IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 4 July 2022, accepted 21 July 2022, date of publication 28 July 2022, date of current version 3 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3194538

RESEARCH ARTICLE

Use of Carry Chain Logic and Design System Extensions to Construct Significantly Faster and Larger Single-Stage N-Sorters and N-Filters

ROBERT B. KENT, (Life Member, IEEE), AND MARIOS S. PATTICHIS, (Senior Member, IEEE)
Department of Electrical and Computer Engineering, The University of New Mexico, Albuquerque, NM 87131, USA
Corresponding author: Robert B. Kent (rkent@unm.edu)

ABSTRACT The authors' recently published design system for the creation of single-stage N-sorter/N-filter sorting devices, which were implemented using a particular example hardware block, is here expanded and applied to a second hardware type, FPGA carry chain logic. Although several researchers have published applications which use FPGA carry chain logic, most do not use carry chain logic as is done here, and none of the applications target sorting devices. Prior to the introduction of the single-stage N-sorter/N-filter design system, the fastest state-of-the-art hardware devices which sorted more than 2 input values were multistage sorting networks, in which the sorting process is performed by one or more 2-sorters and 2-max/2-min filters, operating in each sequential stage. Using the authors' original design system, single-stage N-sorters and N-filters were shown to be faster than the fastest comparable sorting networks when sorting 3 to 9 inputs. Here, product term splitting and a new Sum-of-Products output multiplexer equation are added to the design system, and this expanded design system is then implemented in carry chain logic to build faster and larger N-sorters, and much larger and still fast N-max/N-min filters. The new carry chain N-sorters are implemented in the FPGA used in the Amazon AWS EC2 F1 instance, which is one of the two example FPGAs utilized in the previous publication. A carry chain logic 16-sorter, not practical when using the original hardware block, has a speedup of 4.61 relative to the fastest 16-network. An example of the new, very large single-stage carry chain N-max filters is the 125-max 5 × 5x5 CNN video max pooling filter, which operates in only 2.075 nS. A 2-stage 1024-max network, using single-stage 32-max carry chain filters, has a speedup of 2.85 versus the existing state-of-the-art 10-stage network of 2-max filters.

INDEX TERMS Field programmable gate arrays, FPGA, sorting, sorting networks, video max pooling.

I. INTRODUCTION

The authors have recently published a system for designing fast, stable, single-stage hardware sorting devices, which sort 3 or more input values [1]. A single-stage device has one set of ports, one set of output ports, and whatever internal logic is needed in order to produce a fully sorted list of the input values at the output ports. If the device's output list is the full sorted list of all N inputs, the device is called an N-sorter. If the device outputs are only a subset of the full sorted list, it is called an N-filter. Examples of N-filters include N-max, N-min, N-median, and lowest-2-of-5 filters. The term "sorters" here will refer to both single-stage N-sorters and N-filters.

Although the general design system introduced in [1] is not hardware-specific, a design logic block (LB) common to two FPGA families was used to show how the sorting devices are constructed. Using synthesis results for two products in those two families, it was shown that the single-stage sorters were significantly faster than the multistage sorting networks considered at that time to be the fastest state-of-the-art hardware sorting devices.

The authors' main motivation for both this and the previously published research is to find design methods

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

that will produce faster single-stage sorting devices than the comparable fastest sorting networks. This work strives to build even faster and larger devices than were created in [1], and to build the new devices with a second hardware type, carry chain logic. To this end, the N-sorter/N-filter design system from [1] is now expanded, with the addition of product term splitting and Sum-of-Products (SOP) output multiplexer (mux) equations, and is then used to build fast sorting devices using carry chain logic.

Hardware vendors use carry chain logic to accelerate addition, subtraction, 2-value comparisons, and other processes that the vendors automatically implement. These carry chain LBs typically include carry lookahead logic internal to the LB to accelerate signals traveling along the chain. The carry chain LBs can also be easily cascaded vertically to form tall carry chains with hardwired routing within the chain.

This dedicated vertical routing produces fast signal paths, versus the general programmable horizontal routing which connects one LB's outputs to the inputs of the next LB in series. A signal's general routing is not finalized until place and route, and must compete with the general routing of all signals in the design. The vertical routing within a carry chain is finalized at synthesis, and therefore the signal's speed through the carry chain is also determined at synthesis.

The speed of a hardware sorting device is determined by the propagation delay of the slowest signals in the device, as the signals travel from the device inputs to outputs. In a product like an FPGA, propagation delay, to a first order, is proportional to the number of LBs in series that the slowest signals travel through. The inputs to each LB are provided by general routing, so a signal's number of series LBs is also a measure of the number of general routing sections that the signal must traverse. Here, the phrase ''series LBs'' refers to the horizontal number of LBs in series that a sorting device's slowest signals propagate through.

In the design of N-sorter/N-filter signals, it is possible to use a single tall carry chain to replace 2 series LBs and their intervening general interconnect routing, which will lower the overall signal propagation delay. Use of tall carry chains also allows implementation of very large single-stage N-max/N-min filters. These N-max/N-min filters can then be used in large, simple, and fast N-max/N-min networks. All of these design and speed improvements will be implemented and characterized in this study.

The design system defined in [1] is the only system which successfully enables the design of novel FPGA sorting devices that sort more 2 devices in a single-stage set of operations. The unsuccessful attempts by other researchers to define a system for single-stage sorting device design were discussed in [1], and that discussion is not repeated here. It does not appear that any competing single-stage sorter systems have been proposed since the publication of [1].

Single-stage 2-sorters and 2-max/2-min filters have been used for many years, and are the devices used in parallel in each stage of a multistage sorting network. The fastest sorting networks that fully sort from 3 up to 16 values have been custom-designed by a number of individuals, and are listed in Donald Knuth's classic text [2] and in a more recent publication [3]. These fastest previous state-of-the-art devices are the sorting networks that the single-stage devices in [1] were compared to, and are the devices that this work's new single-stage devices are compared against as well. The very simple N-max/N-min sorting networks using 2-max and 2-min filters have been briefly covered in [4].

Kenneth Batcher's two sorting network algorithms, Odd-Even Merge Sort and Bitonic Merge Sort [5], typically produce the fastest sorting networks when sorting more than 16 values. Portions of these two networks are also used in special sorting operations, such as 2-way merge sequences for small sorted input lists. Sorting networks in FPGAs are specifically covered in [4] and [6].

The basics of carry chain logic in FPGAs are presented in [6], [7]. A number of researchers have published applications which use FPGA carry chain logic, but most of these carry chain logic uses are not closely related to the way this logic is used in this study [8]–[16]. A very recent publication [17] does use carry chain logic in a manner similar to some of the ways used here for N-sorter/N-filter design, but none of these publications target using carry chain logic for sorting devices.

In this article, the additions proposed above to the N-sorter/N-filter design system have now been implemented, and the expanded system has been used to build and synthesize many sorters using carry chain logic. The new single-stage devices defined here, like those in [1], produce a stable sort. Here, stable sort is defined as a sorting method where duplicate values in the sorted output list are returned in the same order as they appear in the input list.

The carry chain logic block used in the study is found in the AMD-Xilinx xcvu9p UltraScale+ FPGA, the FPGA used in the Amazon AWS EC2 F1 cloud computing instance, and one of the two FPGAs used in [1]. All of the new carry chain logic single-stage devices, the previous single-stage devices constructed in [1], and comparable multistage sorting networks, have been synthesized with the same software tool, targeting the xcvu9p product. This enables direct comparison of speed and hardware usage results between the three types of sorting devices.

Based on the synthesis result comparisons, the predicted speed improvements and series LB reduction for the new carry chain devices have been realized. The new devices are shown to be the fastest state-of-the-art sorting devices for the number of unsorted input values that are processed.

Throughout this work, propagation delay, speedup, and resource usage results for various sorters are reported in the text. The reported results are for devices which sort 32-bit unsigned integer input values. For comparison's sake, there are 4 tables in Sections V-A and V-B that list data for 8-bit as well as 32-bit values, but 8-bit data are not otherwise presented or discussed.

In what follows, the paper provides a detailed summary of the key contributions of the paper. The primary contributions

include significant speedups and the implementation of large N-sorters and N-filters.

The expanded design system and carry chain logic produce N-sorters which require fewer series LBs and are faster than the devices constructed in [1]. For example, the slowest signals for the original state-of-the-art single-stage 7-sorter travelled through 4 LBs in series, and produced a worst case propagation delay of 1.812 nS. The slowest signals for the new single-stage 7-sorter, implemented using carry chain logic, propagate through only 2 LBs in series, with a propagation delay of 1.469 nS, a speedup of **1.23=1.812/1.469** versus the original 7-sorter. In addition, the new 7-sorter has a speedup 4.24 versus the comparable sorting network.

The use of product term splitting and tall carry chains enables implementation of larger N-sorters than were practical using the design system and hardware in [1]. The new single-stage 16-sorter design operates in 2.024 nS, a speedup of 4.61 versus the existing state-of-the-art 16-sorter, a 9-stage network of 2-sorters.

The expanded design system with carry chain logic is especially advantageous for implementation of large N-max/N-min filters. A state-of-the-art 9-max 3×3 2D image window filter was described in [1], but a much larger 125-max $5\times5\times5$ filter is now defined, which is useful for 3D video max pooling in Convolution of Neural Nets (CNN) [18]. A 125-max filter operates in 2.075 nS, less than a 480 MHz clock period.

Simple networks are easily constructed to implement significantly larger N-max/N-min sorting devices. A 1024-max network, constructed using 2 stages of single-stage 32-max filters, has a propagation delay of 3.557 nS, less than the period of a 280 MHz clock. This 2-stage network has a 2.85 speedup versus the current state-of-the-art 1024-max network, which uses 10 stages of 2-max filters.

The rest of this article is organized as follows. Section II discusses the background of sorting networks, the single-stage sorting devices from [1], and applications which have used FPGA carry chain logic. Section III presents the design system additions, and shows how the expanded design system is used to construct N-sorters using carry chain logic. N-filter design using carry chain logic is presented in Section IV, and focuses on N-max/N-min filters and simple networks that use N-max/N-min filters. Speed and resource usage synthesis results for various new devices are presented in Section V, and compared to the results for sorting networks and the earlier single-stage devices. A discussion of the logic blocks, particularly the carry chain logic blocks, for the newest FPGA families is found in Section VI. Section VII contains the conclusion, and Appendix A contains an analysis of how the design system improvements can be used to produce improved sorters using the FPGA hardware from [1].

