Analog Integr Circ Sig Process (2017) 92:327–339
DOI 10.1007/s10470-017-0991-6

Sorting-free digital median filter for SOCs

Saleh Abdel-hafeez[1] · Behrooz Parhami[2] · Arwa Damir[1]

Received: 22 March 2017 / Accepted: 24 May 2017 / Published online: 8 June 2017

Abstract In this work, we propose a new median-finding algorithm which computes the median value in an input list of integers on-the-fly, without any data-sorting operations. We present a complete digital CMOS implementation, associated timing diagrams, and a formal mathematical proof, which show the overall average number of clock cycles for median-finding to be linearly proportional to the input length, that is, $O(N)$ average-time complexity, when N is less than about 100. Hence, our proposed sorting-free median algorithm is suitable for practical applications on 3×3 and 5×5 image scan matrices, which are in common use for hand-held devices and entertainment graphics applications. Our proposed hardware precludes the need for SRAM memory or complex circuitry, such as pipelining structures, but rather uses simple registers to hold the input values, performing comparison-swapping on 3 values, along with counting, to derive the median value. There is no restriction on the input sequence with regard to having repeated elements. We evaluate an ASIC design of our sorting-free median algorithm using 90 nm TSMC technology, with 1 V supply voltage and a clock frequency of 2 GHz, on example cases of 3×3 (9 values) and 5×5 (25 values) image-scan matrices. The resulting designs have a minimum transistor-count ranging from 3202 to 5203. Results show that our sorting-free median algorithm, when used on 512×512 images with 8-bit pixels, takes 0.364 and 1.394 ms to scan the complete image using 3×3 and 5×5 scan matrices, respectively, with the associated power consumption ranging from 3.24 to 1.66 mW.

Keywords ASICs · Digital CMOS · Median filter · Sorting · Scan matrices · SOCs · 2 GHz

✉ Saleh Abdel-hafeez
sabdel@just.edu.jo

[1] Jordan University of Science and Technology, Irbid 22110, Jordan

[2] University of California, Santa Barbara, CA 93106-9560, USA

1 Introduction

Median-finding is an important functionality in hardware ICs used for digital signal processing (DSP). Examples occur in hand-held devices, gaming systems, and home entertainment applications. One of the essential applications of median-finding is in median filtering for image and speech processing, with the goal of reducing the blurring effects due to frame capturing speed overlaps [1, 2]. Consequently, digital median-finding is extensively studied, with solutions ranging from hardware circuits [3, 4] to software algorithms [5, 6] provided as kernel resources. Many median-finding schemes are based on sorting algorithms, where the median value is extracted after arranging the numbers of the given sequence in ascending/descending order. Direct median-finding schemes derive the desired median value without sorting the data.

Hardware median-filter implementations, which provide high-throughput processing are categorized into analog [7–9] and digital [10–14] circuit structures, with each approach offering certain advantages and suffering from some drawbacks. Our work presented here is focused on a digital solution that targets DSP SOCs (systems on chip) related to home entertainment and hand-held devices, where further enhancement of digital signal is necessary for video and audio quality. Commonly, median-finding schemes based on sorting lead to complex, and thus costly, hardware designs that include memory and pipelined circuits [15, 16]. Besides, a sorting-based scheme performs a lot of unnecessary operations beyond those needed for finding the median and thus leads to disadvantages in terms of circuit complexity and power consumption.

By contrast, non-sorting-based median-finding structures can lead to lower hardware cost and energy consumption by virtue of not performing any redundant operations [17, 18]. A relevant factor that makes directly-realized hardware structures cost-effective is that in targeted applications, median-finding is often limited to small digital-image matrices, often ranging in size from 3×3 to 5×5; that is, an input data sequence in the range of 9–25 numbers. Sorting algorithms, on the other hand, are often optimized for very long input sequences, typically in the range of hundreds to many millions of numbers. Therefore, there is a need for fast, cost-effective circuit designs for median-finding in DSP applications featuring graphics, video, and audio in the context of SOCs. In this paper, we propose a new median-finding algorithm targeted for custom SOCs applications that require evaluating the median on small- to moderate-sized input sequences, such as those typically found in graphics accelerators and hand-held video/audio processing DSP chips.

The rest of this paper is organized as follows. Section 2 introduces the principles of our median-finding algorithm, along with illustrative examples. Section 3 provides pertinent mathematical analyses with extreme-case evaluation. Section 4 is devoted to the hardware data path and control logic implementations, along with the requisite timing diagrams. In Sect. 5, we present simulation results and relate our results to recent hardware implementations. Section 6 contains our conclusions and elaboration on the advantages of our proposed design.

2 Algorithm principles and examples

The proposed median-finding algorithm produces the median value without sorting the numbers in a given sequence. For simplicity, assume the input is given by a sequence of integer values $I_1, I_2, \ldots, I_N$, where the length N of the sequence is an odd number. Define the first-set to consist of the pair I_1 and I_N of inputs; that is, $SET_1 = [I_1, I_N]$ is composed of the first and last numbers in the input sequence. The second, third, … sets are similarly defined: $SET_2 = [I_2, I_{(N-1)}]$ contains the second and next-to-last numbers; $SET_3 = [I_3, I_{(N-2)}]$; and so on, until $SET_{(N-1)/2} = [I_{(N-1)/2}, I_{(N+3)/2}]$). Then, consider the middle number of the sequence, that is, $I_{(N+1)/2}$, and call it the pivot element.

Next, begin with comparing three numbers of the input sequence, which are the pivot element and the two numbers in SET_1, applying the following rules:

- Rule 1: If the pivot number falls between the two numbers of SET_1, that is, $I_1 < I_{(N+1)/2} < I_N$, or $I_1 > I_{(N+1)/2} > I_N$ then no change occurs in either the pivot or SET_1.
- Rule 2: If the pivot number does not fall between the two numbers of SET_1, that is, $I_1 < I_N < I_{(N+1)/2}$, then, the pivot is swapped with I_N. In other words, the new pivot number becomes I_N and SET_1 becomes $[I_1, I_{(N+1/2)}]$.
- Rule 3: If the pivot number does not fall between the two numbers of SET_1, that is, $I_{(N+1)/2} < I_1 < I_N$, then, the pivot is swapped with I_1. In other words, the new pivot number becomes I_1 and SET_1 becomes $[I_{(N+1/2)}, I_N]$.
- Rule 4: if the pivot number is equal to either or both numbers of SET_1, then no swap occurs.
- Rule 5: If swapping occurs and thus the pivot number changes, then the process above is repeated.

