High-Performance and Hardware-Efficient Odd-Even Based Merge Sorter

Elsayed A. ELSAYED†,††a, Nonmember and Kenji KISE†b, Member

SUMMARY Data sorting is an important operation in computer science. It is extensively used in several applications such as database and searching. While high-performance sorting accelerators are in demand, it is very important to pay attention to the hardware resources for such kind of high-performance sorters. In this paper, we propose three FPGA based architectures to accelerate sorting operation based on the merge sorting algorithm. We call our proposals as WMS: Wide Merge Sorter, EHMS: Efficient Hardware Merge Sorter, and EHMSp: Efficient Hardware Merge Sorter Plus. We target the Virtex UltraScale FPGA device. Evaluation results show that our proposed merge sorters maintain both the high-performance and cost-effective properties. While using much fewer hardware resources, our proposed merge sorters achieve higher performance compared to the state-of-the-art. For instance, with 256 sorted records are produced per cycle, implementation results of proposed EHMS show a significant reduction in the required number of Flip Flops (FFs) and Look-Up Tables (LUTs) to about 66% and 79%, respectively over the state-of-the-art merge sorter. Moreover, while requiring fewer hardware resources, EHMS achieves about 1.4x throughput from the state-of-the-art merge sorter. For the same number of produced records, proposed WMS also achieves about 1.6x throughput improvement over the state-of-the-art while requiring about 81% of FFs and 76% of LUTs needed by the state-of-the-art sorter.

key words: FPGA, sorting, hardware merge sorter, odd-even sort, merge network

1. Introduction and Background

Sorting is a fundamental operation and widely used in several practical applications such as searching and database and it is highly desirable to accelerate it. Many sorting accelerators have been developed by researchers recently. Some of these accelerators rely on software solutions to accelerate the sorting process. Others rely on hardware such as FPGA based architectures. For example, CPU based optimized sorting algorithms have been provided in [1] and [2], while GPU based accelerators have been presented in [3] and [4]. For hardware based sorting accelerators, they have been used in spot and widely researched in many recent studies including [5]–[11], and [12]. Compared to CPU/GPU based sorting accelerators, FPGA based accelerators provide higher performance. Therefore, we will focus on FPGA based sorting architecture.

There is a variety of sorting algorithms including insertion, bubble, quick, and merge sort. Due to its high level of parallelism, merge sort algorithm has been intensively used in FPGA based sorting accelerators. Usually, the input data to be sorted is input to the sorting accelerator as a set of records, where each record combines two parts: key and data. Records are sorted based on their key field and we call the record that has the smallest key as smallest record. In merge sort, two sorted sequences of N records each are merged to one sorted sequence of 2N records. We define the hardware sorting accelerator which relies on the merge sort as Hardware Merge Sorter. Also, we use a parameter E to indicate the number of output sorted records per clock cycle from a merge sorter. The value of E is important to determine the performance of a merge sorter.

Figure 1 shows the architecture of basic hardware merge sorter that has been widely used in most recent researches, where E = 4 in the figure. There are three major components: input FIFOs, selector logic, and merge network. Two FIFOs are used to store the input sorted records from two input sequences. We define the E−records at the head of $FIFO_X$ as BundleX. Selector logic is responsible for fetching one bundle each clock cycle from either $FIFO_A$ or $FIFO_B$. The head records of BundleA and BundleB are compared and the bundle containing the smallest head is selected to be fetched, assuming the ascending order. Then, the dequeued bundles are sent to the merge network to be merged and output. There are two common techniques to build the merge network: bitonic merging and odd-even merging [13]. In this paper, we rely on the odd-even approach to build our merge network.

Recent researches have proposed a variety of FPGA
based merge sorters, each one of them has its own shortcomings when compared with others. For example, the high-performance merge sorter proposed in [8] achieved 3.13x higher throughput than its predecessor merge sorter proposed in [7], but it requires 7.26x higher FPGA registers.

In this paper, we introduce three architectures for hardware merge sorter and our focus will be to achieve a significant reduction in the required hardware resources while maintaining the high-performance property.

The most recent hardware based merge sorters are introduced in [11] and [12]. Authors of [11] use a modified bitonic merging technique to build their proposed merge sorter. While those of [12] rely on the odd-even merging approach. Both [11] and [12] use the predecessor hardware merge sorter proposed in [10] as a competitor to evaluate their proposed merge sorters. Since we focus on the odd-even approach, we use the hardware merge sorter architecture proposed in [12] as a baseline and competitor for our evaluation. It is called Very massive Merge Sorter or shortly VMS and it is an extension of the merge sorter proposed in [10].

This article is an extension of our previous work [14]. We provide more detailed analysis and evaluation results. The tie record issue that appeared in [10] is discussed. We present a solution for it and prove that our proposals are free of this issue. We add more evaluation results to compare our proposed two merge sorters, shown in [14], against the state-of-the-art work [12]. Evaluation results include larger merge sorters that output not only from 8 to 64 records per clock cycle but also 128, 256, and 512 records.

We examine our proposed merge sorters on various numbers of output records from 8 to 512. We find that in the case of large merge sorter, our proposed architectures not only use much fewer hardware resources than [12] but also achieve much better performance. Therefore, proposals are more efficient than the previous work and can be targeted not only as hardware-saver merge sorters but also as high-performance merge sorters. We also propose a third merge sorter as an extension of the previously proposed two merge sorters. We present the architecture and evaluation result of the new proposed merge sorter and compare it against other approaches. Besides, we use Vivado version 2019.2 instead of Vivado version 2017.2. Finally, we target the large Virtex UltraScale xcvu440-flga2892-3-e FPGA device instead of Virtex-7 xc7vx980t-2 to allow enough room for the place and route process.

Our key contributions of this paper are as follows:

- We propose three architectures for hardware merge sorter that efficiently use fewer hardware resources compared to the state-of-the-art merge sorter.
- We show that our proposed merge sorters not only use fewer hardware resources but also achieve a higher performance than the state-of-the-art research.
- We show the implementation of our proposed architectures and the detailed structure of important components.
- We do a performance evaluation for our proposed merge sorters and compare them with the state-of-the-art research.