II. BACKGROUND

Prior to [1], the only fast and efficient single-stage sorting devices that were available in FPGA hardware were 2-sorters and 2-max and 2-min filters. Sorting more than 2 input values required a sorting network, in which the sorting process consists of a series of stages, with 2-sorters and/or 2-filters utilized in each stage. General 2-sorter networks can be constructed using Kenneth Batcher's Odd-Even Merge Sort or Bitonic Merge Sort [5], and custom networks can be found in Donald Knuth's text [2], and more recently in [3]. N-max/N-min networks using 2-max/2-min filters are perhaps too simple to receive much focus, but they are briefly covered in [4].

A review of the prior research related to single-stage sorting devices is found in [1], and is not repeated here. A summary of [1]'s novel single-stage N-sorters/N-filters design system is presented in Section II-A below.

The primary focus of this work is to use vendor-supplied carry chain logic to build faster and larger single-stage sorters. Various applications have been proposed which take advantage of provided carry chain LBs, and these applications are discussed below in Section II-B.

A. THE AUTHORS' SINGLE-STAGE DESIGN SYSTEM

The general design system for single-stage stable N-sorters was defined in [1]. This section contains notation, definitions, and discussion of that design system. The terms defined here, such as ge_L_R and In_X_goes_to_Out_Y, are used throughout this work. The main definitions that are used throughout the text are listed below:

- A list of N unordered input values is processed into a fully sorted output list, containing those same N values.
- The inputs are identified as inputs In_(N-1) down to input In_0, and the outputs are identified as outputs Out_(N-1) down to output Out_0.
- The output list sorted order is uniformly non-increasing from output Out_(N-1) down to output Out_0.
- All unique $N*(N-1)/2$ input value comparisons are created in order to correctly sort the input list.
- Each 2-input comparison uses the greater-than-or-equal $\geq$ operator, and the input with the higher numeric suffix is always on the left side of the comparison operator.

A comparison result signal name is of the form of ge_L_R, where ''ge'' stands for greater-than-or-equal, L is the higher numeric suffix for the input signal on the left side of operator, and R is the lower numeric suffix for the input signal on the right side of operator.

- For each input In_X, with X>0, **N** Sum-of-Product (SOP) In_X_goes_to_Out_Y equations are defined, each of which maps a specific input In_X to a specific Out_Y output port. For a given In_X, only one In_X_goes_to_Out_Y signal will be true. Likewise, for a given Out_Y, only one In_X_goes_to_Out_Y signal can be true.

In order to build the In_X_goes_to_Out_Y equations for input In_X, all 2^{N-1} unique product terms are constructed in which the variables in each product term are the $N-1$ ge_L_R comparison result signals in which In_X is compared to the other inputs. Each product term is then analyzed in order to determine the number of ''winner'' comparison results for In_X in that product term.

The definition of a comparison result winner for input In_X is based on whether In_X is on the left (L) or right (R) side of the $\geq$ comparison operator, and whether the ge_L_R comparison result is complemented or not. If In_X is on the right side of the $\geq$ comparison operator, then a winner is a complemented (! ge_L_R) value. If In_X is on the left side of the $\geq$ comparison operator, then a winner is a (ge_L_R) value that is not complemented. An In_X product term with **Y** number of winners is then ORd into the associated In_X_goes_to_Out_Y SOP equation.

```
wire    In_6_goes_to_Out_5  =

  (   ge_6_5 &&   ge_6_4 &&   ge_6_3 &&   ge_6_2 &&   ge_6_1 && ! ge_6_0 ) ||
  (   ge_6_5 &&   ge_6_4 &&   ge_6_3 &&   ge_6_2 && ! ge_6_1 &&   ge_6_0 ) ||
  (   ge_6_5 &&   ge_6_4 &&   ge_6_3 && ! ge_6_2 &&   ge_6_1 &&   ge_6_0 ) ||
  (   ge_6_5 &&   ge_6_4 && ! ge_6_3 &&   ge_6_2 &&   ge_6_1 &&   ge_6_0 ) ||
  (   ge_6_5 && ! ge_6_4 &&   ge_6_3 &&   ge_6_2 &&   ge_6_1 &&   ge_6_0 ) ||
  ( ! ge_6_5 &&   ge_6_4 &&   ge_6_3 &&   ge_6_2 &&   ge_6_1 &&   ge_6_0 )   ;
```

FIGURE 1. 7-sorter In_6_goes_to_Out_5 SOP equation.

Fig. 1 shows the In_6_goes_to_Out_5 SOP equation for a 7-sorter's In_6 input, in which each of the six product terms has 5 winners. A 7-sorter's In_6 is always on the left side of the $\geq$ operator, therefore all uncomplemented comparisons are winners, and each product term in Fig. 1 has 5 uncomplemented ge_6_R comparisons. If one of these product terms is true, In_6_goes_to_Out_5 is true, and input In_6 will be mapped and transferred to output port Out_5. Also, no other In_6_goes_to_Out_Y signal will be true, and no other input's In_X_goes_to_Out_5 signal will be true. Fig. 1 and many of the following figures use SystemVerilog syntax [19].

The mapping of a unique input to a target output is implemented in per-bit output multiplexers constructed for that output. An output multiplexer is designed using an equation which utilizes ternary, i.e., conditional syntax.

```
assign  Out_5[ 0 ]
   =
       ( In_6_goes_to_Out_5 ? In_6[ 0 ] :
       ( In_5_goes_to_Out_5 ? In_5[ 0 ] :
       ( In_4_goes_to_Out_5 ? In_4[ 0 ] :
       ( In_3_goes_to_Out_5 ? In_3[ 0 ] :
       ( In_2_goes_to_Out_5 ? In_2[ 0 ] :
       ( In_1_goes_to_Out_5 ? In_1[ 0 ] :
                              In_0[ 0 ]  ) ) ) ) ) ) ;
```

FIGURE 2. 7-sorter out_5[0] output multiplexer ternary equation.

An example of a ternary output mux equation, for a 7-sorter's Out_5, is shown in Fig. 2. Although there are 7 inputs, only 6 In_X_goes_to_Out_5 signals are needed. Specifically in Fig. 2, there is no In_0_goes_to_Out_5 signal. If none of the other In_X_goes_to_Out_5 signals are true, then In_0, by default, will go to Out_5.

Per-bit equations and figures, such as Fig. 2, typically target bit 0. Whenever bits other than bit 0 are targeted, this will be noted.

The system of In_X_goes_to_Out_Y signals, and their use in output multiplexers, ultimately implements a comparison counting sort process. The comparison counting is performed in a combinatorial process by triggering a product term, one product term per input, in an In_X_goes_to_Out_Y SOP equation. Synchronous counters are not needed.

The N-sorter sorting process defined in [1] and summarized here produces a stable sort. Equal values in the input list are presented in the same order in the output list.

N-filter design generally follows the system described above. All $N*(N-1)/2$ comparison signals must still be created, even for a single-output filter. All outputs not in the N-filter's output port list are removed, as well as any internal logic that only supports the removed ports.

The In_X_goes_to_Out_Y signals for N-max and N-min filters only have one product term, which enables unique designs for these filters. This N-max/N-min feature was discussed in [1], and is discussed further and emphasized here.

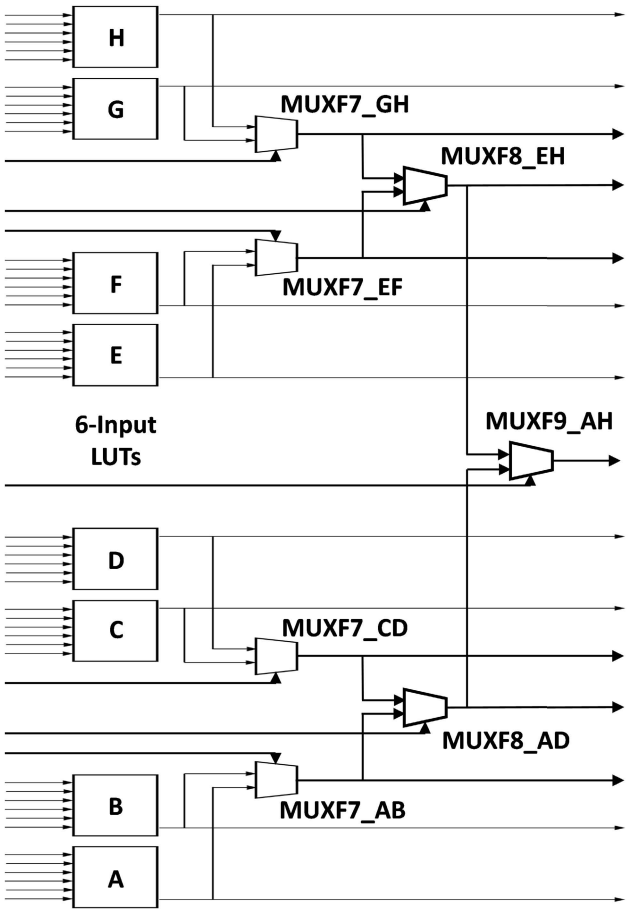


FIGURE 3. UltraScale+ LUT-MUXF7/8/9 logic block: 8 6-input LUTs.

As mentioned earlier, an example hardware LB was used in [1] to show how these novel sorting devices are constructed. The example LB was also utilized to build equivalent sorting networks using 2-sorters/2-filters, so that the novel single-stage devices could be compared to analogous 2-networks, which were the state-of-the-art at that time. The example LB, which is found in the xcvu9p UltraScale+ FPGA [20], is shown in Fig. 3. The Fig. 3 LB

contains 8 6-input lookup tables (LUTs), and 3 levels of 2-to-1 multiplexers, which are used as needed to combine LUT outputs inside the block. This LB is used to build output multiplexers as well as In_X_goes_to_Out_Y signals. Since In_X_goes_to_Out_Y product terms contains $N-1$ comparison variables, the In_X_goes_to_Out_Y signals for 7- and smaller N-sorters can be implemented in one 6-input LUT.

For N-sorters larger than a 7-sorter, one or more of the 2-to-1 multiplexer levels are required in order to build an In_X_goes_to_Out_Y signal in a single Fig. 3 LB. If all 3 2-to-1 multiplexer levels are utilized to implement an In_X_goes_to_Out_Y signal, the full LUT-MUXF7/8/9 structure will support any SOP equation with 9 variables. A 10-sorter requires $10-1=9$ variables in an In_X_goes_to_Out_Y signal, so a 10-sorter is the largest N-sorter whose In_X_goes_to_Out_Y signals can be constructed in a single Fig. 3 LB. Because of this, the largest xcvu9p N-sorter that can be efficiently implemented using only the Fig. 3 LB is a 10-sorter.