Consequently, the new pivot number is to start comparing again, beginning with SET_1. Otherwise, we keep comparing the pivot with SET_2, SET_3, and so on, moving inwards, that is towards the center of the sequence, until we get to the center or we encounter a swap. Although the rules above are defined for SET_1, the same rules are applied to all sets of the sequence of size N. To clarify the process, we offer Table 1 that tracks the step of a simple example, demonstrating the working of our algorithm. We have taken the input list to contain no repeated numbers, in order to focus on the main functionality of the algorithm. However, the algorithm is also valid for the case of repeated numbers, as illustrated in the example of Table 2.

The first line of Table 1 shows the input data sequence as order shown by the subscript $i = 1, 2, \ldots, 9$ for $N = 9$ numbers, as they arrive randomly. As we moved inward during pass 1, the pivot element 1 is compared with $SET_1 = [0, 9]$. Since 1 is between 0 and 9, no changes occur to pivot, nor to SET_1 (Rule 1). In pass 2, we compare

Table 1 An example of computing nine distinct random elements as an input data sequence

	X_1	X_2	X_3	X_4	Pivot	X_6	X_7	X_8	X_9
Data	0	7	4	5	1	8	3	6	9
Pass 1	0	7	4	5	1	8	3	6	9
Pass 2	0	7	4	5	6	8	3	1	9
Pass 3	0	7	4	5	6	8	3	1	9
Pass 4	0	7	4	5	6	8	3	1	9
Pass 5	0	7	6	5	4	8	3	1	9
Pass 6	0	7	6	5	4	8	3	1	9
Pass 7	0	7	6	5	4	8	3	1	9
Pass 8	0	7	6	5	4	8	3	1	9
Pass 9	0	7	6	4	5	8	3	1	9

Table 2 An example of computing nine (non-distinct) random elements as an input data sequence

	X_1	X_2	X_3	X_4	Pivot	X_6	X_7	X_8	X_9
Data	4	7	4	5	1	8	3	3	9
Pass 1	1	7	4	5	4	8	3	3	9
Pass 2	1	7	4	5	4	8	3	3	9
Pass 3	1	7	4	5	4	8	3	3	9
Pass 4	1	7	4	5	4	8	3	3	9
Pass 5	1	7	4	4	5	8	3	3	9
Pass 6	1	7	4	4	5	8	3	3	9
Pass 7	1	7	4	4	5	8	3	3	9
Pass 8	1	7	5	4	4	8	3	3	9
Pass 9	1	7	5	4	4	8	3	3	9
Pass 10	1	7	5	4	4	8	3	3	9
Pass 11	1	7	5	4	4	8	3	3	9
Pass 12	1	7	5	4	4	8	3	3	9

the pivot element 1 with $SET_2 = [6, 7]$, where the pivot is replaced by 6 and SET_2 becomes [1, 7], by rule 2. In pass 3, the new pivot element 6 is compared with $SET_1 = [0, 9]$ due to Rule 5 since the pivot was changed in pass 2. Now, the pivot element 6 falls between the numbers of $SET_1 = [0,9]$. By Rule 1, we have no changes in the pivot or SET_1. In pass 4, the pivot element 6 is compared with $SET_2 = [1, 7]$. Again, Rule 1 applies and we have no changes in the pivot or SET_2. In pass 5, the pivot element 6 is compared with $SET_3 = [3, 4]$. Rule 3 causes the pivot to be replaced with 4 and SET_3 to become [3, 6]. Since a change in the pivot has occurred, we need to start again from the beginning due to Rule 5. The example of Table 1 shows that we have 9 passes to compute the median element (the final pivot value 5). Table 2 illustrates the example when there are duplicate numbers. The same rules are followed here, but Rule 4 is used more frequently.

3 Mathematical analysis

From the description of the algorithm in Sect. 2, along with Tables 1 and 2, it is evident that in the best case, the algorithm performs $N - 1$ iterations (steps) when presented with an odd-length list of N input numbers. The $N - 1$ steps correspond to a single pass through the input, with no swaps or exchanges taking place. For the input list 3, 4, 7, 1, 5, 8, 0, 6, 9, which constitutes a best-case example, four pairs of comparisons are performed to establish that the middle element 5 is between 3 and 9, between 4 and 6, between 7 and 0, and between 1 and 8, before ending execution with the output result 5. Because the algorithm begins its scanning anew after each swap operation, it appears that the worst-case complexity will be quadratic in N. However, this intuition must be proven formally, taking care to confirm that at least $\Omega(N^2)$ steps are required in the worst case and that the exchanges will in fact come to an end after $O(N^2)$ steps and will not go on indefinitely.

Lemma 1 *Our median-finding procedure always stops after a finite number of steps.*

Proof Considering the middle element y and the two symmetrically located elements x and z that are involved in an exchange, we note that the exchange transforms xyz to yxz when $x > y$ and $x < z$ (the other cases are similar). In this case, the new elements ending up in the locations originally occupied by x and z will have a larger difference than $|x - z|$. Thus, every exchange will increase $\sum_{i=1}^{\frac{n-1}{2}} |x_{k+i} - x_{k-i}|$ [k is the midpoint index] by the difference of two of the input numbers. For the example shown in Table 1, the sum above assumes the values 14, 18, 19, 21, and 22 on rows 1–5, respectively. Given that the sum of the symmetric differences in the expression above has a finite maximum possible value, the procedure cannot go on indefinitely. □

We next turn to extreme-case performance assessment by considering all possible arrangements of N numbers (exhaustive evaluation). The evaluation was performed for input sizes from N = 3 to N = 17, taking advantage of symmetries to reduce the running time towards the upper end of this range. Usually the temporal scan matrix median filter ranges in size from 3 × 3 to 5 × 5, with 3 × 3 case being most likely. Table 3 records our median evaluation worst case iterations for input sequences of different lengths N. Additionally, due to the fact that sorting algorithms have also been used for median-finding in some applications, we report in Fig. 1 the simulation time for our proposed algorithm against a well-known sorting algorithm (quick-sort3) for small sequence lengths encountered in typical median-finding applications. We assess only the worst-case time for our proposed algorithm against the average-case time of quick-sort3 as an extreme-case evaluation. Simulation times in Fig. 1 show the proposed median-finding algorithm outperforming the quick-sort3 computing time for sequences of length under 101. From Fig. 1 and Table 3, the worst-case number of iterations counts for a sequence of size N can be approximated as:

$$\text{Worst number of iterations } W(N) = 0.4 * (N - 1)^2 \quad (1)$$

Hence, the upper bound complexity in number of iterations is $O(N^2)$ with a fairly small scale factor <1, while the lower bound on the number of iterations is simply O(N), that is:

Table 3 Extreme case sequences types that generate the worst-case number of iterations

