Minimal Path Delay Leading Zero Counters on Xilinx FPGAs

Gregory Morse [1]
morse@inf.elte.hu

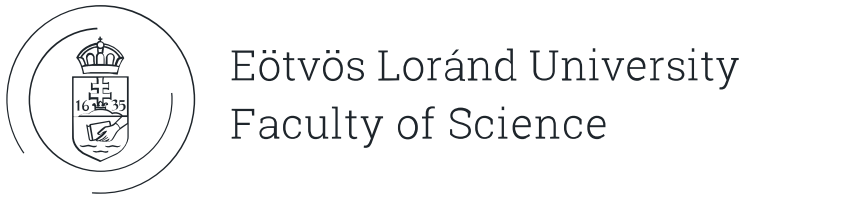
Tamás Kozsik [1]
kto@elte.hu

Péter Rakyta [2]
peter.rakyta@ttk.elte.hu

[1] Department of Programming Languages and Compilers,

[2] Department of Physics of Complex Systems, Eötvös Loránd

Project Collaboration



Introduction

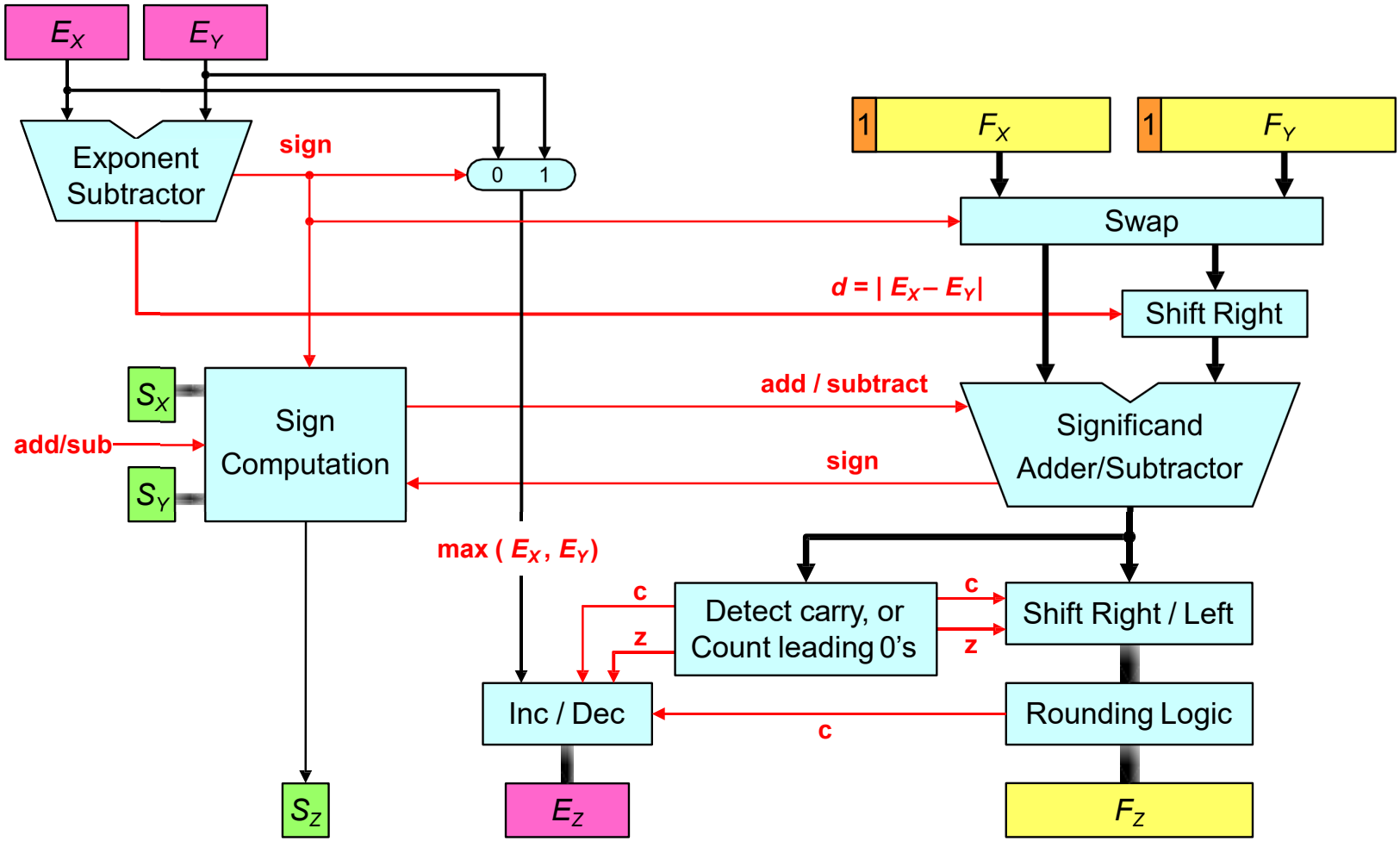
- Leading Zero Counters are necessary in IEEE floating point addition which is very resource-intensive over hundreds of instances
- Target Xilinx Series 7 and Ultrascale Architecture FPGAs
- Configurable Logic Block (CLB) allows efficient implementation
- Use of high-level MaxCompiler to generate efficient low-level circuits
- Generalize to any bit-size
- Prioritize high-performance, low-resources (not power)
- Reduce modularity to further minimize path delay over Zahir, et al. Efficient leading zero count (LZC) implementations for Xilinx FPGAs. IEEE Embedded Systems Letters 14(1), 35–38 (2022)