The fastest signals travelling through a Fig. 3 LB propagate through only a single LUT. Signals travelling through the 2-to-1 multiplexer levels are slowed down relative to a signal traveling only through a single LUT in the block, but any path fully internal to this LB will be a fairly fast hardwired path.

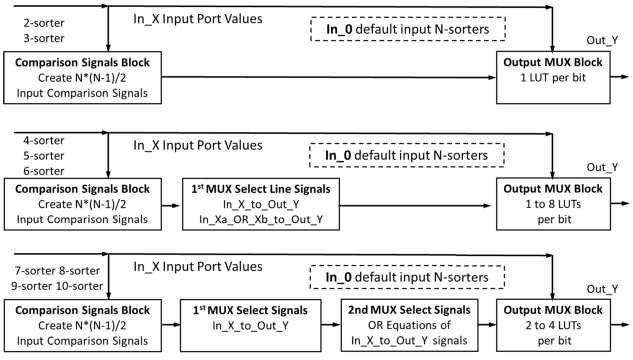


FIGURE 4. Signal flow diagrams for the baseline LUT-MUXF7/8/9 N-sorters.

Signal flow diagrams for the baseline N-sorter designs described in [1] are shown in Fig. 4. An N-sorter's slowest signals always propagate through the Comparison Signals Block, shown at the left in each of the three diagrams. The slowest signals for the 2- and 3-sorter propagate through only 2 LBs in series, and these signals travel through only a single LUT in the second series LB, the Output MUX Block. The 4-, 5-, and 6-sorters in the middle diagram have 3 series LBs, and the slowest signals for 7- through 10-sorters propagate through 4 LBs in series.

B. CARRY CHAIN LOGIC

FPGA vendors such as AMD-Xilinx have implemented carry chain logic, with fast carry lookahead, beginning with some of their earliest product families [7]. The primary purpose of this carry chain logic is to produce fast 2-value comparisons,

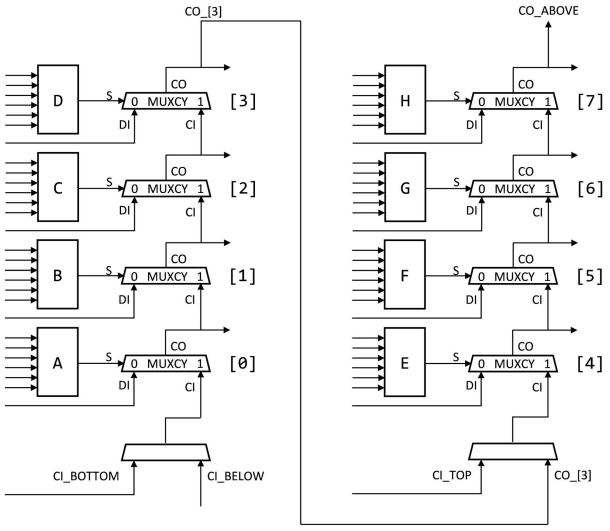


FIGURE 5. UltraScale+ CARRY8 LUT-MUXCY 8 segment carry chain.

addition, subtraction, and other basic arithmetic operations. In addition to its default uses by FPGA vendors, carry chain logic is also available for logic designers to use as desired.

The example carry chain LB used in this work is shown in Fig. 5. This LB is also found in the xcvu9p FPGA, and is an alternate configuration of the basic "slice" hardware that produces the Fig. 3 LB [20]. The Fig. 5 and Fig. 3 LBs are found in all products of the AMD-Xilinx UltraScale and UltraScale+ FPGA families.

Here, each of the 8 LUT-MUXCY pairs is called a segment. Each segment has a carry in (CI) input, a data in (DI) input, a carry out (CO) output, as well as a multiplexer select (S) signal, which is the output of the segment 6-input LUT. As can be seen in Fig. 5, the 8-segment chain can be divided into two independent 4-segment chains, as needed. A segment CI input is typically the CO output of the previous segment in the chain. However, the user must specify the CI input for segment 0, and for segment 4 when there are two 4-segment chains.

When used for per-bit additions, subtractions, etc., each segment has an additional XOR gate and SUM output, which are not shown in Fig. 5. The XOR/SUM structures are not used in the N-sorter/N-filter carry chains, each of which produces a single signal which is tapped off from the CO output of a target segment. Also not shown in Fig. 5 is the fact that the segment DI input can come from a second output in the segment LUT. All N-sorter/N-filter carry chains described here utilize DI signals that only come in from the general interconnect.

A number of research publications have used an FPGA's full carry chain logic, with the SUM output, for multiple input addition and similar arithmetic functions [8]–[12]. As these articles use the per-segment SUM output, they are not closely related to the use of carry chain logic which is being proposed here.

Other publications have used a carry chain LB like Fig. 5 to produce a single output signal, and do not require segment SUM outputs in their implementation. In one patent, one or more common factors were extracted from an SOP equation with numerous product term variables, and then carry chain logic was used to AND the common factors with the narrow SOP equation produced after factoring [13].

A number of researchers have studied use of carry chain logic to build time-to-digital (TDC) converters. One recent publication reviewed the history of these studies and proposed its own TDC using two independent carry chains, each of which is designed into a ring oscillator [14].

Researchers have also used carry chains without segment SUM outputs to map general logic equations into FPGAs, by using AND-Inverter-Graphs (AIG) [15] or Majority-Inverter-Graphs (MIG) [16] systems. Because of their reliance on example carry chain structures and focus on general equations, these systems tend to create slower and inefficient designs compared to the methodology defined in this work, which systematically implements a set of specific N-sorter/N-filter equations on the target Fig. 5 carry chain LB.

A very recent publication also attempts to map a general logic equation into an FPGA [17]. Its methodology creates AND and OR equations which are similar to AND and OR equations used in this work. The AND equations default to 1, use constant 0 DI segment inputs, and the 0 is written to the carry chain whenever the segment equation fails. The OR equations default to 0, use constant 1 DI segment inputs, and the 1 is written to the carry chain whenever the segment equation is true.

III. N-SORTERS USING CARRY CHAIN LOGIC

As shown in Fig. 4 and mentioned above, each output multiplexer for 2-sorters and 3-sorters is simply constructed in a single LUT, so these two N-sorters have no need of MUXF7/8/9 2-to-1 multiplexers or carry chain logic. Therefore, use of carry chain logic here targets sorters with 4 or more input values.

As noted earlier, the example carry chain logic used in this study is shown in Fig. 5. This is the carry chain logic block found in the xcvu9p FPGA used in [1], and in all of the FPGAs in the AMD-Xilinx UltraScale and UltraScale+ product families.

AMD-Xilinx provides the CARRY8 primitive to users who wish to directly implement their designs in the Fig. 5 carry chain LB. Users specify the CARRY8's initial CI signal, the segment DI and S signals, and then tap out the appropriate segment CO output for their target signal. In order to show how the carry chain designs discussed here are implemented using the CARRY8 block, 8-segment figures are used, in which all of the CARRY8 inputs are specified, as well as which segment CO output is tapped out for the target signal.

Fig. 6 is the first such figure, and it shows the multiplexer setup for a carry chain 7-sorter's Out_5 output. This configuration implements the Fig. 2 equation. The initial

		Inputs	Outputs
[7]	S	1	CO
	DI	0	
[6]	S	1	CO
	DI	0	
[5]	S	! In_6_goes_to_Out_5	Out_5[0] CO
	DI	In_6[0]	
[4]	S	! In_5_goes_to_Out_5	CO
	DI	In_5[0]	
		CI[4] = CO[3]	
[3]	S	! In_4_goes_to_Out_5	CO
	DI	In_4[0]	
[2]	S	! In_3_goes_to_Out_5	CO
	DI	In_3[0]	
[1]	S	! In_2_goes_to_Out_5	CO
	DI	In_2[0]	
[0]	S	! In_1_goes_to_Out_5	CO
	DI	In_1[0]	
		CI[0] = CI_BOTTOM = In_0[0]	

FIGURE 6. 7-sorter out_5 bit 0 carry chain setup.

carry in signal is the default In_0 bit. Each segment LUT produces an In_X_goes_to_Out_5 signal, and each segment DI input is the associated In_X bit.

A segment DI signal is mapped to CO when the LUT output S signal is a 0, so the In_X_goes_to_Out_5 signals are complemented in order to select the correct segment MUXCY input. Note that the S signal in segment 5 is the complement of the Fig. 1 signal.

Fig. 6 implements an OR equation, but not the same type of OR equation used in [17]. The [17] OR equation has a default of 0, and a segment true OR result writes a 1 to the CO output. In Fig. 6, the default is the In_0 bit value, and a segment true OR result will write the associated input value to CO.

All of the other bits for Out_5 use the same configuration shown in Fig. 6, with the same In_X_goes_to_Out_5 signals. Only the bit index for the input and output ports changes.

The setup of carry chain 4-, 5-, and 6-sorters is like that of the 7-sorter, as each In_X_goes_to_Out_Y signal can be produced in a segment LUT. The slowest signals for these N-sorters propagate through only 2 LBs in series, as shown in the top flow diagram in Fig. 7. The previous baseline 4- through 6-sorters needed 3 series LBs, as shown in the middle Fig. 4 flow diagram. The bottom flow diagram in Fig. 4 shows that the original baseline 7-sorter required 4 LBs in series.

The In_X_goes_to_Out_Y signals for 8- and larger N-sorters have at least 7 comparison variables in their product terms, so these signals cannot be produced in a 6-input segment LUT. For 8- to 10-sorters designed with the middle flow diagram in Fig. 7, In_X_goes_to_Out_Y signals are built using 2-to-1 mux groups in the Fig. 3 LB. An 8-sorter uses 2 LUTs and a MUXF7 for its signals, and a 10-sorter requires all 8 LUTs and all 7 MUXF7/8/9 multiplexers for its In_X_goes_to_Out_Y signals.