Number size	Sequence numbers order (types)	Worst iterations count
3	2 1 3	2
5	1 4 3 5 2	6
7	1 6 3 5 4 7 2	13
9	9 1 4 3 7 6 5 2 8	23
11	11 1 2 6 5 9 8 7 4 3 10	36
13	13 1 2 10 6 5 9 8 7 11 4 3 12	54
15	13 1 2 12 7 6 5 11 10 9 8 14 4 3 15	78
17	1 17 16 3 8 7 6 5 13 12 11 10 9 4 14 15 2	107

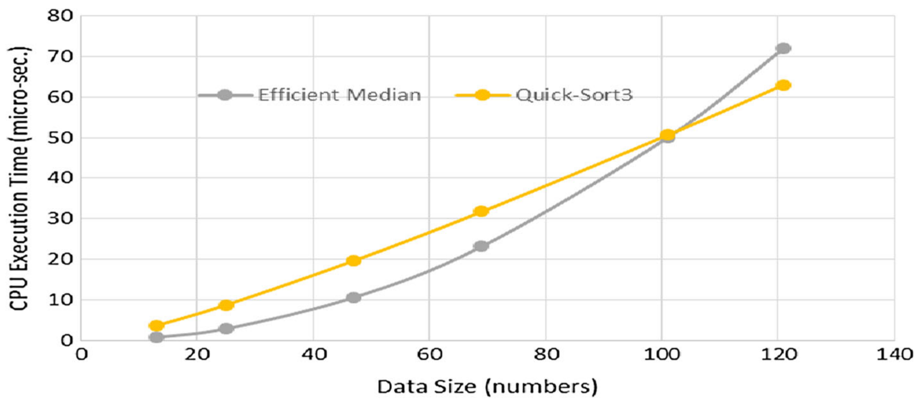

Fig. 1 Simulation time of the proposed median (*upper bound*) versus Quick-Sort3 (average)

$$\text{Best number of iterations } B(N) = N - 1 \quad (2)$$

In practice, the number of iterations will fall between the two bounds above. Figure 2 shows the fraction of worst-case occurrences as a function of the sequence size N. According to Fig. 2, worst-case input patterns constitute only 0.035 (3.5%) of the overall number of arrangements for sequences of length 9. For $N = 25$, the worst-case fraction drops to 10^{-5}. Hence, in a great majority of cases with $N > 5$, the number of iterations for our algorithm is far below the worst-case number. By contrast, the worst-case time for quick-sort3 is $O(N^2)$ when the input sequence is reverse-sorted or when it is already sorted or near-sorted [11].

4 Cmos hardware realization of our median algorithm

The overall hardware structure for our median algorithm is divided into the data path unit and the control unit. Figure 3 depicts the input–output signals of a complete block diagram for our median-finding unit, which evaluates the median for $N = 2^K$ input numbers of size K-bits. The basic architecture operates in two sequential phases: the pivot-initialization phase (Sect. 4.1) followed by the median-evaluation phase (Sect. 4.2). The control unit (Sect. 4.2) is a simple state machine that controls the data path's phases using only a small number of D-type flip-flops (DFFs) and a counter. Median computation begins when the *START-*

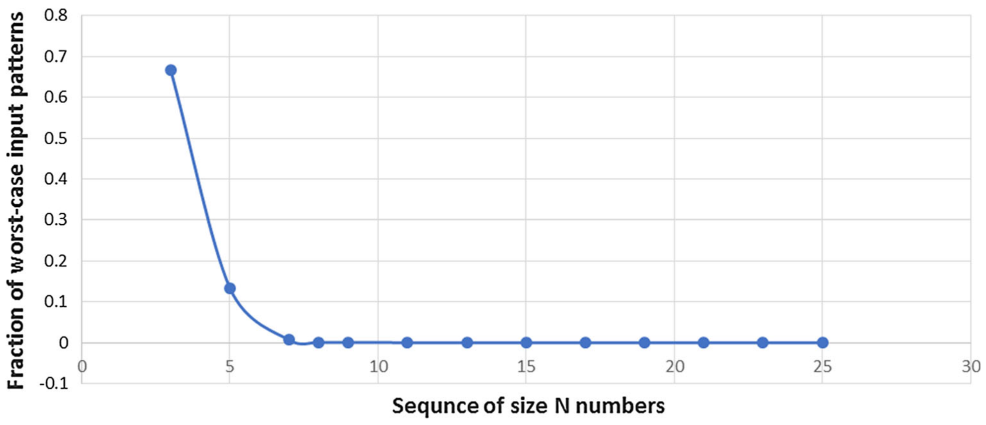

Fig. 2 Occurrences of worst case iterations counts with respect to sequence of size N

EXT signal is asserted, while the design signals that median has been obtained by asserting the *END-EXT* signal.

Figure 4 depicts the flow-chart of the proposed median-finding algorithm using standard symbols [19]. This flow-chart shows the overall simplicity of the design and iteration details to compute the median. The hardware diagram of Fig. 5 shows the details of components and signals in the data path and control units. The design comprises of simple logic components. There is a parallel-in parallel-out serial shifter of size N registers holding k bits each. In addition, tri-state buffers are used to guide the direction of data comparisons, and the equality-detector component to check whether or not the median has been replaced with a new number. A simple one-hot counter is used to control the inward movement of comparisons towards the pivot element for the given sequence, which is located in the serial shifter.

The median-comparator, which is considered the essential component of this design, is realized in Fig. 6. It is used to compare the pivot element with $SET_i = \{L, R\}$ comprised of the right (R) and the left (L) numbers. Subsequently, recalling the algorithm's rules discussed in Sect. 2, the outputs of the median-comparator are classified into one of the following decision actions:

- OM = L; OL = M; OR = R {Pivot number (M) is swapped with Left number (L)}
- OM = R; OL = L; OR = M {Pivot number (M) is swapped with Right number (R)}
- OM = M; OL = L; OR = R {No swap operation takes place}

The median-comparator is comprised of three standard basic comparators, with each comparator evaluating the decisions of *Greater-than (>)* or *Equal (=)* of the given two inputs. Tables 4 and 5 show the Boolean truth table for distinct input numbers and repeated (non-distinct) input numbers of these three comparators. In way of explanation, the "0 0 0″ case in the first row of Table 4 illustrates the condition as follows:

- Pivot number (M) is less than the left number (L) and the right number (R)
- The left number (L) is less than the right number (R)

Following this representation, we notice that the cases "0 0 1" and "1 1 0" do not exist, since they violate consistency requirements for the comparison results. Thus, there are no arrangements of numbers that can lead to either of these two combinations. Hence, the design of median-comparator is further simplified. An additional example provided in Table 5 for the case of repeated numbers "= 0 0" shows median number is less than the numbers of the right (R) and left (L), while both right (R) and left (L) numbers are equal. Following the rules in Sect. 2, the decision action of the median-comparator is to swap the pivot (M) with the numbers of the left (L) or right (R); in our design, we choose to swap the pivot (M) with the left number (L). Following these representations, Table 5 shows several cases that cannot arise due to violations similar to what was cited above. For example, the case "1 1 =" implies that median number equals the right number and is greater than the left number, while the left number is greater than right number. Furthermore, for cases "1 0 =" or "1 = 1", no swap action should occur. On the other hand, for cases "= 0 0" and "= 1 1" swap should occur between the pivot and the left number or the pivot and the right number.