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

Floating point addition/subtraction motivation

- Not needed when adding same-signed values or subtracting opposite-signed values as the result always can be at least half or at most double, easy to check 3-bit positions
- When effective subtraction occurs, result can have any number of leading-zeros
- Traditional clever use of floating point units (FPUs) addition/subtraction unit has been using the normalization process post-subtraction with custom byte-packing
- 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 $m * 2^e$.
- here 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.

Example Schematic of a floating point adder



FPU LZC explicitly used in practice for integer $\log_2$

```
int v; // 32-bit integer to find the log base 2 of
int r; // result of log_2(v) goes here
union { unsigned int u[2]; double d; } t; // temp
t.u[__FLOAT_WORD_ORDER==LITTLE_ENDIAN] = 0x43300000;
t.u[__FLOAT_WORD_ORDER!=LITTLE_ENDIAN] = v;
t.d -= 4503599627370496.0;
r = (t.u[__FLOAT_WORD_ORDER==LITTLE_ENDIAN] >> 20) - 0x3FF;
```

- A C implementation from Bit Twiddling Hacks: Find the integer log base 2 of an integer with an 64-bit IEEE float
- Note: exponent bias of 11-bit exponent is $2^{10} - 1 = 1023$
- $0x433 = 1075 = 1023 + 52$ as the top 12 bits are sign + exponent
- $4503599627370496.0 = 2^{52}$ subtract the implicitly stored bit, invoking the LZC
- $0x3FF = 1023$ subtracted to un-bias the exponent
- All-zero case must be explicitly checked however
- `__builtin_clz` intrinsic more efficiently does this via bit-scan reverse (BSR) x86 assembly instruction

A binary logic viewpoint of LZC via recurrence relations

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

is the all-zero signal and

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

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

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. In the special case that X contains all zeros, then V and all bits of C are set to 1.