The 8- to 10-sorters then use new SOP output multiplexer equations in their Output MUX Block, an example of which is shown in Fig. 8 for a 10-sorter's Out_Y. Here, Out_Y is used to represent each of the 10-sorter's 10 output ports. There are no default inputs in an SOP output mux equation,

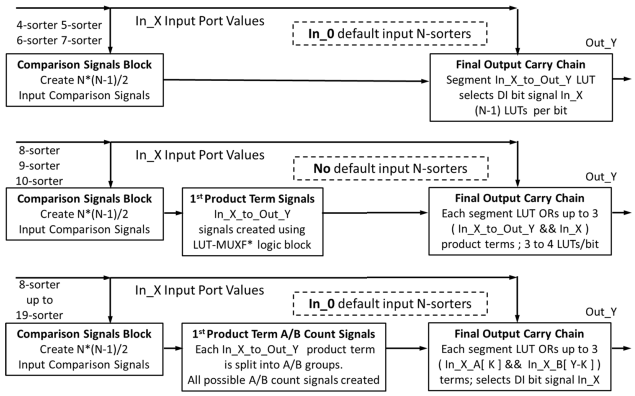


FIGURE 7. Signal flow diagrams for LUT-MUXCY carry chain N-sorters.

```
assign  Out_Y[ 0 ]
   =
       ( In_9_goes_to_Out_Y && In_9[ 0 ] ) ||
       ( In_8_goes_to_Out_Y && In_8[ 0 ] ) ||
       ( In_7_goes_to_Out_Y && In_7[ 0 ] ) ||
       ( In_6_goes_to_Out_Y && In_6[ 0 ] ) ||
       ( In_5_goes_to_Out_Y && In_5[ 0 ] ) ||
       ( In_4_goes_to_Out_Y && In_4[ 0 ] ) ||
       ( In_3_goes_to_Out_Y && In_3[ 0 ] ) ||
       ( In_2_goes_to_Out_Y && In_2[ 0 ] ) ||
       ( In_1_goes_to_Out_Y && In_1[ 0 ] ) ||
       ( In_0_goes_to_Out_Y && In_0[ 0 ] )     ;
```

FIGURE 8. New 10-sorter out_Y[0] output mux SOP equation.

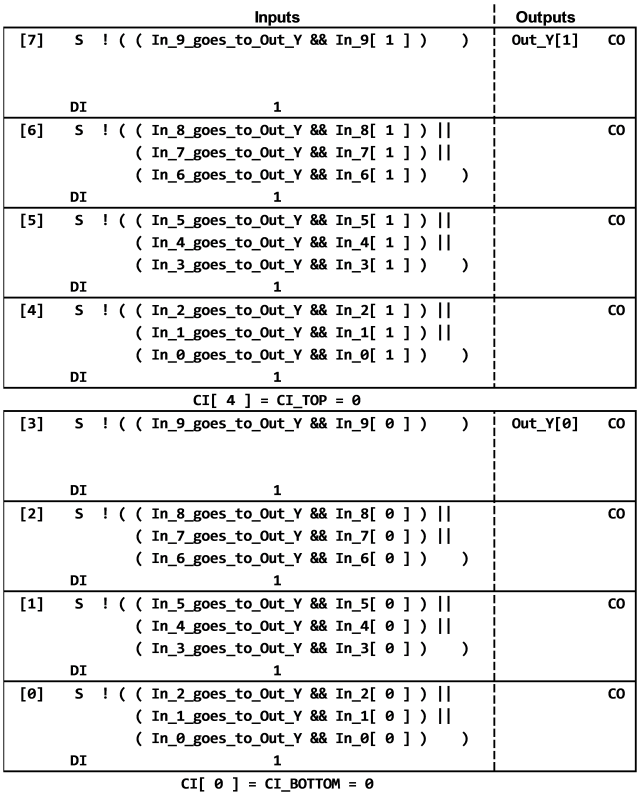


FIGURE 9. 10-sorter LUT-MUXCY bits 0 and 1 output SOP mux setup.

so In_0_goes_to_Out_Y signals must be constructed and used.

Fig. 9 shows the 8-segment carry chain setup for 10-sorter bits 0 and 1, using SOP output mux equations. As each product term contains 2 signals, an In_X_goes_to_Out_Y signal and its associated In_X input bit, up to 3 product terms can be placed in a 6-input segment LUT. Note that segments 0 to 3 implement the Fig. 8 equation.

The Fig. 8 SOP output mux equation does not have a default input, but it does have a default logic value of 0. Therefore, the two Fig. 9 chains are each initialized with logic 0, and then each segment DI value is set to a logic 1. One of the In_X_goes_to_Out_Y signals is a 1, and if the associated input bit is also a 1, the DI 1 value is written to the segment CO output. If the associated input bit is a 0, none of the segment product terms will be true, and Out_Y will be the default 0 value.

The Fig. 8 equations are OR equations that are implemented in essentially the same manner as the OR equations in [17]. The equation default is 0, and any true product term result will cause a DI 1 to be written to CO.

The 10-sorter is the largest carry chain N-sorter that can be constructed using the middle flow diagram in Fig. 7 and the 10-sorter was the largest of the previous Fig. 4 baseline N-sorter designs. As mentioned earlier, any larger N-sorter would require 10 or more variables in an In_X_goes_to_Out_Y product term, which cannot be implemented in a single Fig. 3 LB.

In order to build larger carry chain N-sorters, another new feature has been added to the N-sorter design system: product term splitting for In_X_goes_to_Out_Y SOP equations. With product term splitting, the comparison variables that are found in a given input's In_X_goes_to_Out_Y equations are split into two A/B groups. Winner count SOP equations are determined separately for each group, and the count signals are placed in an array in which the array index indicates the winner count for that signal.

Then, for a particular output Out_Y, all A/B count pairs are found in which the A count plus the B count add up to Y, and then a new SOP equation is created in which each product term consists of an A count signal which is ANDed with the B count signal with which it is paired. This creates a new type of In_X_goes_to_Out_Y equation for N-sorters with $N > 10$.

Examples of this new type of A/B product term are shown in Fig. 10, which shows the setup for the highest 8 of 48 segments for a 16-sorter's Out_8 output. For each A/B pair, the counts, as indicated by the array indices, add up to 8. There are 15 comparison variables in a 16-sorter's In_X_goes_to_Out_Y product terms, and these are split into an A group with 7 variables and a B group with 8 variables. The A group counts range from 0 to 7 and all 8 A group count signals are used for each input. The B group counts range from 0 to 8, but the B group count 0 signal is not used for Out_8. The B group count 0 signal is used for several other 16-sorter outputs.

```
Segment                          Inputs                                     Outputs
[7]   S    ! ( ( In_15_to_Out_8_A[ 0 ] && In_15_to_Out_8_B[ 8 ] ) ||      Out_8[0]  CO
               ( In_15_to_Out_8_A[ 1 ] && In_15_to_Out_8_B[ 7 ] )     )

      DI                       In_15[ 0 ]
[6]   S    ! ( ( In_15_to_Out_8_A[ 2 ] && In_15_to_Out_8_B[ 6 ] ) ||                CO
               ( In_15_to_Out_8_A[ 3 ] && In_15_to_Out_8_B[ 5 ] ) ||
               ( In_15_to_Out_8_A[ 4 ] && In_15_to_Out_8_B[ 4 ] )     )
      DI                       In_15[ 0 ]
[5]   S    ! ( ( In_15_to_Out_8_A[ 5 ] && In_15_to_Out_8_B[ 3 ] ) ||                CO
               ( In_15_to_Out_8_A[ 6 ] && In_15_to_Out_8_B[ 2 ] ) ||
               ( In_15_to_Out_8_A[ 7 ] && In_15_to_Out_8_B[ 1 ] )     )
      DI                       In_15[ 0 ]
[4]   S    ! ( ( In_14_to_Out_8_A[ 0 ] && In_14_to_Out_8_B[ 8 ] ) ||                CO
               ( In_14_to_Out_8_A[ 1 ] && In_14_to_Out_8_B[ 7 ] )     )

      DI                       In_14[ 0 ]
                           CI[ 4 ] = CO[ 3 ]
[3]   S    ! ( ( In_14_to_Out_8_A[ 2 ] && In_14_to_Out_8_B[ 6 ] ) ||                CO
               ( In_14_to_Out_8_A[ 3 ] && In_14_to_Out_8_B[ 5 ] ) ||
               ( In_14_to_Out_8_A[ 4 ] && In_14_to_Out_8_B[ 4 ] )     )
      DI                       In_14[ 0 ]
[2]   S    ! ( ( In_14_to_Out_8_A[ 5 ] && In_14_to_Out_8_B[ 3 ] ) ||                CO
               ( In_14_to_Out_8_A[ 6 ] && In_14_to_Out_8_B[ 2 ] ) ||
               ( In_14_to_Out_8_A[ 7 ] && In_14_to_Out_8_B[ 1 ] )     )
      DI                       In_14[ 0 ]
[1]   S    ! ( ( In_13_to_Out_8_A[ 0 ] && In_13_to_Out_8_B[ 8 ] ) ||                CO
               ( In_13_to_Out_8_A[ 1 ] && In_13_to_Out_8_B[ 7 ] )     )

      DI                       In_13[ 0 ]
[0]   S    ! ( ( In_13_to_Out_8_A[ 2 ] && In_13_to_Out_8_B[ 6 ] ) ||                CO
               ( In_13_to_Out_8_A[ 3 ] && In_13_to_Out_8_B[ 5 ] ) ||
               ( In_13_to_Out_8_A[ 4 ] && In_13_to_Out_8_B[ 4 ] )     )
      DI                       In_13[ 0 ]
                           CI[ 0 ] = CI_BELOW
```

FIGURE 10. **16-sorter LUT-MUXCY out_8 Bit 0: top 8 of 48 segments.**

IV. N-FILTERS USING CARRY CHAIN LOGIC

The slowest signals for carry chain 4-sorters up to 7-sorters travel through the minimum 2 series LBs, as shown in the top flow diagram in Fig. 7. Therefore, N-filters for 4 to 7 inputs use the basic N-sorter design, such as the 7-sorter design shown in Fig. 6, and then simply remove the output multiplexers for the unused outputs.

For N-filters with $N > 7$, if outputs other than the max and min outputs are in the N-filter's output list, the same basic design concept holds. Start with the N-sorter, and then remove the unused outputs and any internal logic which only supports the removed outputs.

When $N > 7$ and the N-filter's output list only contains the max and/or min outputs, the single-stage N-filter's design can be greatly simplified, as is discussed in Section IV-A. Fast N-max and/or N-min networks are very easily designed using single-stage N-max/N-min filters, and Section IV-B discusses these simple and impressive networks.

A. N-MAX/N-MIN FILTERS USING CARRY CHAIN LOGIC

As has been noted previously, the In_X_goes_to_Out_Y equations for both the max and min outputs have only one product term. Therefore, the ge_L_R variables for an In_X_goes_to_Out_Y signal can be split into any number of groups. The fundamental principles in both N-max and N-min filter design are the same, so only N-max filters will be discussed further.