2. Architecture of Proposed Hardware Merge Sorters

We propose three hardware-optimized architectures for merge sorting accelerator, we call them as Wide Merge Sorter (WMS), Efficient Hardware Merge Sorter (EHMS), and Efficient Hardware Merge Sorter Plus (EHMSP). In this section, we present the key idea of WMS, EHMS, and EHMSP and show the implementation for each one of them. Moreover, a behavior example is shown for each merge sorter. As mentioned in Sect. 1, each record of the data set to be sorted is composed of two parts: key and data. When two records or more have the same key field, we call them as tie records. If the input data set contains some tie records, the output sequence may be wrong in some merge sorters. For simplicity, the behavior examples shown in this section do not have any tie records. The tie record issue and its solution will be explained in detail in Sect. 4.
The architecture of MMS and VMS, in case of \( E = 4 \), is shown in Fig. 2 (a) and Fig. 2 (b), respectively. They have a similar merger network architecture. However, MMS is built based on the bitonic merger while VMS is built based on the odd-even merger. We focus on VMS architecture. Two pipelined mergers are combined inside its merge network. We define the merger that has two inputs as \((m, n)\) merger, where \( m \) is the number of records within the first input and \( n \) is the number of records within the second input. Therefore, we call each merger inside the merge network shown in Fig. 2 (b) as \((E, E)\) merger. The first merger receives the values of registers \( R_1 \) and \( R_2 \) as its inputs. The parameter \( ER \) in the figure indicates the width of each input which is \( E \)-records. \( R_1 \) and \( R_2 \) are merged and the largest \( E \)-records of the output are input to the second merger. The other input of the second merger is the output of multiplexer which is delayed into shift register for some clock cycles equal to the depth of the first pipelined merger.

We have observed that the three inputs to these two mergers are in fact two inputs only. As explained in [12], \( R_1 \) and \( R_2 \) are used to store the last dequeued bundles from sequence A and sequence B. Thus, each register of \( R_1 \) and \( R_2 \) contains a unique bundle that was fetched either from sequence A or sequence B. It is not possible for \( R_1 \) and \( R_2 \) to have bundles from the same sequence. Moreover, the multiplexer selects one bundle either from sequence A or sequence B. Therefore, we have two possibilities for the inputs to mergers based on multiplexer selection: two \( E \)-records from sequence A and one \( E \)-records from sequence B or two \( E \)-records from sequence B and one \( E \)-records from sequence A. As each input sequence contains sorted records, we can consider the inputs in either case as one input with width 2\( E \)-records and the other input with width \( E \)-records. Thus, we can replace the two \((E, E)\) mergers inside the merge network of VMS with one \((2E, E)\) merger. This is the key idea of our proposed WMS which is shown in Fig. 4 for \( E = 4 \), where input named MUX is the output of multiplexer and it is determined by the selector logic to be bundle \( D_A \) from sequence A or bundle \( D_B \) from sequence B. We define \( D_X \) as the bundle at the head of \( FIFO_X \). The records on the right side of the vertical dash line are already dequeued from FIFOs while those on the left side are still inside FIFOs. At any cycle, the \((2E, E)\) merger will consider its two inputs as either \((\{MUX, R_A\}, R_B)\) or \((\{MUX, R_B\}, R_A)\).

There are three advantages for our proposed WMS. First, WMS uses only one \((2E, E)\) merger inside the merge network. Second, there is no need for the shift register shown in Fig. 2 (b). Third, latency of the output sorted records in WMS is smaller than that in VMS. Figure 3 (a) depicts the architecture of our proposed merge sorter WMS. The head of bundles \( D_A \) and \( D_B \) are passed to the selector logic, where each head contains the smallest record of its own FIFO. Selector logic compares the two heads and generates the control signals used to manage the selection of multiplexers and remove a bundle from the FIFO containing the smallest head. The selector logic architecture will be explained in Sect. 3.1.

In WMS architecture, at clock cycle \( t \), the output of merge network is the middle \( E \)-records of the sorted \( 3E \)-records because the records inside \( R_A \) and \( R_B \) have already been merged and the smallest \( E \)-records among them have been output during clock cycle \( t - 1 \). Moreover, we follow VMS merge sorter and build our \((2E, E)\) merger based on the odd-even merge sorting algorithm, as discussed in Sect. 3.2.

Figure 3 (b) shows a behavior example of proposed WMS in case of \( E = 4 \). Two input sorted sequences are input to WMS, merged, and sorted. The first sequence contains \\{40 38 35 34, 29 25 22 15, [12 9 5 2]\} and the second sequence contains \\{45 41 39 37, [33 32 27 23], [19 13 7 4]\}. Let \( D_X^t \) to be the head bundle of \( FIFO_X \) at clock cycle \( t \).
At clock cycle 1, the heads \([2, 4]\) are compared and \(D_1^1\) is selected. Based on this selection, \(INP_1\) of the merger will be \(((D_1^1, R_A) = [(12 9 5 2), (0 0 0 0)])\) and \(INP_2\) will be \((R_B = [0 0 0 0])\). The output at this cycle will be the middle sorted 4-records \((0 0 0 0)\). Note that these zeros will be ignored and will not be counted as a valid output.

At clock cycle 2, \(R_A\) is updated to the value of \(D_1^1\) and the selector logic selects \(D_2^2\). Therefore, \(INP_1\) of the merger will be \(((D_2^2, R_A) = [(19 13 7 4), (0 0 0 0)])\) and \(INP_2\) will be \((R_A = [12 9 5 2])\). The middle sorted 4-records \((7 5 4 2)\) will be output.