A look at the Xilinx CLB logic slice (SLICEL)

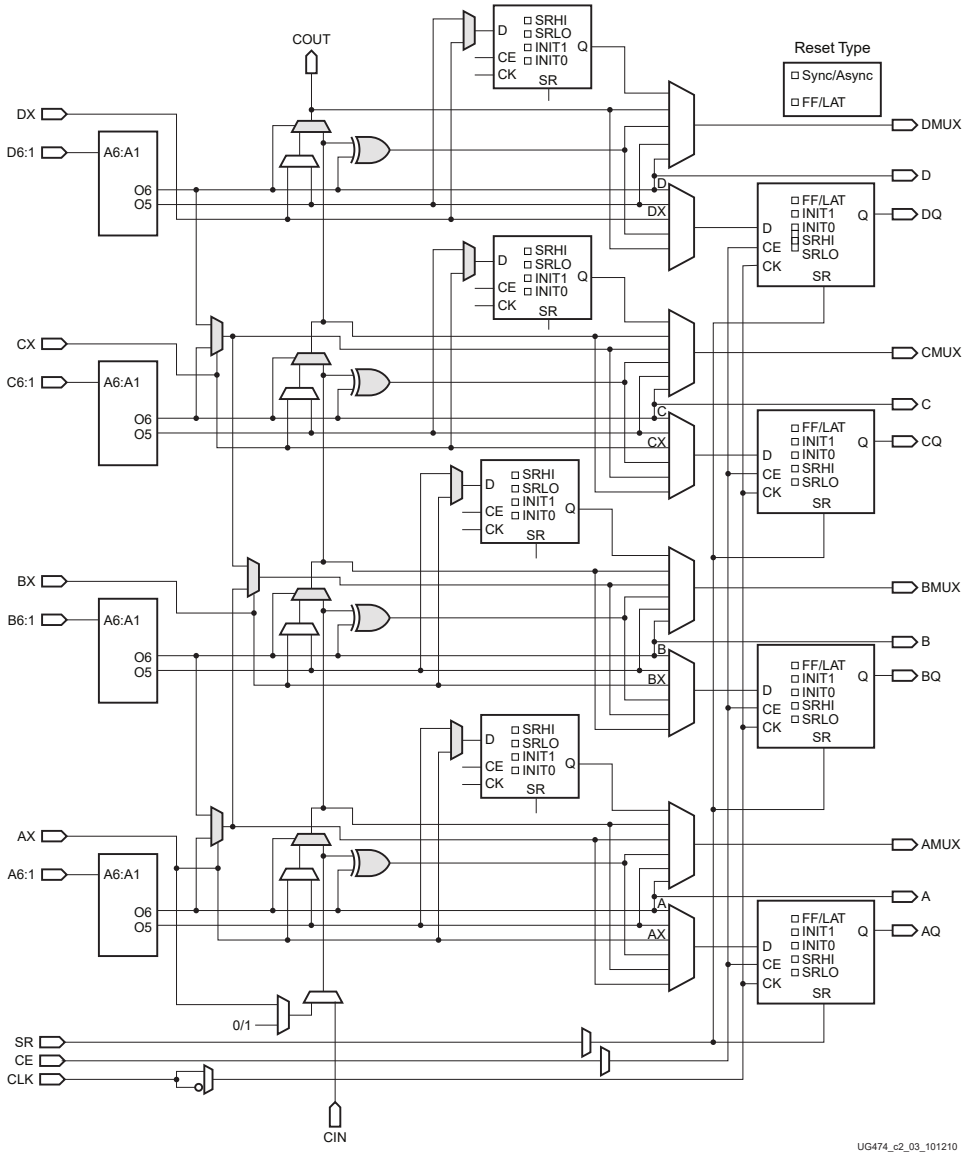


Figure 2-4: **Diagram of SLICEL**
www.xilinx.com

7 Series FPGAs CLB User Guide
UG474 (v1.8) September 27, 2016

- memory slice (SLICEM) are a superset for shift register look-up tables (SRLs)
- 4 Look-Up Tables (LUTs) on the far left
- 8 Flip-flops of D-type with Reset and Enable (FDRE) on the right
- Between the two is the vertical column carry in/out logic