Another important component is the equality-detector circuit that is realized in Fig. 7. The equality-detector circuit is used to verify whether the pivot has been changed during the current iteration with respect to previous iteration or has remained the same. The output of this circuit is directed to the reset input of the counter, which controls the releasing of SET_i from tri-state buffers. The circuit receives the input signal *D1* from the comparator-median circuit and stores it in the DFF at the falling edge of the *CLK* signal. After a small delay, which is measured by an even number of cascaded inverters, the DFF's stored value (*D1*) is re-initialized back to zero in order to release the counter from the reset

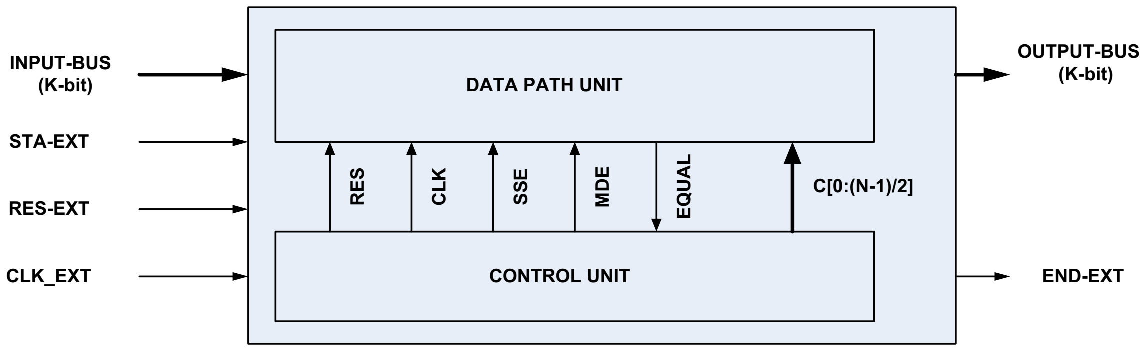

Fig. 3 Block diagram of the hardware structure for our median-finding unit

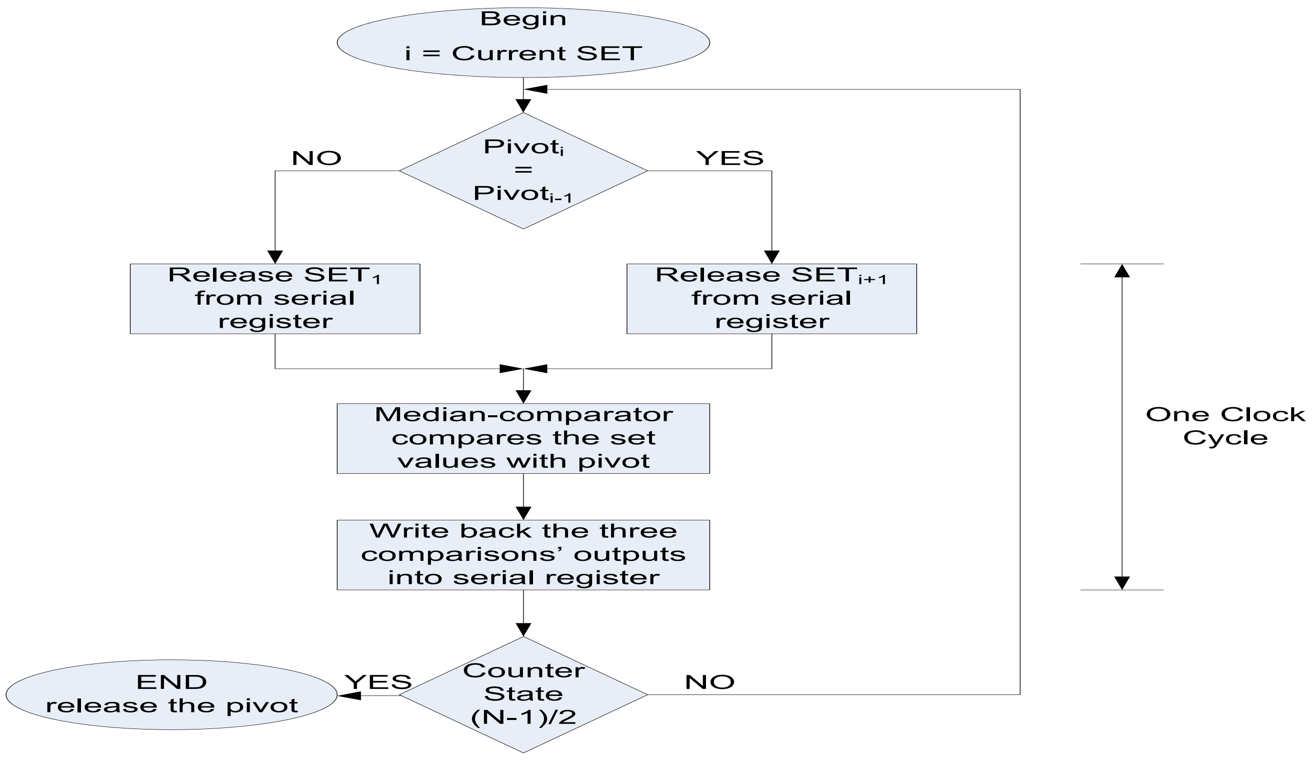
Begin
i = Current SET
Pivot_i = Pivot_{i-1}
NO
YES
Release SET_1 from serial register
Release SET_{i+1} from serial register
Median-comparator compares the set values with pivot
Write back the three comparisons' outputs into serial register
Counter State (N-1)/2
YES
NO
END release the pivot
One Clock Cycle

Fig. 4 Hardware flow for the median-evaluate phase

condition. The small delay provides a small pulse to the input of the counter's reset port in order to reset the counter completely and then releases it from reset, this reset time of the counter takes approximately 1–2 gate delays.

4.1 Data path unit

The operation of data path is divided into two phases, which are the pivot-initialize phase and the median-evaluate phase. The pivot-initialize phase is triggered by *START-EXT* signal and ends by the de-assertion of the *SSE* signal. The median-evaluate phase starts by the assertion of the *MDE* signal and ends by the de-assertion of *MDE* signal and the assertion of the *END-EXT* signal, indicating the median value of the stored sequence is available. Each iteration of the data path happens within a single clock cycle.