At clock cycle 3, \(R_B\) is updated to the value of \(D_2^2\) while \(R_A\) keeps its old value. \(D_2^2\) is selected. \(((D_1^2, R_A) = [(29 25 22 15), (12 9 5 2)])\) and \((R_B = [19 13 7 4])\) are input to \(INP_1\) and \(INP_2\), respectively. 4-records \((115 13 12 9)\) are the output at this cycle. The process continues in the same manner as shown in Fig. 3 (b) until both \(FIFO_A\) and \(FIFO_B\) becomes empty. It is clear from Fig. 3 (b) that the merging and sorting processes are performed correctly.

2.2 Proposed Efficient Hardware Merge Sorter (EHMS)

To further reduce the hardware resources of our proposed merge sorter WMS, we propose EHMS as an extension of WMS.

The key idea of proposed EHMS is that instead of selecting one bundle to be dequeued from \(FIFO_A\) or \(FIFO_B\), we select two bundles, as shown in Fig. 6. \(R_A\) and \(R_B\) contain the largest dequeued \(E\)-records from \(FIFO_A\) and \(FIFO_B\), respectively. \(MUX_1\) and \(MUX_2\) indicate the two bundles selected by the selector logic to be dequeued from \(FIFO_A/FIFO_B\), where the head record of \(MUX_1\) is smaller than the head record of \(MUX_2\). \(D_{xi}\) indicates the \(i^{th}\) head bundle of \(FIFO_X\).

As shown in Fig. 6, there are two possibilities for \(MUX_1\) and four possibilities for \(MUX_2\). \(MUX_1\) may be \(D_{A1}\) or \(D_{B1}\) while \(MUX_2\) may be one of \(D_{A2}, D_{B2}, D_{A3}\), or \(D_{B3}\). \(MUX_2\) is selected based on the selection of \(MUX_1\). For example, if \(D_{A1}\) from \(FIFO_A\) is selected to be \(MUX_1\), then \(MUX_2\) may be \(D_{A2}\) or \(D_{B1}\). But if \(D_{B1}\) is selected as \(MUX_1\), then \(MUX_2\) may be \(D_{A1}\) or \(D_{B2}\).

The merge network architecture of EHMS for \(E = 8\) is shown in Fig. 5 (a). The heads of four bundles \(D_{A1}, D_{A2}, D_{B1}\), and \(D_{B2}\) are passed to the selector logic. Then, selector logic determines the smallest two heads to dequeue their bundles. The dequeued bundles may be both from \(FIFO_A\), both from \(FIFO_B\), or mixed from the two FIFOs.

At any cycle \(t\), inputs of the merger are rearranged as two inputs to be one of three combinations:

- \([D_{A1}, R_A]\) and \([D_{B1}, R_B]\),
- \([D_{A2}, D_{A1}, R_A]\) and \([R_B]\),
- \([D_{B2}, D_{B1}, R_B]\) and \([R_A]\).

As shown in Fig. 5 (a), selector logic in EHMS controls many components and has many responsibilities than in WMS. It contains the critical path of the overall merge sorter and this is the reason behind the decrease in the maximum operating frequency of EHMS, as will be discussed in Sect. 5.2.3. The architecture of selector logic for EHMS will be discussed in Sect. 3.1.

Comparing Fig. 4 and Fig. 6 for the same number of output records per clock cycles (\(E = 4\)), we can say that there are three contributions for proposed EHMS over pro-
posed WMS. First, the width of registers $R_A$ and $R_B$ in EHMS is half of that in WMS. Second, the width of FIFOs needed to store sequences $A$ and $B$ in EHMS is also half of that in WMS. Third, all of merger inputs $R_A$, $R_B$, and $MUX_i$ in WMS have a width of $E$-records while all of merger inputs $RA$, $RB$, and $MUX\_1$ and $MUX\_2$ in EHMS have a width of $E/2$-records. Therefore, the merger inside WMS merges and sorts an input of $3E$-records while the merger inside EHMS merges and sorts only $2E$-records. Thus, the complexity of the EHMS merger is less than that of the WMS merger. However, there is one drawback for EHMS architecture that is complexity of the selector logic that selects two bundles in EHMS is larger than complexity of the selector logic that selects one bundle in WMS.

Figure 5 (b) shows a behavior example of proposed EHMS using the same input records used in Fig. 3 (b). At clock cycle 1, $RA$ and $RB$ are initialized to zeros. Four heads $\{15, 2\}$ and $\{23, 4\}$ are passed to the selector logic. As 2 and 4 are the smallest two heads, $D\_1\_A$ and $D\_1\_B$ are selected to be dequeued as $INP\_1$ and $INP\_2$, respectively. According to that, the merger rearranges its inputs as $(\{D\_1\_A, RA\} = \{125 23 22 19\}, \{0 0 0 0\})$ and $(\{D\_1\_B, RB\} = \{19 13 7 4\}, \{0 0 0 0\})$. As shown in Fig. 5 (a), the output of EHMS is the middle two sorted $E/2$-records. Therefore, the output will be $\{7 5 4 2\}, \{0 0 0 0\}$.

At clock cycle 2, $RA$ and $RB$ are updated to the value of $D\_2\_A$ and $D\_2\_B$, respectively. $D\_2\_A$ and $D\_2\_B$ are selected as inputs to the merger. The merger rearranges its inputs to be $(\{D\_2\_A, RA\} = \{29 25 22 15\}, \{12 9 5 2\})$ and $(\{D\_2\_B, RB\} = \{33 32 27 23\}, \{19 13 7 4\})$. The output at this cycle is $\{125 23 22 19\}, \{15 13 12 9\}$ and the process for merging and sorting is continued in the same manner.

2.3 Proposed Efficient Hardware Merge Sorter Plus (EHMSP)

As an extension for EHMS, we propose EHMSP. The key idea of proposed EHMSP is to increase the number of fetched bundles to three bundles instead of one bundle as in WMS or two bundles as in EHMS. As shown in Fig. 8, three bundles are selected from $FIFO\_A/FIFO\_B$ using three multiplexers $MUX\_1$, $MUX\_2$, and $MUX\_3$. Note that head record of the bundle selected by $MUX\_1$ is smaller than head record of the bundle selected by $MUX\_2$ and head record of the bundle selected by $MUX\_2$ is smaller than head record of the bundle selected by $MUX\_3$.