A more detailed look at LUT6-2

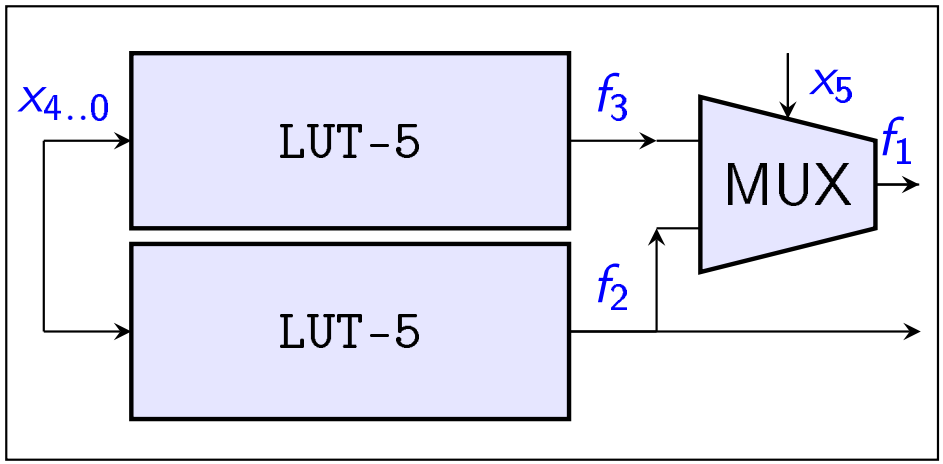


Figure: LUT6-2 in Xilinx Ultrascale Configurable Logic Blocks (CLBs).

$$f_1(x_0, \ldots, x_5) = \begin{cases} f_3(x_0, \ldots, x_4) & \text{if } x_5 \\ f_2(x_0, \ldots, x_4) & \text{otherwise} \end{cases}. \quad (4)$$

- In general, termed a LUTNM (where N=6, M=2)

A LUT-6 also provides a 4:1 multiplexer:

$$f(x_0, x_1, x_2, x_3, x_4, x_5) = \begin{cases} x_0 & \text{if } \overline{x_4} \wedge \overline{x_5} \\ x_1 & \text{if } \overline{x_4} \wedge x_5 \\ x_2 & \text{if } x_4 \wedge \overline{x_5} \\ x_3 & \text{otherwise} \end{cases} \quad (5)$$