If there are 6 or fewer groups, the group results can be ANDed together in an output mux segment equation, as is shown in Fig. 11 for the setup of the top of 4 CARRY8s for a 32-max output mux. This filter has 6 A/B/C/D/E/F groups, with 2/6/6/6/6/5 ge_L_R variables in each group. Only the max count for each group is needed, as should be clear from Fig. 11.

```
                          Inputs                               Outputs
[7]   S  ! ( In_31_A_counts[2] && In_31_B_counts[6] &&        Out_31[0]  CO
             In_31_C_counts[6] && In_31_D_counts[6] &&
             In_31_E_counts[6] && In_31_F_counts[5]     )
      DI                 In_31[ 0 ]
[6]   S  ! ( In_30_A_counts[2] && In_30_B_counts[6] &&                   CO
             In_30_C_counts[6] && In_30_D_counts[6] &&
             In_30_E_counts[6] && In_30_F_counts[5]     )
      DI                 In_30[ 0 ]
[5]   S  ! ( In_29_A_counts[2] && In_29_B_counts[6] &&                   CO
             In_29_C_counts[6] && In_29_D_counts[6] &&
             In_29_E_counts[6] && In_29_F_counts[5]     )
      DI                 In_29[ 0 ]
[4]   S  ! ( In_28_A_counts[2] && In_28_B_counts[6] &&                   CO
             In_28_C_counts[6] && In_28_D_counts[6] &&
             In_28_E_counts[6] && In_28_F_counts[5]     )
      DI                 In_28[ 0 ]
                      CI[ 4 ] = CO[ 3 ]
[3]   S  ! ( In_27_A_counts[2] && In_27_B_counts[6] &&                   CO
             In_27_C_counts[6] && In_27_D_counts[6] &&
             In_27_E_counts[6] && In_27_F_counts[5]     )
      DI                 In_27[ 0 ]
[2]   S  ! ( In_26_A_counts[2] && In_26_B_counts[6] &&                   CO
             In_26_C_counts[6] && In_26_D_counts[6] &&
             In_26_E_counts[6] && In_26_F_counts[5]     )
      DI                 In_26[ 0 ]
[1]   S  ! ( In_25_A_counts[2] && In_25_B_counts[6] &&                   CO
             In_25_C_counts[6] && In_25_D_counts[6] &&
             In_25_E_counts[6] && In_25_F_counts[5]     )
      DI                 In_25[ 0 ]
[0]   S  ! ( In_24_A_counts[2] && In_24_B_counts[6] &&                   CO
             In_24_C_counts[6] && In_24_D_counts[6] &&
             In_24_E_counts[6] && In_24_F_counts[5]     )
      DI                 In_24[ 0 ]
                      CI[ 0 ] = CI_BELOW
```

FIGURE 11. **32-max LUT-MUXCY out_31 bit 0: top 8 of 32 segments.**

If the single product term must be split into more than 6 groups, a separate carry chain is used to produce each In_X_goes_to_Out_Y signal, as is shown for a portion of 125-max In_124_goes_to_Out_124 carry chain shown in Fig. 12. The 124 ge_L_R variables for each product term are split into 20 groups containing 6 variables, and a single group with 4 variables. Fig. 12 shows the bottommost CARRY8 setup for the signal, with 4 groups of 6 and the single group of 4. The other 16 groups of 6 are setup in two CARRY8s connected above the Fig. 12 CARRY8.

The Fig. 12 CARRY8 produces a tall AND function, with a default value of 1. Each segment DI input is a 0, which is written to the chain whenever a segment equation fails. This is the same type of AND equation used in [17].

The Fig. 12 carry chains are found in the 2nd series logic block for the Out_124 output. The In_X_goes_to_Out_124 signals are then used in the Out_124 output multiplexers, in the same manner as was shown for each 10-sorter bit in Fig. 9. The slowest signals for the 125-max device still travel through only 3 logic blocks in series.

B. CARRY CHAIN N-MAX/N-MIN FILTER NETWORKS

Sorting networks which only use N-max or N-min filters are easily constructed and easily understood. As N-min filter networks are constructed in a manner akin to that of N-max filter networks, only N-max filter networks are discussed here.

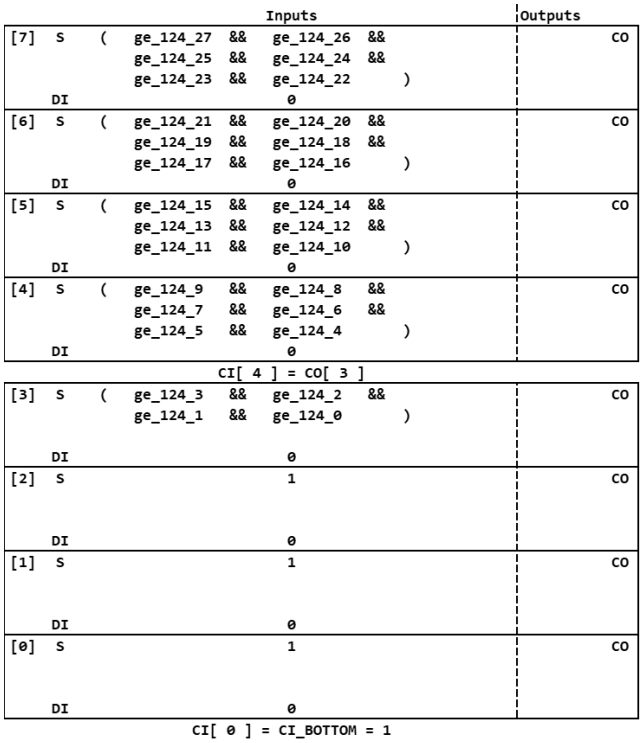


FIGURE 12. 125-max in_124_goes_to_out_124: bottom 8 of 24 segments.

Since $\mathbf{1024{=}32^2}$, a 1024-max network can be built using 2 stages of 32-max filters. In the first stage, the 1024 values are placed into 32 lists, with each list having 32 values. In parallel, the maximum of each list is determined using this first set of 32-max filters. In the 2nd and final stage, the 32 results from the first stage are processed using one more 32-max filter, which produces the final 1024-max result. After some thought, it should be clear that, if $\mathbf{P{=}N^q}$, the max of $\mathbf{P}$ values can be determined in a $\mathbf{q}$-stage network of N-max filters. If $\mathbf{P}$ is not a power of $\mathbf{q}$, then number of stages $\mathbf{q}{=}$ CEILING($\mathbf{\log_N(P)}$).

As $\mathbf{1024{=}2^{10}}$, the existing state-of-the-art 1024-max network is also constructed in a straightforward manner, using 10 stages of 2-max filters. The two different 1024-max networks are compared later in Section V-C.

V. RESULTS

Many single-stage N-sorter and N-filter designs have been defined here using SystemVerilog, targeting the AMD-Xilinx UltraScale+ xcvu9p FPGA. These designs have been synthesized using the Vivado 2018.2 synthesis tool, the same tool used in [1] when synthesizing devices which used the original design system and only used the Fig. 3 LB.

The design details and the synthesis run options which were used here have targeted low propagation delays, which may have also produced high LUT hardware resource usage. In [1], speed was important, but the design methodology and synthesis options there tended to focus on reducing resource usage.

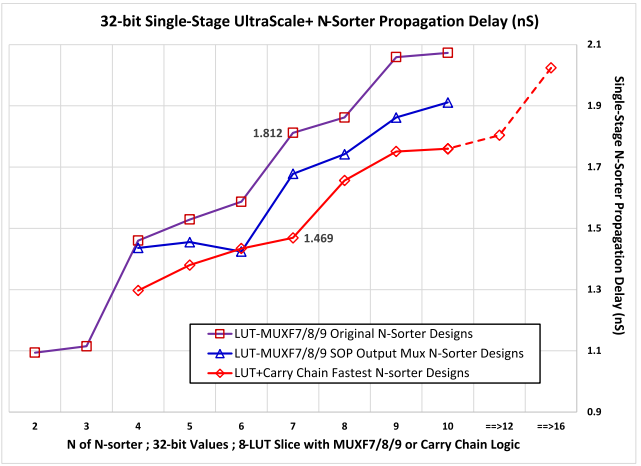


FIGURE 13. Single-stage 32-bit N-sorter propagation delay.

The single-stage N-sorter/N-filter results are compared to results from the historical state-of-the-art designs, which use 2-sorter/2-filter networks. The 2-network designs are also synthesized here with settings that emphasize speed, which tends to increase their LUT resource usage as well.

N-sorter and N-max filter designs for both 8-bit and 32-bit unsigned integers have been synthesized for this work. The synthesis results report both worst case propagation delay and LUT resource usage for each single-stage or network sorting device. For comparison's sake, data for both 8-bit and 32-bit devices are listed in the four tables found in the following two sections. However, as mentioned earlier, only data for 32-bit devices are plotted and discussed.

There is one caveat which concerns the LUT resource usage reported by the synthesis tool. During this study, it was noticed that the synthesis tool does not count LUTs which produce a constant 1, a constant 0, or passthrough/buffer a single input, even if these LUTs are an essential part of the design. There has been some effort put into counting these type of LUTs, but a rigorous method for counting these ignored LUT types for this work has yet to be developed.

A. SINGLE-STAGE N-SORTER SYNTHESIS RESULTS

Fig. 13 displays N-sorter propagation delay curves for 3 types of N-sorter designs. The purple top curve contains data for the original baseline LUT-MUXF7/8/9 designs, which use only the Fig. 3 LB and whose design flows are shown in Fig. 4. The data points for 2-sorters and 3-sorters, whose output muxes are simple LUTs and cannot be improved upon, are only shown in the purple curve.

The data in the middle blue curve come from new LUT-MUXF7/8/9 designs, which use only the Fig. 3 LB but take advantage of the new SOP output mux equations, like the one shown in Fig. 8. These designs are discussed in Appendix A. Note that the 6-sorter in this group is the only LUT-MUXF7/8/9 N-sorter that is (slightly) faster than a comparable carry chain logic N-sorter.

The bottom red curve contains the propagation delay data for the new carry chain logic N-sorters which are the focus

here. This curve shows that 7-sorters and larger carry chain sorters are clearly faster than LUT-MUXF7/8/9 N-sorters. The largest LUT-MUXF7/8/9 N-sorter is a 10-sorter, but product term splitting enables the larger carry chain N-sorters to be constructed, and data points for carry chain 12-sorters and 16-sorters are shown in the plot. The X-axis increments by 1 only up to 10, and any data points past 10 are shown using a dashed line. Note that the new carry chain 16-sorter is faster than the baseline 9-sorter.