As shown in Fig. 8, $MUX\_1$ has two possibilities, $MUX\_2$ has four possibilities, and $MUX\_3$ has six possibilities. Candidates for $MUX\_1$ are $\{D\_A\_1, D\_B\_1\}$. Candidates for $MUX\_2$ are $\{D\_A\_1, D\_B\_1, D\_A\_2, D\_B\_2\}$. Candidates for $MUX\_3$ are $\{D\_A\_1, D\_B\_1, D\_A\_2, D\_B\_2, D\_A\_3, D\_B\_3\}$.

$MUX\_2$ makes a decision based on the selection of $MUX\_1$. $MUX\_3$ also makes a decision based on the selection of $MUX\_2$. For example, if $D\_A\_1$ from $FIFO\_A$ and $D\_B\_1$ from $FIFO\_B$ are selected to be $MUX\_1$ and $MUX\_2$, respectively, then $MUX\_3$ may be $D\_A\_2$ or $D\_B\_2$.

The merge network architecture of EHMSP for $E =$
12 is shown in Fig. 7 (a). The heads of six bundles \( \{D_{A1}, D_{A2}, D_{A3}, D_{B1}, D_{B2}, D_{B3}\} \) are passed to the selector logic. Then, selector logic determines the smallest three heads to dequeue their bundles. The three dequeued bundles may be all from \( FIFO_A \), all from \( FIFO_B \), or mixed from the two FIFOs.

At any clock cycle \( t \), inputs of the merger are rearranged as two inputs to be one of four combinations:

- \( \{D'_{A1}, D'_{A2}, D'_{B1}, R_A\} \) and \( \{R_B\} \),
- \( \{D'_{B1}, D'_{B2}, D'_{B3}, R_B\} \) and \( \{R_A\} \),
- \( \{D'_{A2}, D'_{A1}, R_A\} \) and \( \{D'_{B1}, R_B\} \),
- \( \{D'_{B2}, D'_{B1}, R_B\} \) and \( \{D'_{A1}, R_A\} \).

The advantage of proposed EHMSP over proposed EHMS is that by increasing the number of input records to the merger from \( 2E \)-records (in EHMS) to \( 2.5E \)-records (in EHMSP), the number of output records per clock cycle increased from \( E \)-records (in EHMS) to \( 1.5E \)-records (in EHMSP), as shown in Fig. 6 and Fig. 8.

Figure 7 (b) shows a behavior example of proposed EHMSP using the same input records used in Fig. 3 (b).

At clock cycle 1, \( R_A \) and \( R_B \) are initialized to zeros. Six heads \( \{34, 15, 2\} \) and \( \{37, 23, 4\} \) are passed to the selector logic. As \( \{15, 4, 2\} \) are the smallest three heads, \( \{D_{A1}^1, D_{B1}^1, D_{A1}^2\} \) are selected to be dequeued as \( \{\text{INP}_3, \text{INP}_2, \text{INP}_1\} \), respectively. According to that, the merger rearranges its inputs as \( \{D_{A2}^1, D_{A1}^1, R_A\} = \{(29\ 25\ 22\ 15), (12\ 9\ 5\ 2), \{0\ 0\ 0\ 0\}\} \) and \( \{D_{B1}^1, R_B\} = \{(19\ 13\ 7\ 4), \{0\ 0\ 0\ 0\}\} \). As shown in Fig. 7 (a), the output of EHMS is the middle three sorted \( E/3 \)-records. Therefore, the output will be \( \{(15\ 13\ 12\ 9), (7\ 5\ 4\ 2), \{0\ 0\ 0\ 0\}\} \).

At clock cycle 2, \( R_A \) and \( R_B \) are updated to the value of \( D_{A1}^2 \) and \( D_{B1}^1 \), respectively. \( \{D_{B2}^2, D_{A1}^2, D_{B1}^2\} \) are selected as inputs \( \{\text{INP}_3, \text{INP}_2, \text{INP}_1\} \), respectively. The merger rearranges its inputs to be \( \{D_{A1}^1, R_A\} = \{(40\ 38\ 35\ 34), (29\ 25\ 22\ 15)\}\) and \( \{D_{B2}^2, D_{B1}^2, R_B\} = \{(45\ 41\ 39\ 37), (33\ 32\ 27\ 23), (19\ 13\ 7\ 4)\}\). The output at this cycle is \( \{(38\ 37\ 35\ 34), (33\ 32\ 29\ 27), (25\ 23\ 22\ 19)\} \). As a result, tie record issue does not exist in EHMSP as well.

3. **Internal Structure of Important Components**

In this section, we show the internal structure of important components used to build our proposed merge sorters.

3.1 **Selector Logic Circuit**

In this subsection, we focus on the selector logic component and show how it looks in VMS, WMS, EHMS, and EHMSP. Figure 9 shows the selector logic used in VMS, proposed WMS, proposed EHMS, proposed EHMSP. For the proposed WMS, we use the same selector logic circuit used in VMS. The key part is extracted from each record and sent separately to the selector logic to be stored in \( FIFO_{KA} \) or \( FIFO_{KB} \). Then, as shown in Fig. 9 (a), the selection is done three clock cycles early using one of the three comparison results (\( \text{IN1}, \text{IN2}, \text{IN3} \)) based on the selection history (\( H1 \) and \( H2 \)). The behavior of this selector logic circuit is discussed in detail in [10].