A more detailed look at MUXF7/MUXF8

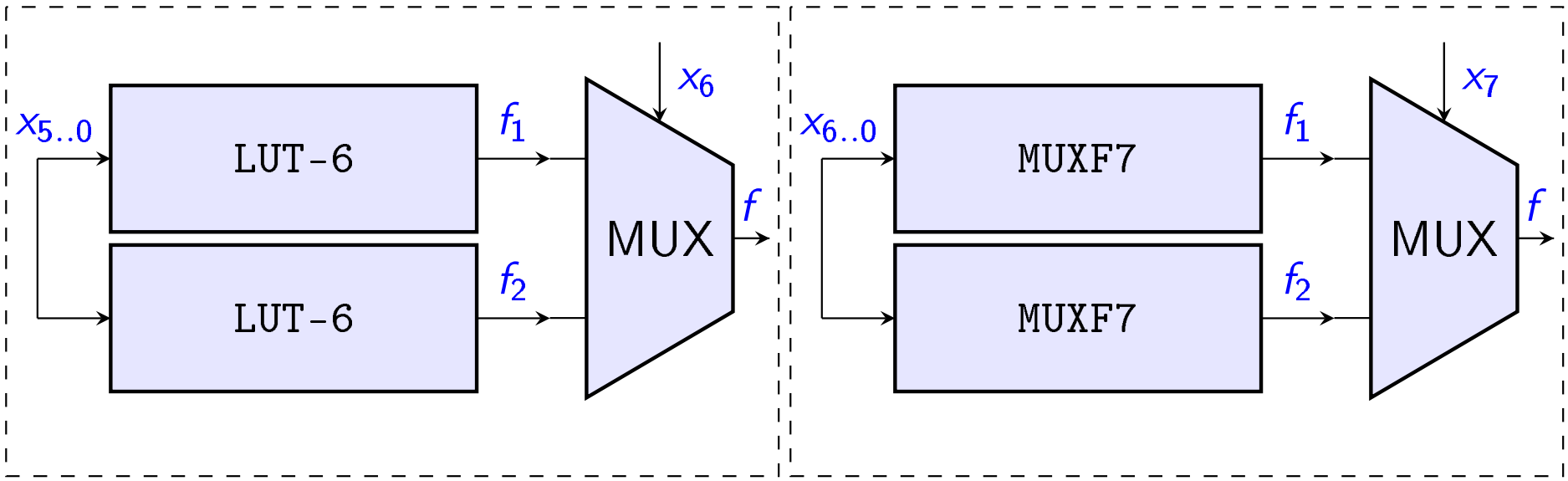


Figure: MUXF7 and MUXF8 in Xilinx CLBs when used as a 7 and 8 bit multiplexer.

- Conceptually, 2 or 4 LUT-6s to be part of an 8:1 or 16:1 multiplexer
- Ultrascale architecture also has added a MUXF9 which is not inferred in synthesis
- Explicitly specified via the *MUXF_MAPPING* VHDL property
- Can be converted to LUT-3s via the *-muxf_remap* option in the design optimization phase

High-Level Synthesis and Implementation

- Synthesis phase: map VHDL onto device logic elements while allowing constraints via Xilinx Design Constraints (XDC)
- Sub-processes from Xilinx Design Suite User Guide: Implementation
  - 1. **Opt Design**: Optimizes the logical design to make it easier to fit onto the target Xilinx device.
  - 2. Power Opt Design (optional): Optimizes design elements to reduce the power demands of the target Xilinx device.
  - 3. **Place Design**: Places the design onto the target Xilinx device and performs fanout replication to improve timing.
  - 4. Post-Place Power Opt Design (optional): Additional optimization to reduce power after placement.
  - 5. **Post-Place Phys Opt Design (optional)**: Optimizes logic and placement using estimated timing based on placement. Includes replication of high fanout drivers.
  - 6. **Route Design**: Routes the design onto the target Xilinx device.
  - 7. **Post-Route Phys Opt Design (optional)**: Optimizes logic, placement, and routing using actual routed delays.
  - 8. **Write Bitstream**: Generates a bitstream for Xilinx device configuration. Typically, bitstream generation follows implementation.

Controlling synthesis from MaxCompiler

- MaxCompiler Data-Flow Engine (DFE) framework provides a high-level Java description of the circuit
- Can disable/enable automatic pipeline registers via a stack of states
- Can manually pipeline a signal explicitly
- Can use custom Intellectual Property (IP) solutions at a kernel-level
- Through undocumented low-level customization, can attempt to control VHDL inference:
  - VHDL *KEEP* directives on LUT output signals to prevent any sort of combining at synthesis time
  - VHDL *DONT_TOUCH* attribute is similar but also applied during implementation having the unfortunate side-effect of preventing LUTNM combining during placement
- Xilinx Vivado also provides its own High-Level Synthesis (HLS) tool supporting C++ but is not as easy to use