The propagation delay for the original LUT-MUXF7/8/9 7-sorter design, 1.812, is labelled in Fig. 13's high purple curve. Likewise, the 7-sorter propagation delay for the new carry chain N-sorter, 1.469, is labelled in the low red curve. As was noted earlier in the introduction, the speedup for the carry chain 7-sorter is then $\mathbf{1.23{=}1.812/1.469}$ versus the original LUT-MUXF7/8/9 7-sorter.

The speeds of the carry chain logic N-sorters shown in the red curve in Fig. 13 are entered into the 3rd column of Table 1. The speed data of comparable sorting networks is found in the 5th column. These sorting networks, generally considered the state-of-the-art, have been custom-designed for speed [2].

TABLE 1. Carry chain 8-bit/32-bit N-sorter speed vs. 2-sorter N-network.

N	N Srter Prop Delay 8-bit	N Srter Prop Delay 32bit	2srtr Ntwrk Prop Delay 8-bit	2srtr Ntwrk Prop Delay 32bit	N Srter Estim Spdup	N Srter True Spdup 8-bit	N Srter True Spdup 32bit
4	1.128	1.297	2.617	3.154	3	2.32	2.43
5	1.203	1.380	4.297	5.192	5	3.57	3.76
6	1.303	1.434	4.319	5.214	5	3.31	3.64
7	1.335	1.469	5.148	6.222	6	3.86	4.24
8	1.522	1.656	5.148	6.222	4	3.38	3.76
9	1.617	1.751	5.999	7.252	4.7	3.71	4.14
10	1.630	1.760	6.021	7.274	4.7	3.69	4.13
12	1.671	1.804	6.872	8.304	5.3	4.11	4.60
16	1.891	2.024	7.723	9.334	6	4.08	4.61

All propagation delay values are in nS.
N-sorter LBs in Series = 2 for 7- and smaller N-sorters ; otherwise = 3.
Estimated speedup = (2-Network LBs in Series) / (N-sorter LBs in Series)
True speedup = (2-Network Prop Delay) / (N-sorter Prop Delay)
Yellow rows highlight N-sorters only implemented in carry chain logic.

The last column in Table 1 lists the speedups of the carry chain N-sorters versus the fastest comparable sorting networks. In [1], the maximum single-stage N-sorter speedup was 3.56. Here, all of the N-sorters in the last column of Table 1, except for the 4-sorter, have a speedup higher than 3.56. The maximum speedups in this table are for the carry chain 12- and 16-sorters, which cannot efficiently be built using design methodologies that only use the LUT-MUXF7/8/9 Fig. 3 LB. Both 12-/16-sorter speedups are at least 4.60.

Table 2 shows the LUT resource usage for the carry chain N-sorters and sorting networks whose speed data are shown in Table 1. N-sorter LUT increase ratio, which is (N-sorter LUTs/2-sorter network LUTs), is used to compare N-sorter LUT hardware usage to that of the associated 2-sorter network.

The fastest carry chain 8-sorter, whose data are found in Fig. 13 and Tables 1 and 2, is designed according to the middle flow diagram in Fig. 7. The LUT increase ratio is very low for this 8-sorter, even less than 1.0 for the 32-bit version, which is highlighted in blue in Table 2.

TABLE 2. Carry chain 8-bit/32-bit N-sorter LUT usage vs. 2-sorter N-network.

N	N Srter 8-bit LUT Usage	N Srter 32bit LUT Usage	2srtr Ntwrk 8-bit LUT Usage	2srtr Ntwrk 32bit LUT Usage	N Srter 8-bit Incrs Ratio	N Srter 32bit Incrs Ratio
4	120	480	100	400	1.20	1.20
5	200	800	180	720	1.11	1.11
6	300	1200	240	960	1.25	1.25
7	420	1680	320	1280	1.31	1.31
8	432	1344	380	1520	1.14	0.88
9	800	2960	500	2000	1.60	1.48
10	1287	4851	620	2480	2.08	1.96
12	1991	7535	800	3200	2.49	2.35
16	4860	17100	1220	4880	3.98	3.50

N-sorter LUT increase ratio = (N-sorter LUTs) / (2-Network LUTs)
Yellow rows highlight N-sorters only implemented in carry chain logic.

The LUT increase ratios for the fastest carry chain 9- up to 16-sorters are not as low as the 8-sorter's. These sorters are designed using the bottom flow diagram in Fig. 7.

B. SINGLE-STAGE N-MAX SYNTHESIS RESULTS

Propagation delay curves are shown for 4 types of single-stage N-max filters in Fig. 14. Once again, the curves and additional data are representative of comparable N-min filters as well.

The purple top curve contains data for the original baseline designs, which use only the Fig. 3 LB and whose design flows are generally shown in Fig. 4. The data points for 2-max and 3-max filters, like the 2-sorter and 3-sorter data points in Fig. 13, are only shown in the top curve.

The blue curve in Fig. 14 shows data for ge_L_R Direct N-max filters, which were previously discussed in Section III-F in [1]. The ge_L_R Direct 4-, 5-, and 6-max filters only use 2 series LBs, and Fig. 14 shows that these are the fastest N-max filters for these filter sizes.

The Fig. 14 green curve shows data for larger LUT-MUXF7/8/9 N-max filters that use A/B group product term splitting. These are the fastest LUT-MUXF7/8/9 N-max filters for 9 or more input values. These N-max filters are discussed in more detail in Appendix A.

The red curve in Fig. 14 shows the propagation delays for carry chain N-max filters. These are the fastest 7- and larger N-max filters, and they include much larger, but still fast, N-max filters.

As listed in Table 3, which contains speed data for single-stage and network N-max filters, a single-stage 125-max 5 × 5x5 CNN video max pooling filter [18] has a propagation delay is 2.075 nS, which is a speedup of 3.43 relative to the 7.113 nS delay of the analogous 7-stage 2-max filter network. This speedup value of 3.43 is the largest speedup for the single-stage carry chain N-max filters.

Table 4 shows the LUT resource usage for the N-max filters whose speed data is listed in Table 3. The N-max

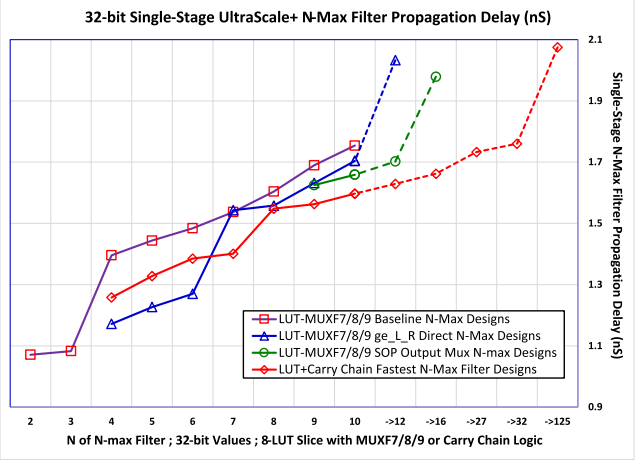


FIGURE 14. Single-stage N-max filter 32-bit propagation delay.

TABLE 3. Carry chain 8-bit/32-bit N-max filter speed vs. 2-max N-network.

N	N-Max Flter Prop Delay 8-bit	N-Max Flter Prop Delay 32bit	2-max Ntwrk Prop Delay 8-bit	2-max Ntwrk Prop Delay 32bit	N Max Estim Spdup	N Max True Spdup 8-bit	N Max True Spdup 32bit
4	1.088	1.258	1.720	2.078	2	1.58	1.65
5	1.158	1.328	2.526	3.063	3	2.18	2.31
6	1.254	1.385	2.526	3.063	3	2.01	2.21
7	1.267	1.401	2.526	3.063	3	1.99	2.19
8	1.373	1.548	2.548	3.085	2	1.86	1.99
9	1.388	1.563	3.354	4.070	2.7	2.42	2.60
10	1.469	1.597	3.354	4.070	2.7	2.28	2.55
12	1.498	1.629	3.354	4.070	2.7	2.24	2.50
16	1.534	1.662	3.376	4.092	2.7	2.20	2.46
27	1.602	1.733	4.182	5.077	3.3	2.61	2.93
32	1.629	1.760	4.204	5.099	3.3	2.58	2.90
64	1.845	1.975	5.032	6.106	4	2.73	3.09
125	1.945	2.075	5.860	7.113	4.7	3.01	3.43

All propagation delay values are in nS.
N-max LBs in Series = 2 for 7- and smaller N-max filters ; otherwise = 3.
Estimated speedup = (2-Network LBs in Series) / (N-max LBs in Series)
True speedup = (2-Network Prop Delay) / (N-max Prop Delay)
Yellow rows highlight N-max filters only implemented in carry chain logic.

LUT increase ratio grows significantly for the largest N-max filters. The LUT resource usage for the 125-max filter, 127969, is 10.8% of the available xcvu9p LUT resources.

C. N-MAX FILTER NETWORK SYNTHESIS RESULTS

The 2-stage 1024-max network, using a total of 33 single-stage 32-max filters, has a worst case propagation delay of 3.557 nS. The existing state-of-the-art 1024-max sorter, using 10 stages of 2-max filters, has a propagation delay of 10.134 nS. The 2-stage 1024-max network therefore has a speedup of $\mathbf{10.134/3.557=2.85}$.

The 1024-max network using 32-max filters uses 300762 LUTs, which are 25.4% of the available xcvu9p LUTs. This 2-stage network has a N-max LUT increase ratio of 6.125 versus the 10-stage 2-max filter network.

VI. LOGIC BLOCKS IN THE LATEST FPGA FAMILIES

The logic blocks shown in Figs. 3 and 5 are found in the FPGA products in the AMD-Xilinx UltraScale and UltraScale+ families. These two logic blocks have been used

TABLE 4. Carry chain 8-bit/32-bit N-max filter LUTs vs. 2-max N-network.

