trivially equivalent leading/trailing-one-counter, but counting bits set (sometimes called *popcount*), checking for powers of two, or rounding up to the nearest power of two, etc.

We have given a survey of the synthesis features for a specific modern tool and architectures where it comes to optimizing a small but important circuit as it relates to HPC. It shows that a generic design with a reasonable signal layout will synthesize by sophisticated tools with success in likely any optimization scenario when the logic is minimally constrained. However, due to the various complications of the physicality and details of setup and hold, trying multiple options may be necessary as the minimal path delay is not always the build that ultimately succeeds.

We furthermore carefully provided details towards a research methodology for designing small-scale circuits with HLS tools, understanding the ways of constraining the underlying build tool, as well as measuring and collecting data points. This methodology is geared specifically toward achieving high frequency and minimal path delay builds.

Although the synthesis and implementation process is complicated and depends on a broad set of constraints and target goals from performance to power to resource usage and chip area, the design here should from all perspectives have better versatility and be applicable for maximizing performance and/or minimizing area and power. The fact that LZC-16 and higher circuits builds at the highest synthesize-able frequency of 650MHz from high-level language tools, is indicative of both the flexibility of modern high-level synthesis tools like MaxCompiler, and the diverse features of modern FPGA architectures like Ultrascale.

Acknowledgements. This research was supported by the Ministry of Culture and Innovation and the National Research, Development and Innovation Office within the Quantum Information National Laboratory of Hungary (Grant No. 2022-2.1.1-NL-2022-00004), by the ÚNKP-22-5 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund. RP. acknowledge support from the Hungarian Academy of Sciences through the Bolyai János Stipendium (BO/00571/22/11) as well.

References

1. Anderson, J.H., Wang, Q.: Area-efficient FPGA logic elements: architecture and synthesis. In: 16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011), pp. 369–375 (2011). https://doi.org/10.1109/ASPDAC.2011.5722215
2. Anderson, S.E.: Bit twiddling hacks (2005). http://graphics.stanford.edu/seander/bithacks.html
3. Chapman, K.: Multiplexer design techniques for Datapath performance with minimized routing resources. Xilinx All Programmable **1**, 1–32 (2014)
4. Dimitrakopoulos, G., Galanopoulos, K., Mavrokefalidis, C., Nikolos, D.: Low-power leading-zero counting and anticipation logic for high-speed floating point units. IEEE Trans. Very Large Scale Integr. Syst. **16**(7), 837–850 (2008). https://doi.org/10.1109/TVLSI.2008.2000458
5. Dohi, K., Okina, K., Soejima, R., Shibata, Y., Oguri, K.: Performance modeling of stencil computing on a stream-based FPGA accelerator for efficient design space exploration. IEICE Trans. Inf. Syst. **98**(2), 298–308 (2015)
6. Miao, J., Li, S.: A design for high speed leading-zero counter. In: 2017 IEEE International Symposium on Consumer Electronics (ISCE), pp. 22–23 (2017). https://doi.org/10.1109/ISCE.2017.8355536
7. Milenković, N.Z., Stanković, V.V., Milić, M.L.: Modular design of fast leading zeros counting circuit. J. Electr. Eng. **66**(6), 329–333 (2015). https://doi.org/10.2478/jee-2015-0054

8. Nadjia, A., Mohamed, A.: Efficient implementation of AES S-box in LUT-6 FPGAs. In: 2015 4th International Conference on Electrical Engineering (ICEE), pp. 1–4 (2015). https://doi.org/10.1109/INTEE.2015.7416679
9. Rakyta, P., Morse, G., Nádori, J., Majnay-Takács, Z., Mencer, O., Zimborás, Z.: Highly optimized quantum circuits synthesized via data-flow engines (2022). arXiv preprint arXiv:2211.07685
10. Srivastava, P., Chung, E., Ozana, S.: Asynchronous floating-point adders and communication protocols: a survey. Electronics **9**(10), 1687 (2020). https://doi.org/10.3390/electronics9101687, https://www.mdpi.com/2079-9292/9/10/1687
11. Summers, S., Rose, A., Sanders, P.: Using MaxCompiler for the high level synthesis of trigger algorithms. J. Instrumentation **12**(02), C02015 (2017). https://doi.org/10.1088/1748-0221/12/02/C02015
12. Suzuki, H., Morinaka, H., Makino, H., Nakase, Y., Mashiko, K., Sumi, T.: Leading-zero anticipatory logic for high-speed floating point addition. IEEE J. Solid-State Circ. **31**(8), 1157–1164 (1996). https://doi.org/10.1109/4.508263
13. Walters, E.G.: Array multipliers for high throughput in xilinx FPGAs with 6-input LUTs. Computers **5**(4), 20 (2016). https://doi.org/10.3390/computers5040020, https://www.mdpi.com/2073-431X/5/4/20
14. Wang, H., Gante, J., Zhang, M., Falcão, G., Sousa, L., Sinnen, O.: High-level designs of complex FIR filters on FPGAs for the SKA. In: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 797–804 (2016). https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0115
15. Xilinx, I.: Ultrascale architecture configurable logic block user guide (2023). https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb
16. Xilinx, I.: Vivado design suite user guide: High level synthesis UG902 (v2020.1) (2023). https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis
17. Zahir, A., Ullah, A., Reviriego, P., Hassnain, S.R.U.: Efficient leading zero count (LZC) implementations for xilinx FPGAs. IEEE Embed. Syst. Lett. **14**(1), 35–38 (2022). https://doi.org/10.1109/LES.2021.3101688
18. Zhang, H., Chen, D., Ko, S.B.: High performance and energy efficient single-precision and double-precision merged floating-point adder on FPGA. IET Comput. Digit. Tech. **12**(1), 20–29 (2018). https://doi.org/10.1049/iet-cdt.2016.0200, https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-cdt.2016.0200