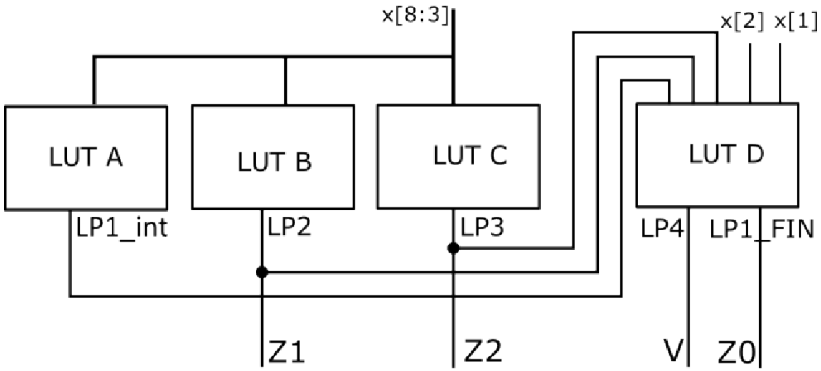
Zahir, et al. Design for LZC-8

X8	X7	X6	X5	X4	X3	LP3	LP2	LP1_int
1	X	X	X	X	X	0	0	0
0	1	X	X	X	X	0	0	1
0	0	1	X	X	X	0	1	0
0	0	0	1	X	X	0	1	1
0	0	0	0	1	X	1	0	0
0	0	0	0	0	1	1	0	1
0	0	0	0	0	0	1	1	0

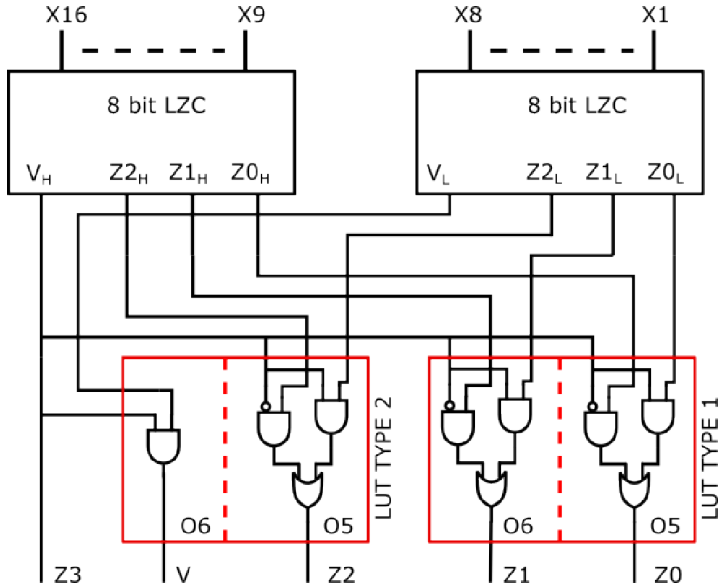
(a)

LP3	LP2	LP1_int	X2	X1	LP4	LP1_FIN
1	1	0	1	X	0	0
1	1	0	0	1	0	1
1	1	0	0	0	1	1
Rest of combinations					0	LP1_int

(b)



(c)