Let us first discuss the pivot-initialize phase in some detail. During this phase, each binary input element is directed into the serial shifter (SS) of size $N = 2^K$ registers, with *k-bit* register size. The register size (*k-bit*) is the size of the input bus, which is also the width of input numbers, usually 8 bits. The sequence of numbers is stored in the serial shifter (SS) by shifting every number to the next register at every rising clock edge. The serial shifter is a simple register array (SS_i), where the i-th register stores the i-th input number. This operation is equivalent to a sequence numbers given the first row of Tables 1 and 2, where data of the sequence is stored and the pivot element appears at the center of the array. The serial shifter (SS) is controlled by two signals (*SSE* and *RES*) with negative edge-trigger clock that is fed from the control unit, as shown in Fig. 5. In our proposed design the input sequence is stored in the serial shifter in parallel, by using the Parallel-in Parallel-out option. This option allows the sequence of data numbers to be stored in one clock cycle.

Figure 8 depicts the associated timing diagram, which shows the detailed streamlined sequential timing for the pivot-initialize phase. In this diagram, the *START-EXT* signal indicates the beginning of a new block of $N = 2^K$ input numbers, each k bits wide. The *START-EXT* signal actually arrives randomly; hence, the control circuit is designed to consecutively trigger several intermediate signals during the pivot-initialize phase. First, the reset signal (*RES)* is asserted high for one clock cycle to initialize all registers. Next, the *SSE* signal is asserted for one clock cycle to store the block of $N = 2^K$ input values into serial shifter using parallel-in parallel-out option.

Following the timing diagram in Fig. 8 for the pivot-initialize phase cycle time, we see clearly that the *RES* signal requires a D-type Flip-Flop (DFF) access time to be triggered. Similarly, the *SSE* requires a DFF access time to be triggered with respect to clock edge. Besides, the serial shifter, store the numbers in parallel within the DFF access