N	N-max Flter 8-bit LUT Usage	N-max Flter 32bit LUT Usage	2-max Ntwrk 8-bit LUT Usage	2-max Ntwrk 32bit LUT Usage	N-max Flter 8-bit Incrs Ratio	N-max Flter 32bit Incrs Ratio
4	48	192	36	144	1.33	1.33
5	72	288	48	192	1.50	1.50
6	100	400	60	240	1.67	1.67
7	132	528	72	288	1.83	1.83
8	160	592	84	336	1.90	1.76
9	202	754	96	384	2.10	1.96
10	240	900	108	432	2.22	2.08
12	336	1272	132	528	2.55	2.41
16	656	2480	180	720	3.64	3.44
27	1742	6578	312	1248	5.58	5.27
32	2418	9114	372	1488	6.50	6.13
64	8944	33664	756	3024	11.83	11.13
125	33961	127969	1488	5952	22.82	21.50

N-max LUT increase ratio = (N-max LUTs) / (2-max network LUTs)
Yellow rows highlight N-max filters only implemented in carry chain logic.

to design all of sorting devices defined here and in [1]. However, AMD-Xilinx has introduced a new FPGA, Versal ACAP, with significantly different logic structures. It is worthwhile to perform an initial analysis on how well the new logic blocks will support the N-sorter/N-filter design system described in [1] and expanded on here.

It is also worthwhile to analyze how well the N-sorter/N-filter design system can be implemented in the Intel Agilex FPGA logic blocks, particularly since the Agilex carry chain logic differs a great deal from both the UltraScale+ and the Versal ACAP carry chain structures. Carry chain logic in the new families is discussed in Section VI-A, and standard FPGA logic blocks, such as the one shown in Fig. 3, are analyzed in Section VI-B.

A. CARRY CHAIN LOGIC IN THE LATEST FPGAs

The type of carry chain logic shown in Fig. 5 is here called Carry Out MUX (COM) logic. The segment CO signal is the output of a 2-to-1 mux. The two data inputs to the MUX are the segment CI signal and a variable DI segment input. The select line for the 2-to-1 mux is driven from the output of the segment 6-input LUT.

The carry chain logic in the AMD-Xilinx Versal ACAP FPGAs continues to be a COM type, but with significant modifications [21]. The output mux select line is the output of a 4-input LUT portion of the full segment 6-input LUT. The DI signal is normally a LUT input that is not an input to the 4-input LUT that drives the select line.

Because of the 4 input limitation for the Versal ACAP output mux select line, the Versal ACAP N-sorter/N-filter carry chain design methodology consistently underperforms the methodology using the UltraScale+ Fig. 5 carry chain. This is shown in the first two blue columns in Table 5.

The Versal ACAP data in Table 5 is the result of a simple analysis of the Versal ACAP carry chain logic. There are additional features of Versal ACAP logic that may improve its results compared to UltraScale+. One very interesting Versal ACAP feature is the CASC signal, which is directly routed

TABLE 5. Carry chain N-sorter/N-filter comparisons of 3 carry chain types.

Uscale+ Design Figure	Segment Attribute	Uscale+ Carry Chain	V ACAP Carry Chain	Agilex Carry Chain
6	S eq. variables	6	4	3
6	Largest N-sorter	7–	5–	4–
10	Prod Terms/Seg	3	2	2
11	Groups/Segment	6	4	3
9	Pairs/Segment	3	2	3
12	Comparisons/Seg	6	4	6

UltraScale+ : DI = input bit in Figs. 6, 10, and 11
UltraScale+ : DI = 1 in Fig. 9 – Simple OR equation
UltraScale+ : DI = 0 in Fig. 12 – Simple AND equation

from the main LUT output to the LUT directly above the first LUT. Another new feature in Versal ACAP is that the carry chain logic is organized in a LUT pair, and the CO signal from the pair comes from a 4-to-1 multiplexer, with 2 control signals, and data inputs from both paired LUTs.

The Agilex carry chain logic found in its LUT structure consists of two full adders in a Ripple Carry Adder (RCA) sequence [22]. There is no carry out mux, and no separate DI signal. Two of the LUTs 6 inputs only support Adder0, two only support Adder1, and two are common to both adders. Each adder has 2 data inputs, driven from separate 4-input LUT sections of the 6-input structure.

The N-sorter/N-design carry chain design outputs use a CO signal, but it is not clear from the Agilex documentation how often CO signals are routed out of the logic blocks. However, a CO signal can be sent to the SUM output of the subsequent adder by forcing its two data inputs to opposite Boolean values, and all SUM outputs can be routed out to the interconnect.

The first four lines in Table 5 contain data for UltraScale+ carry chain designs in which the DI input is an input data bit. The Agilex carry chain logic does not efficiently support this type of N-sorter/N-filter design, as can be seen in last column of the table. However, the last two lines in Table 5 target designs in which the DI input is a constant 0 or 1. Since a full adder is able to generate a 0 or 1 based on its data input values, the Agilex segment LUT is able to use its 6 inputs as efficiently as UltraScale+ for the Figs. 9 and 12 designs in the last 2 table rows.

B. STANDARD LOGIC BLOCKS IN THE LATEST FPGAs

AMD-Xilinx has historically referred to their FPGA logic block as a CLB. The 8-LUT UltraScale+ group shown in Figs. 3 and 5 is also referred to as a slice. In UltraScale+, the CLB and 8-LUT slice are identical structures.

The Versal ACAP CLB is changed rather dramatically [21]. An 8-LUT group is still called a slice, but there are now 4 slices per CLB. The Fig. 3 MUXF7, MUXF8, and MUXF9 2-to-1 hardware multiplexers have been removed. In their place, the CLB now has local routing, intended to provide fast paths between LUTs in the CLB.

The UltraScale+ In_X_goes_to_Out_Y signals for 8- to 10-sorters utilized Fig. 3 MUXF7/8/9 multiplexers, so these type of signals will need to use additional LUTs and local routing in Versal ACAP if they are still to be as fast. The output of an UltraScale+ MUXF7/8 block requires 4 LUT outputs and 3 control signals as inputs, too many to be implemented in a 6-input LUT.

However, the two MUXF7 control signals can be combined into a single signal, using a LUT with all 3 control signals as inputs. This control signal reduction happens in parallel with the operations in the 4 base LUTs, so there is no speed impact. The MUXF7/8 equivalent operation is now completed in a LUT whose 6 inputs are the outputs of the 4 base LUTs, and the 2 control signals. Therefore, 6 LUTs are required for the equivalent of a MUXF7/8 structure in a Versal ACAP CLB. The slowest signals travel through 2 series LBs, but the routing between the series LBs is fast local routing.

The slice equivalent in the Intel Agilex devices is called a LAB [22]. A LAB has 10 LUTs, versus the 8 LUTs in a AMD-Xilinx slice. The Agilex LAB also has local routing between the LUTs in the LAB. The equivalent of a MUXF7/8 operation in a LAB is similar to that in Versal ACAP. The outputs of four base inputs would be locally routed to an additional LUT, along with the two control signals after control signal reduction in a LAB LUT. This six LUT group allows for the 4 remaining LAB LUTs to be utilized for other operations.

VII. CONCLUSION

The single-stage N-sorter/N-filter design system that the authors previously published is expanded here, and then applied to a hardware type not previously utilized, FPGA carry chain logic. The carry chain logic targeted here is found in the same FPGA that was used for the previous analysis, and the same software is used to implement carry chain logic single-stage devices. The new carry chain devices are compared to the previous single-stage devices, and to the multistage sorting networks that had been the fastest state-of-the-art FPGA sorting devices.

The carry chain logic N-sorters are shown to be considerably faster than the previous single-stage N-sorters, and much faster than comparable sorting networks. Also, significantly larger and still fast N-sorters, such as a carry chain 16-sorter, have now been implemented and characterized.

Perhaps the most striking aspect of carry chain logic is its ability to enable fast and dramatically larger single-stage N-max/N-min filters, which can aid video max pooling in CNN applications. These large N-max/N-min filters can then be used in simple N-max/N-min sorting networks, which are able to process large input lists much faster than the previous state-of-the-art networks, consisting of many serial stages which use 2-max/2-min filters.

This study has focused on the logic blocks found in the products of two currently popular FPGA families. Newer FPGA families have been introduced, and their ability to accommodate efficient N-sorter/N-filter design has been discussed. One possible focus for future work is to adapt this N-sorter/N-filter design system to one or more of the

latest FPGA product families, and to compare the new FPGA results to the results obtained in this study.

APPENDIX A
LUT-MUXF7/8/9 SOP MUX DESIGNS

As shown in Fig. 4's flow diagrams for the baseline LUT-MUXF7/8/9 N-sorters, all of the input signal bits were inputs to the last Output MUX Block. For 5- and larger N-sorters, MUXF 2-to-1 multiplexers are required in the last LB, and ORs of multiple In_X_goes_to_Out_Y signals are typically needed as select lines for the 2-to-1 multiplexers.

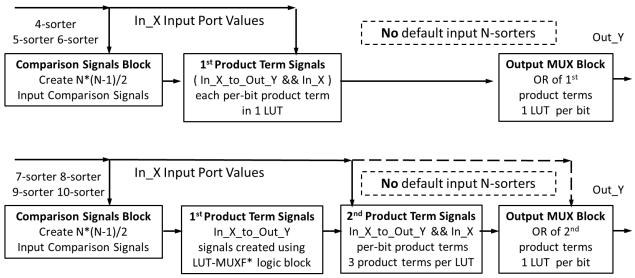


FIGURE 15. LUT-MUXF7/8/9 SOP output mux signal flow diagrams.

For the new LUT-MUXF7/8/9 design flows shown in Fig. 15, the input signal bits are normally the inputs to the 2nd-to-last series LB, and the final Output MUX Block consists of a single 6-input LUT, which enables faster N-sorters. Another feature that enables faster speeds for these N-sorters is that each LUT output in the 2nd-to-the-last LB is simply routed to one LUT input in the final LB, as shown for the 6-sorter in Fig. 16.

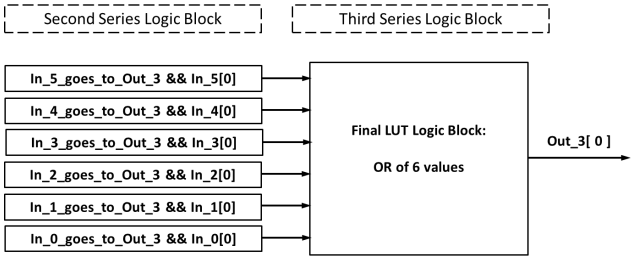


FIGURE 16. Last 2 stages of LUT-MUXF7/8/9 6-sorter SOP output mux.