Our Design Formula for LZC-8

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

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

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

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

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

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

Design of LZC-15/16 with LZC-8-Intermediate versus LZC-8-High and LZC-8-Low

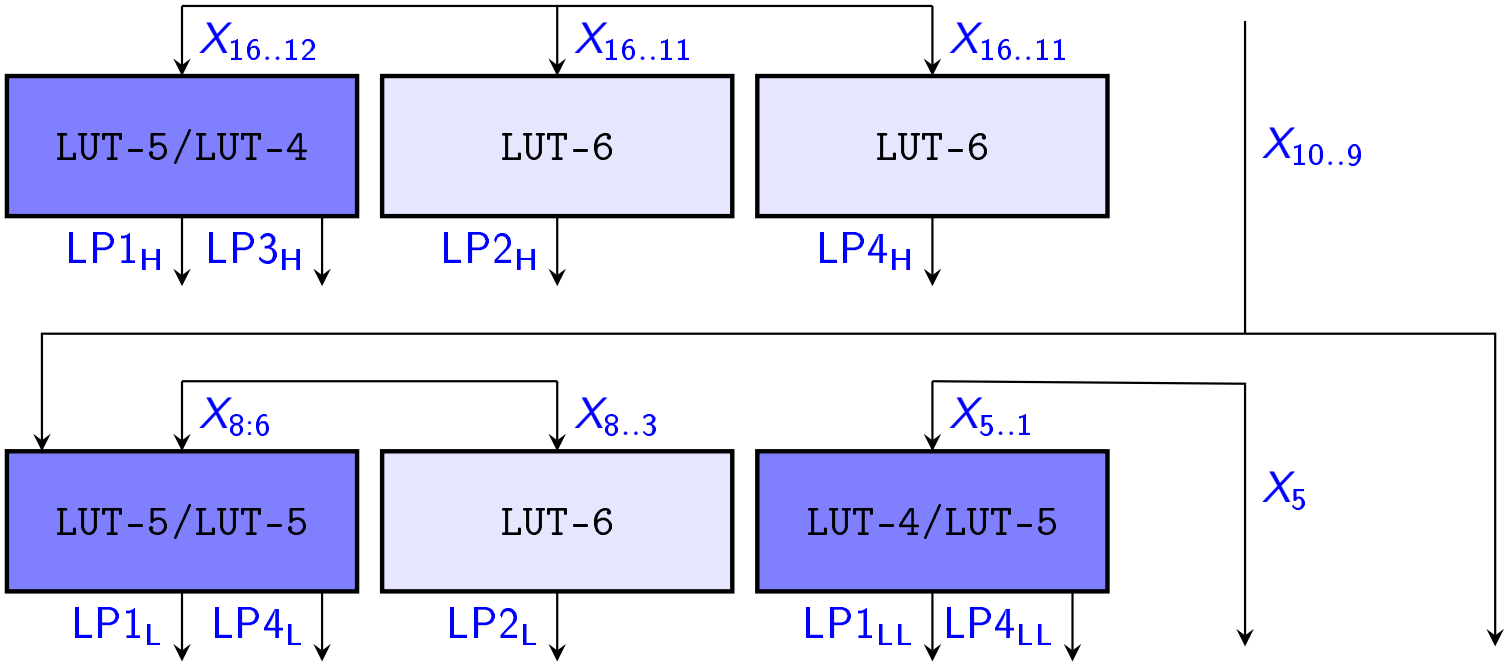


Figure: Fully Parallel LZC-8-Intermediate circuit for LZC-15/16.

- Combined LUT6s are colored dark.
- $LP1_{LL}$ is computed by the LP1 truth table by setting $X_6 = 0$

LZC-15/16 formulae

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

- 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, 2 or 3 signals, respectively.
- $V_L, Z_{2_L}, Z_{1_L}, Z_{0_L}$ are equivalent to the high equations for LZC-9 up to LZC-14
- V_L and Z_{0_H}, Z_{0_L} are not needed in further processing as will be described shortly.

Generalization to LZC of non-power of two sizes

- 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, 2 or 3 signals, respectively.
- When $k \mod 16 < 8$, we can assume when no low pair is present that:

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

- For $8 \leq (k \mod 16) \leq 14$, 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.

Modular framework to adapt 2 LZC-n to an LZC-$2n$, $n \geq 16$

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

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

$$Z_n = \left(\overline{V_H} \wedge Z_{n_H}\right) \vee \left(V_H \wedge Z_{n_L}\right), 0 \leq n \leq 2 \tag{14}$$

- Genearalizes to Z_n
  - 2 LZC16 to LZC32 introduces Z_4 and modifies Z_3
  - 2 LZC32 to LZC64 introduces Z_5 and modifies Z_4, Z_3

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

$$V = \mathrm{LP4}_H \wedge \mathrm{LP4}_L \wedge \mathrm{LP4}_{LL} \tag{15}$$

$$Z_0 = \mathrm{LP4}_H \wedge \left(\left(\mathrm{LP4}_L \wedge \mathrm{LP1}_{LL}\right) \vee \left(\overline{\mathrm{LP4}_L} \wedge \mathrm{LP1}_L\right)\right) \vee \left(\overline{\mathrm{LP4}_H} \wedge \mathrm{LP1}_H\right) \tag{16}$$