We will explain how \( sel_A \) and \( sel_B \) are used to control MUX1, MUX2, \( R_A \), and \( R_B \) in Fig. 3. There are two possible cases depending on the values of \( sel_A \) and \( sel_B \):

- Case 1: If \( sel_A \) is true, that is, \( FIFO_A \) was selected, then MUX1 and MUX2 will output \( \{D_A, R_A\} \) and \( \{R_B\} \), respectively. Also, register \( R_A \) will be updated next clock cycle by the value of bundle \( D_A \).
- Case 2: If \( sel_B \) is true, that is, \( FIFO_B \) was selected, then MUX1 and MUX2 will output \( \{D_B, R_B\} \) and \( \{R_A\} \),
respectively. Also, register \( R_B \) will be updated next clock cycle by the value of bundle \( D_B \).

For the proposed EHMS, the selector logic circuit should select two bundles each cycle. Therefore, we propose to replace the selector logic of EHMS with \( E = 2 \) merge sorter, as shown in Fig. 9 (b). As in VMS and WMS, the key part is decoupled and stored in the corresponding \( FIFO_{XX} \). Moreover and unlike WMS, we do the selection two cycles early using two comparisons (\( C_1, C_2 \)) and one selection history register \( H \). At each cycle, we dequeue one bundle with two keys from \( FIFO_{KA} \) or \( FIFO_{KB} \) based on this two cycles selector logic. Only the head or first key of each bundle is used in the comparison. Then, this selected bundle is sent to a merge network that is exactly as the proposed WMS in Fig. 3 (a). We use one fixed (4, 2) merger, as each input bundle to the merge network has a fixed width of two keys. Outputs of the selector logic are two bits used to determine whether \( FIFO_{KA}, FIFO_{KB} \), or both is selected to fetch the upcoming bundles.

To show how \( sel_A \) and \( sel_B \) in Fig. 9 (b) are used to control MUX1, MUX2, MUX3, \( R_A \), and \( R_B \) in Fig. 5. There are three possibilities depending on the values of \( sel_A \) and \( sel_B \).

- **Case 1**: If only \( sel_A \) is true, that is, \( FIFO_A \) was selected to fetch two bundles, then \( \{D_{A2}\} \) and \( \{D_{A1}\} \) are selected as the outputs of MUX3. Moreover, register \( R_A \) will be updated next clock cycle by the output of MUX1. In this case, MUX1 output will become bundle \( D_{A2} \).
- **Case 2**: If only \( sel_B \) is true, that is, \( FIFO_B \) was selected to fetch two bundles, then \( \{D_{B2}\} \) and \( \{D_{B1}\} \) are selected as the outputs of MUX3. Moreover, register \( R_B \) will be updated next clock cycle by the output of MUX2. In this case, MUX2 output will become bundle \( D_{B2} \).
- **Case 3**: If both \( sel_A \) and \( sel_B \) are true, that is, both \( FIFO_A \) and \( FIFO_B \) were selected, then \( \{D_{B1}\} \) and \( \{D_{A1}\} \) are selected as the outputs of MUX3. Moreover, registers \( R_A \) and \( R_B \) will be updated next clock cycle by the output of MUX1 and MUX2, respectively. In this case, MUX1 output will become bundle \( D_{A1} \) and MUX2 output will become bundle \( D_{B1} \).

As shown in Fig. 9 (c), we follow the same technique as in EHMS to build the selector logic circuit of EHMSP. Each bundle inside \( FIFO_{KA} \) or \( FIFO_{KB} \) contains three keys. Therefore, we use one fixed (6, 3) merger inside the merge network shown in Fig. 9 (c) to merge the three input keys: \( R_A \), \( R_B \), and the bundle selected by multiplexer. Moreover, the selector logic of EHMS has 3-bit output to distinguish between the four cases mentioned in Sect. 2.3 and the case when neither \( FIFO_{KA} \) nor \( FIFO_{KB} \) is selected such as during stall period.

For EHMSP, there are four possible cases that explain how \( sel_{AB1}, sel_{AB2}, \) and \( sel_{AB3} \) in Fig. 9 (c) are used to control MUX1, MUX2, MUX3, \( R_A \), and \( R_B \) in Fig. 7.

- **Case 1**: If only \( sel_{AB1} \) is true, that is, \( FIFO_A \) was selected to fetch three bundles, then \( \{D_{A3}\}, \{D_{A2}\}, \) and \( \{D_{A1}\} \) are selected as the outputs of MUX3. Moreover, register \( R_A \) will be updated next clock cycle by the output of MUX1. In this case, MUX1 output will become bundle \( D_{A3} \).
- **Case 2**: If only \( sel_{AB2} \) is true, that is, \( FIFO_B \) was selected to fetch three bundles, then \( \{D_{B3}\}, \{D_{B2}\}, \) and \( \{D_{B1}\} \) are selected as the outputs of MUX3. Moreover, register \( R_B \) will be updated next clock cycle by the output of MUX2. In this case, MUX2 output will become bundle \( D_{B1} \).
- **Case 3**: If both \( sel_{AB3} \) and \( sel_{AB1} \) are true, that is, two bundles will be fetched from \( FIFO_A \) and one bundle will be fetched from \( FIFO_B \), then \( \{D_{B1}\}, \{D_{A2}\}, \) and \( \{D_{A1}\} \) are selected as the outputs of MUX3. Moreover, registers \( R_A \) and \( R_B \) will be updated next clock cycle by the output of MUX1 and MUX2, respectively. In this case, MUX1 output will become bundle \( D_{A2} \) and MUX2 output will become bundle \( D_{B2} \).

### 3.2 Internal Structure of Mergers

The structure of the merger used in proposed merge sorters is presented in this subsection. Figure 10 shows the internal structure of the merger inside VMS and our proposed merge sorters. As mentioned in Sect. 2.1, VMS requires two cascaded \((E, E)\) mergers. Mergers of VMS are built based on the odd-even merging algorithm and it is depicted in Fig. 10 (a) for \( E = 4 \). Basically, each \((E, E)\) odd-even merger is recursively composed of two \((E/2, E/2)\) mergers and \( E - 1 \) compare-and-swap2 (CAS2) modules until we reach the basic building block which is \((1, 1)\) merger that contains one CAS2 module. Each CAS2 module receives two inputs and outputs them in sorted form. As shown in Fig. 10 (a), CAS2 is pipelined into two stages and it includes one comparison.