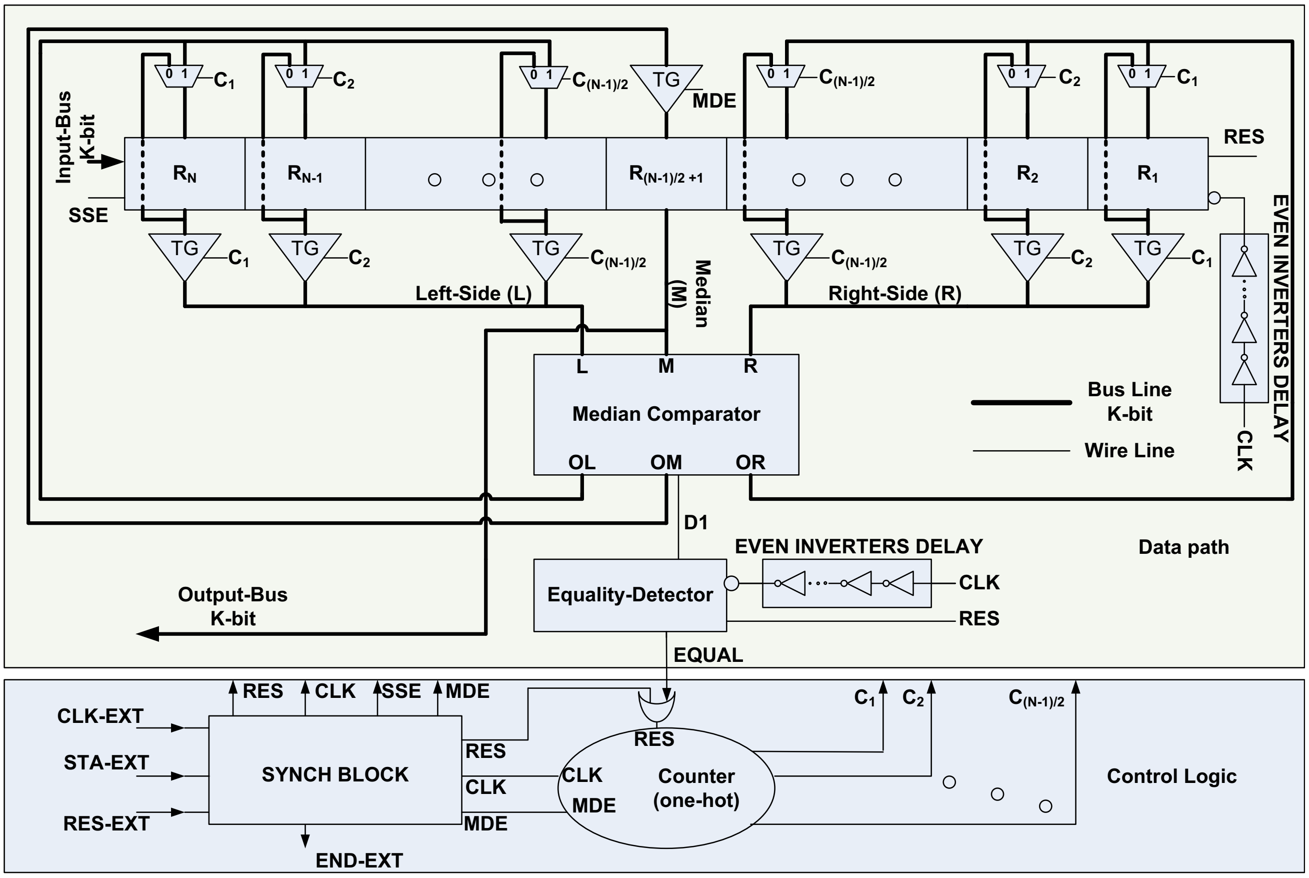

Fig. 5 Complete hardware structure with all related signals and components

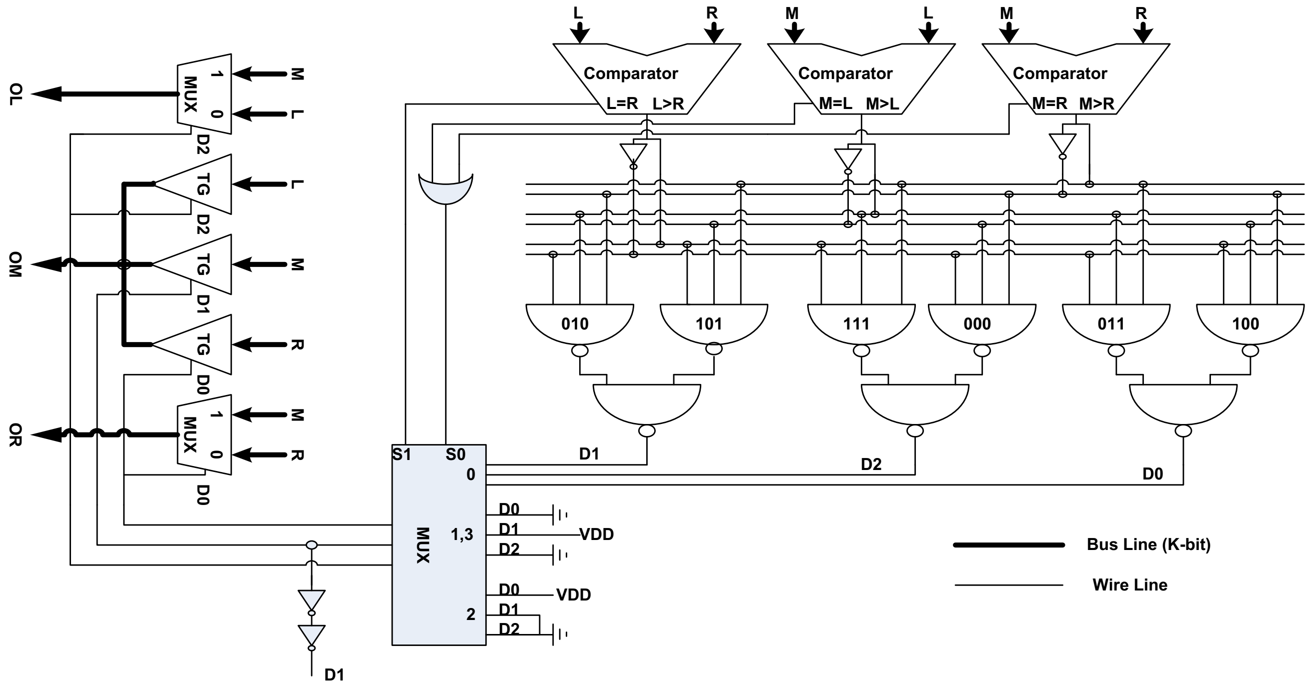

Fig. 6 The median-comparator circuit

Table 4 Truth table for the case non-repeated numbers for the three comparators in comparator-median circuit

L > R	M > L	M > R	D0	D1	D2	Comments
0	0	0	0	0	1	Swap middle with left
0	0	1	X	X	X	Impossible case
0	1	0	0	1	0	No swapping
0	1	1	1	0	0	Swap middle with right
1	0	0	1	0	0	Swap middle with right
1	0	1	0	1	0	No swapping
1	1	0	X	X	X	Impossible case
1	1	1	0	0	1	Swap middle with left

Table 5 Truth table for the case repeated numbers for the three comparators in comparator-median circuit

L ≥ R	M ≥ L	M ≥ R	D0	D1	D2	Comments
=	0	0	1	0	0	Swap middle with right
=	1	1	1	0	0	Swap middle with right
1	=	1	0	1	0	No swap
0	=	0	0	1	0	No swap
1	0	=	0	1	0	No swap
0	1	=	0	1	0	No swap
=	0	1	X	X	X	Impossible case
=	1	0	X	X	X	Impossible case
0	=	1	X	X	X	Impossible case
1	=	0	X	X	X	Impossible case
1	1	=	X	X	X	Impossible case
0	0	=	X	X	X	Impossible case

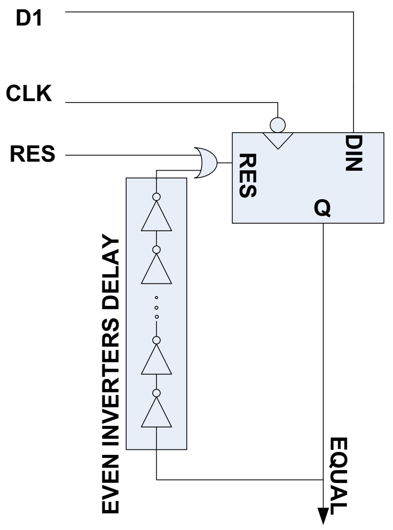

Fig. 7 The equality-detector circuit

time. It is important to note that all related signals (*RES*, *SSE*) and storing in serial shifter take place within the DFF access time with respect to clock edge, as clearly shown in the timing diagram of Fig. 8. Therefore, the pivot-initialize phase's cycle time ($T_{pivot\text{-}initialize}$) is the DFF access time (T_{DA}). We have:

$$T_{pivot-initialize} = T_{DA} \quad (3)$$

Consequently, the pivot-initialize phase according to suggested timing diagram requires two cycles (the reset cycle and the parallel-in/parallel-out store data cycle) to store *N* input values.

We next discuss the median-evaluate phase in detail. The median-evaluate phase begins after the *SSE* signal is de-asserted and the *MDE* signal is asserted, as seen in Fig. 8. Hence, the median-evaluate phase timing diagram shown in Fig. 9 starts by enabling the parallel counter at clock rising edge (*CLK-EXT*). Subsequently, the parallel counter enters the first state that enables the tri-state switches of registers SS_1 and SS_N that belong to serial shifter (SS). Hence, it releases $SET_1 = [I_1, I_N]$ for comparison with the pivot number. Then, the median-comparator compares these values and releases the middle value (*OM*), right value (*OR*), and left value (*OL*), based on the rules given in Sect. 2.

Finally, the three values of median-comparator outputs (*OM*, *OR*, *OL*) are stored back at the negative edge of clock (*CLK*) into serial shifter (SS) at the register locations for SET_1 and the pivot register. Concurrently, the equality-detector receives the *D1* value from median-comparator and stores this value into DFF at the falling edge of clock (*CLK*). The *D1* value of median-comparator determines whether or not the pivot number has been swapped with the set values. Thus, this *D1* value is used to reset the counter or allow the counter to move to next state at the following rising edge of clock cycle (*CLK-EXT*). Once the counter state reaches $(N-1)/2$, the median is available and the *MDE* signal is de-asserted. Hence, upon the de-assertion of the *MDE* signal, the *END-EXT* signal is asserted indicating that the median has been obtained.