Since a 6-sorter's In_X_goes_to_Out_Y signals use 5 comparison signals, (**In_X_goes_to_Out_3** && **In_X[0]**) product terms in Fig. 16 are able to be implemented in a 6-input LUT in the next-to-the-last LB. The 3rd and final LUT simply ORs the 6 product terms and completes the SOP output mux equation. This 6-sorter is the one LUT-MUXF7/8/9 N-sorter that is (slightly) faster than the comparable carry chain N-sorter, as shown in the blue curve Fig. 13.

Use of SOP output mux equations, along with A/B group product term splitting, enable the fastest and largest LUT-MUXF7/8/9 N-max filters for 9 or more input values, whose data are plotted in the green curve in Fig. 14. Fig. 17 shows the 3rd, next-to-last stage of the 12-max design.

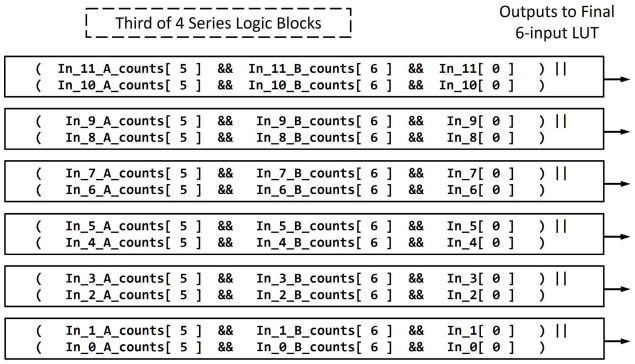


FIGURE 17. LUT-MUXF7/8/9 12-max 3rd of 4 stages using A/B groups.

The A/B count signals are created in the 2nd stage. Once again, the final stage consists of a single LUT which ORs all 6 of the LUT outputs from the 3rd stage. Although these are the fastest and largest LUT-MUXF7/8/9 N-max filters, Fig. 14 shows that comparable carry chain N-max filters are faster and much larger carry chain filters can be implemented.

REFERENCES

[1] R. B. Kent and M. S. Pattichis, "Design, implementation, and analysis of high-speed single-stage N-sorters and N-filters," *IEEE Access*, vol. 9, pp. 2576–2591, 2021.

[2] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3. London, U.K.: Pearson Education, 1997.

[3] M. Codish, L. Cruz-Filipe, T. Ehlers, M. Müller, and P. Schneider-Kamp, "Sorting networks: To the end and back again," *J. Comput. Syst. Sci.*, vol. 104, pp. 184–201, Sep. 2019, doi: 10.1016/J.JCSS.2016.04.004.

[4] I. Skliarova and V. Sklyarov, "Reconfigurable devices and design tools," in *FPGA-BASED Hardware Accelerators*. Cham, Switzerland: Springer, 2019, pp. 1–38.

[5] K. E. Batcher, "Sorting networks and their applications," in *Proc. Spring Joint Comput. Conf.*, vol. 2. New York, NY, USA, 1968, pp. 307–314, doi: 10.1145/1468075.1468121.

[6] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on FPGAs," *VLDB J.*, vol. 21, no. 1, pp. 1–23, Feb. 2012, doi: 10.1007/S00778-011-0232-Z.

[7] H.-C. Hsieh, W. S. Carter, J. Ja, E. Cheung, S. Schreifels, C. Erickson, P. Freidin, L. Tinkey, and R. Kanazawa, "Third-generation architecture boosts speed and density of field-programmable gate arrays," in *Proc. IEEE Proc. Custom Integr. Circuits Conf.*, May 1990, p. 31, doi: 10.1109/CICC.1990.124841.

[8] J. Poldre and K. Tammemae, "Reconfigurable multiplier for Virtex FPGA family," in *Proc. Int. Workshop Field Program. Log. Appl.*, Glasgow, Scotland, Sep. 1999, pp. 359–364.

[9] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Efficient synthesis of compressor trees on FPGAs," in *Proc. Asia South Pacific Design Autom. Conf.*, Jan. 2008, pp. 138–143, doi: 10.1109/ASPDAC.2008.4483927.

[10] A. Petkovska, G. Zgheib, D. Novo, M. Owaida, A. Mishchenko, and P. Ienne, "Improved carry chain mapping for the VTR flow," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Dec. 2015, pp. 80–87, doi: 10.1109/FPT.2015.7393133.

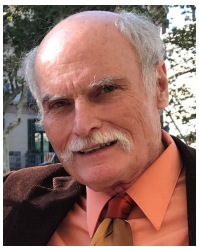
[11] T. B. Preuser, "Generic and universal parallel matrix summation with a flexible compression goal for Xilinx FPGAs," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–7.

[12] M. Kumm and J. Kappauf, "Advanced compressor tree synthesis for FPGAs," *IEEE Trans. Comput.*, vol. 67, no. 8, pp. 1078–1091, Aug. 2018, doi: 10.1109/TC.2018.2795611.

[13] S. R. Ganesan, "Softpal implementation and mapping technology for FPGAs with dedicated resources," U.S. Patent 7 111 273 B1, Sep. 19, 2006.

[14] K. Cui, Z. Ren, X. Li, Z. Liu, and R. Zhu, "A high-linearity, ring-oscillator-based, Vernier time-to-digital converter utilizing carry chains in FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 64, no. 1, pp. 697–704, Jan. 2017, doi: 10.1109/TNS.2016.2632168.

[15] T. B. Preusser and R. G. Spallek, "Enhancing FPGA device capabilities by the automatic logic mapping to additive carry chains," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2010, pp. 318–325, doi: 10.1109/FPL.2010.70.
[16] Z. Chu, X. Tang, M. Soeken, A. Petkovska, G. Zgheib, L. Amarù, Y. Xia, P. Ienne, G. De Micheli, and P.-E. Gaillardon, "Improving circuit mapping performance through MIG-based synthesis for carry chains," in *Proc. Great Lakes Symp. (VLSI)*, May 2017, pp. 131–136, doi: 10.1145/3060403.3060432.
[17] R. Senhadji-Navarro and I. Garcia-Vargas, "Mapping arbitrary logic functions onto carry chains in FPGAs," *Electronics*, vol. 11, no. 1, p. 27, Dec. 2021, doi: 10.3390/ELECTRONICS11010027.
[18] A. E. U. Cerna, L. Jing, C. W. Good, D. P. vanMaanen, S. Raghunath, J. D. Suever, C. D. Nevius, G. J. Wehner, D. N. Hartzel, J. B. Leader, A. Alsaid, A. A. Patel, H. L. Kirchner, J. M. Pfeifer, B. J. Carry, M. S. Pattichis, C. M. Haggerty, and B. K. Fornwalt, "Deep-learning-assisted analysis of echocardiographic videos improves predictions of all-cause mortality," *Nature Biomed. Eng.*, vol. 5, no. 6, pp. 546–554, Jun. 2021.
[19] *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*, Standard 1800, 2017.
[20] *UltraScale Architecture Configurable Logic Block, UG574 (v1.5)*, Xilinx Inc., San Jose, CA, USA, Feb. 2017. Accessed: Oct. 8, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf
[21] *Versal ACAP Configurable Logic Block Architecture Manual, AM005 (v1.1)*, Xilinx Inc., San Jose, CA, USA, Apr. 2021. Accessed: Sep. 14, 2021. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/architecture-manuals/am005-versal-clb.pdf
[22] *Intel AgilexT Logic Array Blocks and Adaptive Logic Modules User Guide, UG-20204*, Intel Corporation, Santa Clara, CA, USA, Nov. 2019. Accessed: Sep. 18, 2021. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/667015?fileName=ug-ag-lab-683577-667015.pdf



ROBERT B. KENT (Life Member, IEEE) received the B.S. degree in physics from the University of Notre Dame, Notre Dame, IN, USA, in 1970, and the M.S. degree in electrical engineering from The University of Utah, Salt Lake City, UT, USA, in 1983. He is currently pursuing the Ph.D. degree with the Electrical and Computer Engineering Department, The University of New Mexico.

He has worked for various semiconductor companies: National Semiconductor, from 1983 to 1990; Intel Corporation, from 1990 to 1998; Philips Semiconductor, from 1998 to 1999; and Xilinx, Inc., from 1999 to 2011. He then worked as an independent contractor, also in the semiconductor field, from 2012 to 2017. His main research interests include the design of single-stage N-sorters and N-filters in hardware, particularly in FPGAs; and the use of these sorters and filters in sorting networks or other hardware sorting systems.

MARIOS S. PATTICHIS (Senior Member, IEEE) received the B.Sc. degree (Hons.) in computer sciences, the Bachelor of Arts degree (Hons.) in mathematics and a minor in electrical engineering, the M.S. degree in electrical engineering, and the Ph.D. degree in computer engineering from The University of Texas at Austin, in 1991, 1993, and 1998, respectively.

He is currently a Professor with the Department of Electrical and Computer Engineering, The University of New Mexico (UNM). His current research interests include image and video processing, video communications, dynamically reconfigurable computer architectures, and biomedical image analysis. He holds the 2019–2022 ECE Gardner Zemke Professorship for teaching.

Dr. Pattichis was a fellow of the Center for Collaborative Research and Community Engagement with the UNM College of Education, from 2019 to 2020. He was a recipient of the 2016 Lawton-Ellis and the 2004 Distinguished Teaching Awards from the Department of Electrical and Computer Engineering, UNM. For his development of the digital logic design labs at UNM, he was recognized by the Xilinx Corporation, in 2003; and by the UNM School of Engineering's Harrison Faculty Excellence Award, in 2006. He was the Lead PI and a Board Member of the Configurable Space and Microsystems Innovations and Applications Center (COSMIAC) at UNM. At UNM, he also serves as the Director for the Image and Video Processing and Communications Laboratory (ivPCL). He was the General Chair of the 2008 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI), where he also served as the General Co-Chair, in 2020. He is currently serving as a Guest Associate Editor for the Special Issue on Large Scale Video Analytics for Clinical Decision Support of IEEE JOURNAL OF BIOMEDICAL AND HEALTH INFORMATICS and the Special Issue on Teaching and Learning Mathematics and Computing in Multilingual Contexts of *Teachers College Record*. He has served as a Senior Associate Editor for the IEEE TRANSACTIONS ON IMAGE PROCESSING and IEEE SIGNAL PROCESSING LETTERS, an Associate Editor for IEEE TRANSACTIONS ON IMAGE PROCESSING and IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, and a Guest Associate Editor for two additional special issues published in the IEEE TRANSACTIONS ON INFORMATION TECHNOLOGY IN BIOMEDICINE and another special issue published in *Biomedical Signal Processing and Control*.