We also use odd-even based merger for our proposed merge sorters. As discussed in Sect. 2.1, proposed WMS requires one \((2E, E)\) odd-even merger which is shown in Fig. 10 (b) for \( E = 4 \). Our \((2E, E)\) odd-even merger is built recursively using two \((E, E/2)\) odd-even mergers and \( E + 1 \) CAS2 modules. The basic building block of \((2E, E)\) odd-even merger is compare-and-swap3 (CAS3) module that is shown in Fig. 10 (b). Each CAS3 module receives 3 inputs and outputs them in sorted form. As shown in Fig. 10 (b), CAS3 is pipelined into two stages and it requires only two comparisons.

For the proposed EHMS merger, there are 3 combinations for its inputs, as discussed in Sect. 2.2. Two cases require \((1.5E, E/2)\) merger and one case requires \((E, E)\) merger. In order to minimize the required hardware
resources and prevent having an idle merger, we propose to use one generic merger for the three cases. For simplicity, we adopt the same $(2E,E)$ odd-even merger of WMS.

4. Solving Tie Record Issue

As mentioned in Sect. 2, the tie record issue arises in some merge sorters when more than one record have the same key field. In that case, some records may appear in the output more than one time, while others may not appear at all, as explained in detail in [10] and [12].

Figure 11 (a) shows input data that contains some tie records. In this data, each record is composed of one-digit number followed by an alphabet letter. The number represents the key while the alphabet letter represents the data.

Authors of VMS [12] show that using odd-even merger and a timelined bundle ordering could solve the tie record issue. The timelined bundle ordering is based on using different merge network structure. It uses two registers $R_1$ and $R_2$ as shown in Fig. 2 (b) instead of registers $R_A$ and $R_B$ shown in Fig. 2 (a). Using this timelined bundle ordering scheme makes the bundle in $R_1$ always older than the bundle in $R_2$, that is, head of $R_1$ is smaller than head of $R_2$. While the multiplexed bundle is always the most recent bundle having a head key larger than that of $R_1$ and $R_2$.

To solve the tie record issue in our proposed merge sorter, we follow VMS and use the odd-even based merger. In our proposed mergers and unlike VMS, only one merger is used. This merger has three inputs: $R_A$, $R_B$, and the multiplexer output. To distinguish between them, we use a different scheme from the timelined bundle ordering used in VMS. We add one-digit number to each record inside $R_A$, $R_B$, and the multiplexer output before sending them to the merger. This additional digit represents the timing order of each bundle and it is added as the least significant digit of key part. Because we have to distinguish three items, the timing order number may be 0, 1, or 2.

Finally, these timing numbers are omitted before outputting the sorted records.

In WMS, one selected bundle is sent to the merge network each clock cycle. Records within this bundle always have a timing order equals 2 because it is the most recent fetched bundle. Records within $R_A$ and $R_B$ registers have a timing order equals 0 or 1 depending on the decision of selector logic in previous clock cycle. For example, if the selected bundle at clock cycle $t$ is from FIFO $A$, then, at clock cycle $t + 1$, timing order of records within $R_A$ will be 1 and timing order of records within $R_B$ will be 0. This means that the bundle stored in $R_B$ is older than the bundle stored in $R_A$.

Figure 11 (b) shows a behavior example of WMS using input data with some tie records shown in Fig. 11 (a). The timing order number is highlighted in green color.

At clock cycle 1, bundle $D_{A1}$ is selected and timing order 2 is inserted. Initial timing order for $R_A$ and $R_B$ is zero. After inserting the timing order, merger inputs are $\{32B\ 32A\ 12B\ 12A\}$ and $\{000 \ 000 \ 000 \ 000\}$. After removing the timing order numbers from the merger output, it becomes $00 \ 00 \ 00 \ 00$.

At clock cycle 2, $R_A$ is updated by $D_{A1}$ and its timing order becomes 1 while that of $R_B$ becomes 0. Bundle $D_{B1}$ is selected at this cycle and timing order 2 is inserted. Merger inputs becomes $\{32F\ 32E\ 22B\ 22A\}$, $\{000 \ 000 \ 000 \ 000\}$ and $\{31B \ 31A \ 11B \ 11A\}$. Output will be $\{2B \ 2A \ 1B \ 1A\}$.

At clock cycle 3, $R_B$ is updated with $D_{B1}^2$ and its

†For simplicity, this additional timing number is omitted from the behavior examples shown before.
Fig. 11 Examples show the tie record issue solution in our proposed merge sorters. Output records are in red color. Highlighted numbers are the timing order of each record.

In EHMS, the most recent two bundles selected each clock cycle will receive a timing order 2. Timing order of registers $R_A$ and $R_B$ at cycle $t$ has three possible cases based on the selected two bundles at cycle $t-1$:

- When $\{D_{A1}, D_{A2}\}$ are selected: timing order of records within $R_A$ is 1 and timing order of records...
within \( R_b \) is 0.
- When \( \{D_{B1}, D_{B2}\} \) are selected: timing order of records within \( R_A \) is 0 and timing order of records within \( R_B \) is 1.
- When \( \{D_{A1}, D_{A2}\} \) are selected: timing order of records within \( R_A \) is 1 and timing order of records within \( R_B \) is 1.

A behavior example of EHMS is shown in Fig. 11 (c) using the same data in Fig. 11 (a).

At clock cycle 1, \( \{D_{A1} \text{ and } D_{B1}\} \) are selected and they receive a timing order 2. Inputs to the merger become ((32B 32A 12B 12A), (000 000 000 000)) and ((32F 32E 22B 22A), (000 000 000 000)). After removing timing orders, the output will be ((2B 2A 1B 1A), (00 00 00 00)).

At clock cycle 2, \( R_A \) and \( R_B \) will be updated by \( D^1_{A1} \) and \( D^1_{B1} \), respectively, and both of them will have a timing order 1. The input bundles are selected as \( D_{A2} \) and \( D_{A1} \). Output becomes ((4B 4A 3D 3C), (3B 3A 3F 3E)).