Following the timing diagram in Fig. 9, we use two clock signals, which are *CLK-EXT* and *CLK*. The *CLK* signal is actually a delayed version of *CLK-EXT* with inverted assertion level. Taking the measure time from the rising edge of *CLK-EXT* to the fall of *CLK* gives more time than only working at the two edges of *CLK-EXT*. Thus, starting from clock rising edge *CLK-EXT*, the counter (*Tpc*) provides the state that enables the tri-state buffer (*Trs*) of serial shifter (SS). Then the median-comparator (*Tmc*) releases three numbers, and finally the equality-detector (*Ted*) releases the decision to initialize the counter or keep counting (i.e. enable/reset the counting process). The equality-detector time (*Ted*) occurs in parallel with re-store the numbers of the SET_i and the pivot number. Since re-storing the numbers in serial shifter is only storing within a DFF access time, we consider the path of equality-detector

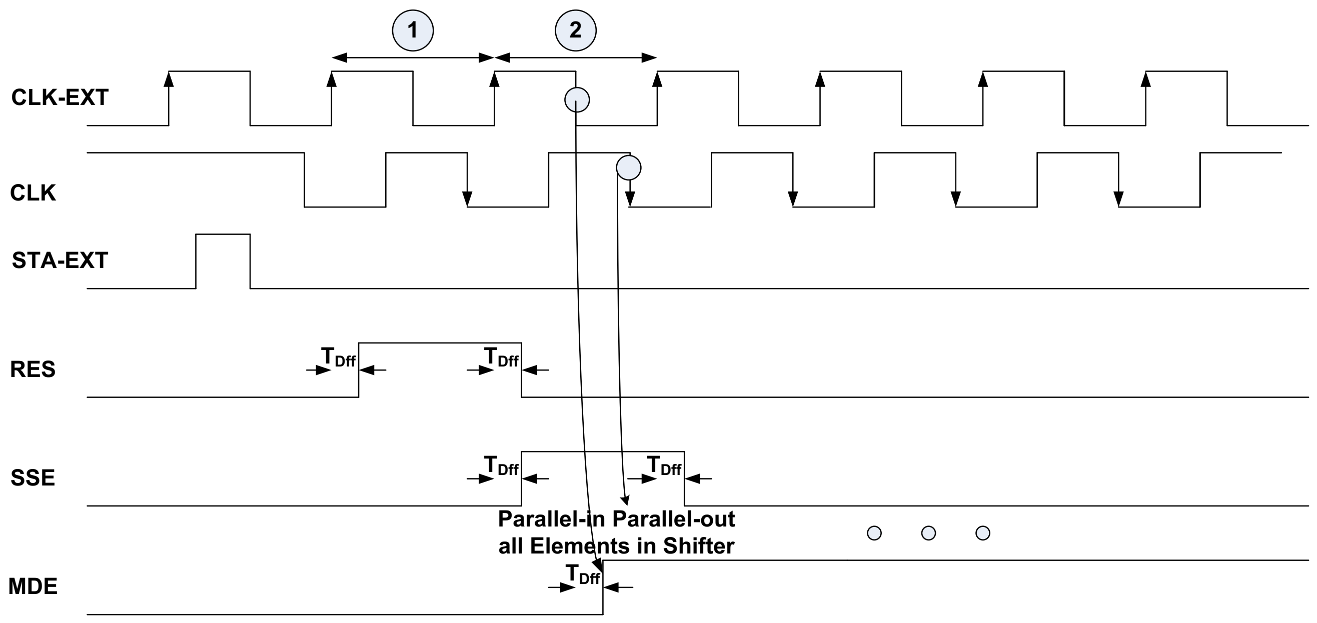

Fig. 8 Timing diagram for the pivot-initialize phase

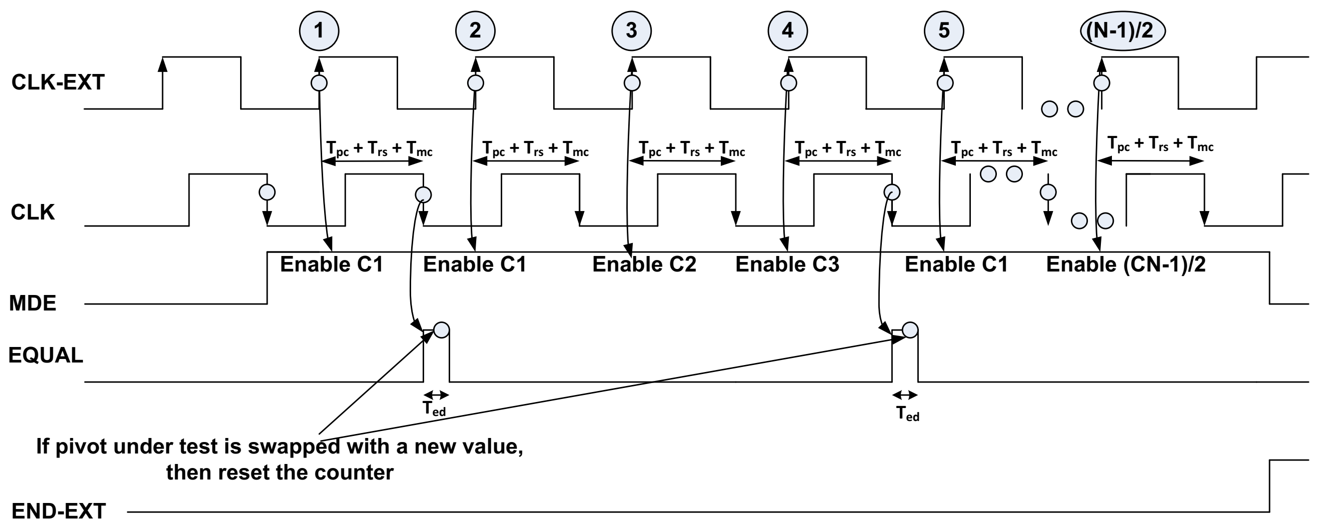

Fig. 9 Timing diagram for the median-evaluate phase

(*Ted*) to be the critical path. Therefore, the median-evaluate phase critical path can be summarized as follow:

$$Tmedian - evaluate = Tpc + Trs + Tmc + Ted \tag{4}$$

Clearly, the median-evaluate phase determines the maximum clock frequency.

The number of cycles in this phase depends on the number of iterations since each iteration is executed within one clock cycle. The number of iterations is actually bounded by Eqs. (1)–(2) derived in Sect. 3. For example, a 3 × 3 scan matrix, which contains nine numbers, requires a maximum of 23 iterations (upper-bound), while the minimum number of iterations (lower-bound) is 8. Hence, the 3 × 3 scan matrix requires between (8 + 2) to (23 + 2) cycles to compute the median of a sequence of nine numbers. In most cases, the input sequence will require a number of cycles closer to the lower-bound, as the exhaustive simulations show in Sect. 2 (see Fig. 3).