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

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

IP solution proposal

$$LP4 = \begin{cases} 0 & \text{if } X_7 \\ \overline{X_6} \wedge \overline{X_5} \wedge \overline{X_4} \wedge \overline{X_3} \wedge \overline{X_2} \wedge \overline{X_1} & \text{otherwise} \end{cases},$$

$$V = \begin{cases} 0 & \text{if } X_8 \\ LP4 & \text{otherwise} \end{cases}, \tag{17}$$

$$LP1 = \begin{cases} 0 & \text{if } X_8 \\ X_7 \vee LP1(X_{6..1}) & \text{otherwise} \end{cases}$$

- *LP2*, *LP3* are computed by Eqs. (7) and (8)
- Could not infer the circuit from MaxCompiler, synthesis inferrence uses heuristics without explicit instantiations or custom VHDL attributes.
- 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, beyond delay of the MUXF7 and MUXF8 units

Small-sized example

- 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
  - $\mathrm{LP1}_{LL}, LP4_{LL}$ 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.

Experiment Configuration

- Targetted Ultrascale+ architecture and specifically **Alveo U250** FPGAs
- **MaxCompiler version 2021.1** working alongside **Vivado 2020.1**
- Vivado implementation was based upon the versatile "Performance_ExplorePostRoutePhysOpt" strategy
- MaxCompiler provides no built-in method
- The closest built-in approach would be the 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.
- Synthesis strategy optimized for performance based on Vivado's "Flow_PerfOptimized_high"

Experimentation methodology

- LZC algorithms validated with comprehensive test cases using GNU multi-precision (MP) BigNum library via `mpz_sizeinbase(x, 2)`
- Ultrascale+ has a 16nm process (as opposed to Virtex 7 with a 28nm process)
- Custom Tool Command Language (TCL) script collected the results from Vivado
- The number of Logical LUTs introduced is LUTs plus LUTNMs.
- Data gathered by compiling 2 independent identical circuits - LUTs and slices ceiling divided by 2
- Measured the power to the whole MaxCompiler kernel core in milli-Watts, the finest granularity of Vivado
- Python automated the builds searching for maximum buildable frequency

Results

LZC bitwidth	LUTs(LUTNMs /MUXF7/MUXF8)	Slices	Power (mW)	Delay (ns)	Freq. (MHz)
8 new/old [1]	4 (1)	2	10	0.808	600
16 old [1]	12 (1)	3	13	1.016	650
32 old [1]	29 (1)	7	11	1.226	650
64 old [1]	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

Table: Performance Results for various LZC sizes. "new" is synthesis without the *KEEP* attribute. [1] Zahir, et al.

Explanation of Results

- At very high frequencies, deeper circuits can in some cases perform better as the path delay effects two slack values for registers latching result signals:
  - setup (which balances the clock skew against the path delay, clock uncertainty and setup time)
  - hold (balancing path delay against clock skew, uncertainty and hold time) slacks for the registers
- For example at a clock speed of 650MHz, the clock period is $\frac{10^3}{650\text{MHz}} = 1.538$ nanoseconds (Vivado timing scores actually use picoseconds)
  - 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.
- VHDL attributes constraining synthesis can significantly change the result
- Sometimes not using attributes has smaller path delay but uses more resources as Vivado uses physical device timing details
- Our implementation scales better than prior ones and minimizes the path delay

Conclusion and Future Research

- Optimal circuit not provable due to NP-complete circuit satisfiability problem (circuit-SAT)
- LZC can benefit from understanding of underlying architecture with more complicated logic over modularity
- 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
- Ideas applicable to more applications like counting bits set (sometimes called *popcount*), checking for powers of two, or rounding up to the nearest power of two, etc.

Active Option