At clock cycle 3, only \( R_A \) is updated by \( D^2_{A2} \) and a timing order 1 is inserted while timing order of \( R_B \) is changed to 0. Output of this cycle is ((6B 6A 5C 5B), (5E 5D 5A 4C)). Therefore, EHMS too does not have tie record issue.

In EHMS, three bundles are selected each clock cycle and timing order 2 is inserted into them. For timing of \( R_A \) and \( R_B \) at cycle \( t \), there are four cases based on the selected bundles at cycle \( t-1 \):
  - When \( \{D_{A1}, D_{A2}, D_{A3}\} \) are selected: timing order of records within \( R_A \) is 1 and timing order of records within \( R_B \) is 0.
  - When \( \{D_{B1}, D_{B2}, D_{B3}\} \) are selected: timing order of records within \( R_A \) is 0 and timing order of records within \( R_B \) is 1.
  - When \( \{D_{A1}, D_{A2}, D_{B1}\} \) are selected: timing order of records within both \( R_A \) and \( R_B \) is 1.
  - When \( \{D_{B1}, D_{B2}, D_{A1}\} \) are selected: timing order of records within both \( R_A \) and \( R_B \) is 1.

A behavior example of EHMS is shown in Fig. 11 (d) using same data in Fig. 11 (a).

At clock cycle 1, \( D_{A1}, D_{A2}, \) and \( D_{B1} \) are selected. Merger inputs become ((42B 42A 32D 32C), (32B 32A 12B 12A), (000 000 000 000)) and ((32F 32E 22B 22A), (000 000 000 000)). Output is ((3F 3E 3B 3A), (2B 2A 1B 1A), (00 00 00 00)).

At clock cycle 2, \( R_A \) and \( R_B \) will be updated by \( D^1_{A2} \) and \( D^1_{B1} \), respectively. Therefore, timing order of records within both \( R_A \) and \( R_B \) becomes 1. The selected bundles are \( D_{B1}, D_{B2}, \) and \( D_{A1} \). Thus, output of this cycle is ((6B 6A 5C 5B), (5E 5D 5A 4C), (4B 4A 3D 3C)). From the output, EHMS is also free of the tie record issue.

5. Performance Evaluation

We choose the most recent hardware merge sorter VMS proposed in [12] as a competitor. In this section, we show evaluation results of the three proposed merge sorters WMS, EHMS, and EHMPSP. Moreover, we compare our architectures against VMS in terms of the required hardware resources and the achieved performance.

The results shown in this section, including results for VMS, are based on using node structure merge sorter for evaluation where each node contains two input FIFOs, selector logic, and a merge network.

5.1 Evaluation Setup

We use Verilog HDL to implement all architectures. Also, we use Vivado version 2019.2 targeting the Virtex UltraScale FPGA xcvu440-flg2892-3-e.

We use four factors for the evaluation: FPGA hardware resources, active ratio \( A \), maximum operating frequency \( F \), and throughput \( T \). We use the number of occupied Slices, Look-Up Tables (LUTs), and Flip-Flops (FFs) as the FPGA hardware resources. In terms of the targeted Virtex UltraScale FPGA device, each slice is composed of eight 6-input LUTs and sixteen FFs. We define the active ratio as the number of clock cycles when the merge sorter outputs valid records divided by the total number of elapsed clock cycles. Finally, throughput is calculated as \( T = E \times R \times F \times A \), where \( R \) is record width†.

Simulation has been done to ensure that all the architectures merge and sort the input records correctly. We used a large randomly generated sequences of length 1 million records per sequence. Finally, we verified that all architectures perform the merge sort operation correctly by comparing the results with that of a software based merge sorter.

5.2 Evaluation of Proposed WMS, EHMS, and EHMPSP vs. VMS

Table 1, Table 2 and Table 3 summarize the evaluation results for the state-of-the-art merge sorter and our proposed merge sorters based on the four aspects mentioned above.

5.2.1 FPGA Hardware Resources

Figure 12 (a), Fig. 12 (b), and Fig. 12 (c) show percentage of the number of FFs, LUTs, and slices, respectively, required to implement VMS, WMS, EHMS, and EHMPSP for \( E = 8 \) to 512 records. It is clear from the results that VMS requires the largest number of FFs, LUTs, and slices for all cases. Moreover, there is a significant decrease in the number of FFs, LUTs, and slices required for our three proposed merge sorters WMS, EHMS, and EHMPSP.

For example, in the case of outputting 512 sorted records per clock cycle, the required number of FFs for VMS, proposed WMS, and proposed EHMS are 1,293,443 (25.53%), 918,128 (18.12%), and 754,054 (14.88%), respectively. Therefore, our proposed WMS and EHMS require about 71% and 58%, respectively, of the required

†VMS uses the record width of 64-bit where key part is 32-bit and data part is 32-bit. For a fair comparison, we use same width in WMS, EHMS, and EHMPSP architectures.
As discussed in Sect. 2.2, this increased number of LUTs in the case of EHMS and EHMSP is the result of increasing the complexity of the selector logic to select two or three bundles instead of one as in VMS and WMS. However, for a larger number of output records per clock cycles, all proposed merge sorters are still using much fewer LUTs than VMS.

Regarding the number of slices, percentage of the number of required slices for VMS, WMS, EHMS, and EHMSP are shown in Fig. 12 (c). The slice utilization reflects the total utilization of both FFS and LUTs where each slice contains eight 6-input LUTs and sixteen FFs as mentioned in Sect. 5.1. From Fig. 12 (c), it is clear that the proposed EHMS is the most slices-saver merge sorter. In case of \( E = 512 \), VMS, WMS, and EHMS require 176,604 (55.78%), 135,522 (42.80%), and 124,286 (39.25%) slices, respectively. This means that WMS and EHMS require 77% and 70%, respectively, of the required number of slices to implement VMS. Also, it is noticeable that when \( E \) is smaller than 256, VMS uses a slightly higher number of slices than EHMSP. However, for larger values of \( E \), VMS and EHMSP use almost same number of slices.
5.2.2 Active Ratio

In terms of active ratio, all of VMS, WMS, EHMS, and EHMSP achieve a very high active ratio that is about 100% for a large number of output records per clock cycles. The active ratio of each merge sorter is shown in Fig. 13. The high active ratio means that the occupied hardware resources are efficiently and fully used.

5.2.3 Maximum Operating Frequency

Figure 14 presents the maximum operating frequency of each merge sorter that can run at. For large number of output records per clock cycle, the proposed merge sorters are much faster than VMS. For instance, When $E = 256$, the operating frequencies of VMS, WMS, and EHMS are 222, 357, and 313 MHz, respectively. WMS is always the fastest merge sorter achieving about 200 MHz when $E = 512$ records. In contrast, VMS is the slowest merge sorter. For EHMSP, it runs at lower operating frequency than VMS in
case of small number of output records but in case of large number of output records, it beats VMS gradually to run at higher operating frequency. Note that VMS merge sorter could not meet the timing constraint in case of $E = 512$ because of the high level of congestion. Therefore, it is clear from the figure that the performance gap between VMS and our proposed merge sorters increases when $E$ is increased. As a result, we can say that our proposed merge sorters remain faster than VMS.

5.2.4 Throughput

The performance of a merge sorter is measured by its throughput. In average, proposed WMS achieves the best throughput than other merge sorters, including the state-of-the-art VMS, in all cases, as shown in Fig. 15. For instance, the throughput of VMS, WMS, and EHMS when $E = 256$ are about 3641, 5851, and 5110 Gb/s, respectively. This means that WMS and EHMS achieve about 1.6x and 1.4x higher throughput, respectively, than VMS while using much fewer hardware resources.

6. Conclusion

This paper proposes three efficient hardware architectures for merge sorter (WMS, EHMS, and EHMSP). We adopt the odd-even merging approach to build our proposed hardware merge sorters. We implement our proposals on Virtex UltraScale FPGA and evaluate them against the state-of-the-art hardware merge sorter named VMS as a competitor.

The results show a significant decrease in the required number of LUTs and FFs for our proposed merge sorters. For example, when outputting 512 records per clock cycle, the required number of FFs and LUTs for proposed EHMS is 58% and 80%, respectively, of those required for VMS. Proposed WMS also uses fewer resources as 71% of FFs and 78% of LUTs required by VMS.

Moreover, all proposed merge sorters not only use fewer hardware resources but also their performances are significantly higher than the state-of-the-art merge sorter. For instance, the performance of WMS and EHMS are 1.6x and 1.4x higher than that of VMS when 256–records are output per clock cycle ($E = 256$).

References

[1] H. Inoue and K. Taura, “Simd- and Cache-Friendly Algorithm for Sorting an Array of Structures,” Proc. VLDB Endow., vol.8, no.11, pp.1274–1285, 2015.
[2] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri, “PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort,” Proc. VLDB Endow., vol.8, no.12, pp.1518–1529, 2015.
[3] D. Merrill and A. Grimshaw, “High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing,” Parallel Processing Letters (PPL), vol.21, no.02, pp.245–272, 2011.
[4] A. Davidson, D. Tarjan, M. Garland, and J.D. Owens, “Efficient Parallel Merge Sort for Fixed and Variable Length Keys,” Proc. Innovative Parallel Computing (InPar), pp.1–9, 2012.
[5] D. Koch and J. Torresen, “FFGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting,” Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp.45–54, 2011.
[6] J. Casper and K. Olukotun, “Hardware Acceleration of Database Operations,” Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp.151–160, 2014.
[7] W. Song, D. Koch, M. Lujan, and J. Garside, “Parallel Hardware Merge Sorter,” Proc. 24th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.95–102, 2016.
[8] S. Mashimo, T.V. Chu, and K. Kise, “High-Performance Hardware Merge Sorter,” Proc. 25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.1–8, 2017.
[9] E.A. Elsayed and K. Kise, “Design and Evaluation of a Configurable Hardware Merge Sorter for Various Output Records,” Proc. 12th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc), pp.201–208, 2018.
[10] M. Saitoh, E.A. Elsayed, T.V. Chu, S. Mashimo, and K. Kise, “A High-Performance and Cost-Effective Hardware Merge Sorter without Feedback Datapath,” Proc. 26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.197–204, 2018.
[11] P. Papaphilippou, C. Brooks, and W. Luk, “FLiMS: Fast Lightweight Merge Sorter,” Proc. International Conference on Field-Programmable Technology (FPT), 2018.
[12] M. Saitoh and K. Kise, “Very Massive Hardware Merge Sorter,” Proc. International Conference on Field-Programmable Technology (FPT), 2018.
[13] K.E. Batcher, “Sorting Networks and Their Applications,” Proc. Spring Joint Computer Conference, AFIPS, pp.307–314, ACM, 1968.
[14] E.A. Elsayed and K. Kise, “Towards an Efficient Hardware Architecture for Odd-Even Based Merge Sorter,” Proc. 13th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc), pp.249–256, 2019.
Elsayed A. Elsayed received the B.E. degree and M.E. degree from Aswan University, Egypt in 2009 and 2014, respectively. He is currently an assistant lecturer at Electrical Engineering Department, Aswan University, Egypt and PhD student at Department of Computer Science, Tokyo Institute of Technology, Japan. He is interested in computer architecture, matrix/vector processing, and FPGA-based architectures.

Kenji Kise received the B.E. degree from Nagoya University in 1995, the M.E. degree and the Ph.D. degree in information engineering from the University of Tokyo in 1997 and 2000, respectively. He is currently an associate professor of the School of Computing, Tokyo Institute of Technology. His research interests include computer architecture and parallel processing. He is a member of ACM, IEEE, IEICE, and IPSJ.