4.2 Control unit operation

The control unit receives input signals from the data path and sends the appropriate control signals back to the data path; hence, it synchronizes the iterations of the data path. The control unit also receives the external and handshaking signals in order to interface the proposed design with the external components that are producing the

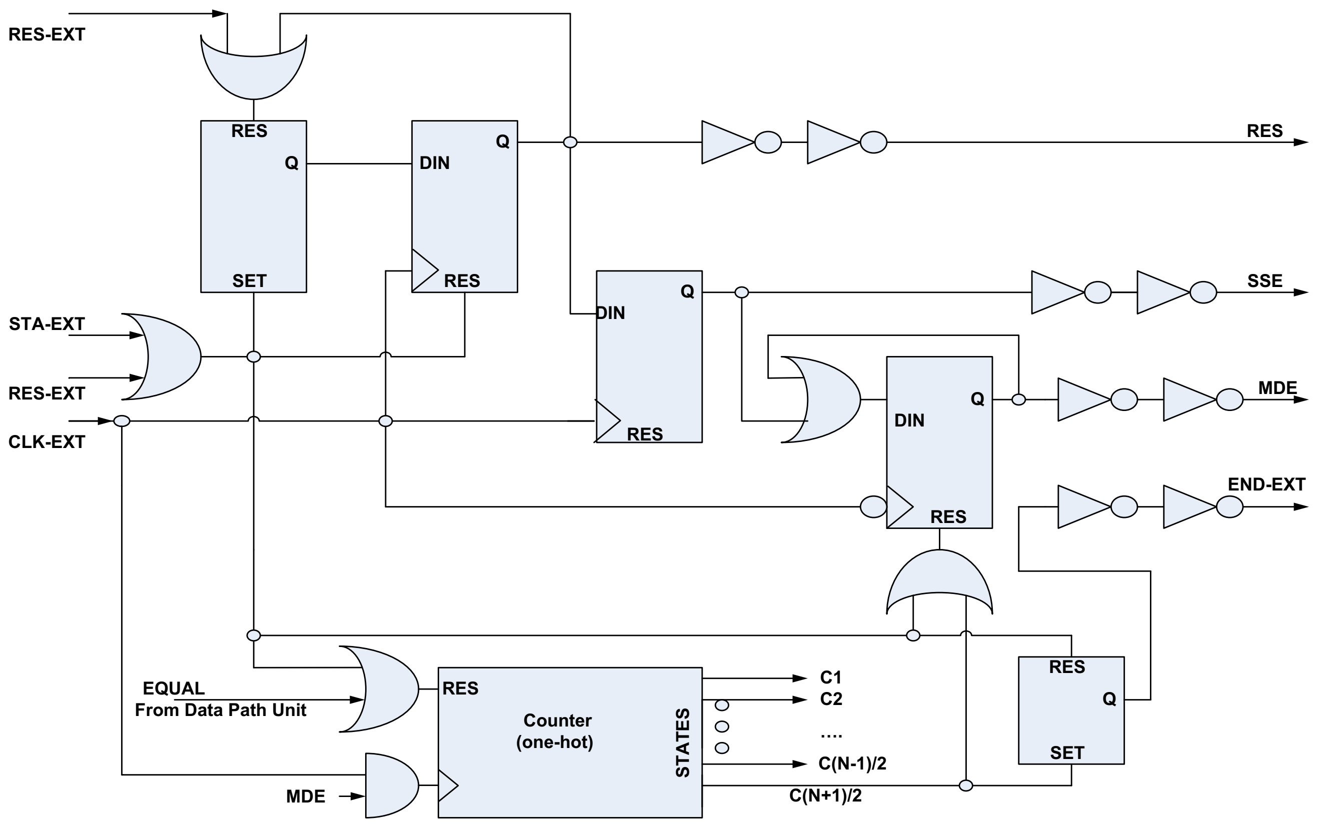

Fig. 10 Control unit circuit for the proposed efficient median hardware circuit

sequence of numbers. There are several methods for designing the control unit [19], but prior work on median-finding hardware [10–18] stopped at the data path level, providing no details on the control logic design. Here, we present the complete control unit design in order to provide a fully functional and assessable implementation with all the required components and signals, leaving no details out. The detailed presentation illustrates the overall simplicity of our design.

Based on the timing diagrams given in Figs. 8 and 9 for the two phases, the complete control unit is depicted in Fig. 10. The control unit is relatively simple, using only five DFFs and a counter in order to generate the timing signals required for the two phases of our proposed design. This simplicity of the control unit arises from the simplicity of the data path unit which obviates the need for complex handshake signals and special high-cost components.

5 Imulation and experimental results

In order to facilitate comparison with existing designs for image median filter applications, we implemented, tested, and verified our median algorithm hardware architecture using example systems with input lengths $N = 9$ and $N = 25$, each input being 8 bits wide. These example system sizes are used in many prior digital hardware median integrated circuits (ICs) for DSP SOC designs [12−18]. We designed our proposed median-finding unit at the CMOS transistor level using 90 nm TSMC technology with a 1 V power supply [20]. We obtained timing delay values, total power consumption, and total transistor counts using HSPICE simulations [21].

Table 6 summarizes all of the components' delay times and associated transistor counts. The median-comparator comprises of three comparators, whose designs are based on parallel prefix-tree structure [22]. The 4-bit one-hot counter used in control block is implemented based on a state-look ahead logic counter structure [23] with front-end decoder states, giving an equal timing delay to all states. The equality-detector comprise of only one DFF and few gates. Additionally, the rest of the components are simple DFF registers, tri-state buffers, and simple multiplexers, where we assume all of these simple components have the same delay time, $T_{TB} = T_{MUX} = T_{DFF}$, since they have one logic delay. From Table 6, the total transistor count is about 3222 and 6294 for 3×3 and 5×5 scan matrix, respectively, which represent very small circuits.

Based on the time delay Eqs. (3) and (4) and latencies reported in Table 6, we show that the pivot-initialize

Table 6 Component time delays and transistor counts assuming 90 nm technology

Component	3 × 3			5 × 5		
	Specification	Delay (ns)	Transistor count	Specification	Delay (ns)	Transistor count
Parallel-in/out serial register	9 registers 8 DFFs per register	0.06	1440	25 registers 8 DFFs per register	0.06	4000
Tri-state and multiplexers	8-bit each 18 buffers	0.06	288	8-bit each 50 buffers	0.06	800
Median comparator	8-bit (R) 8-bit (L) 8-bit (M)	0.21	1234	8-bit (R) 8-bit (L) 8-bit (M)	0.21	1234
Equality detector	1 DFF	0.12	20	1 DFF	0.13	20
DFFs for control	5 DFFs	0.06	100	5 DFFs	0.06	100
One-hot counter	3-bit Module 4	0.06	120	4-bit Module 12	0.06	160

Table 7 Comparison between prior works and our proposed efficient median

Recent works	Image time (ms) 512 × 512	Maximum freq. (MHz)	Power (mW)	Structure and complexity
[15]	15.72/12.6	90/125	12.3/15.3	Sorting unit Determine threshold NxN registers
[16]	12/10.7	50/70	Not available	Pipelining architecture Cascaded comparators Modify shear sort
[17]	9.3/8.6	330/350	Not available	Pipelining architecture Quantum representations Long accumulative addition Comparisons
[18]	6.1/5.2	625/714	9.8/5.3	Rank sort Token ring Additional hardware
Proposed	1.394/0.364	2000	3.24/1.66	Non-sort based Short critical path Minimum hardware Suitable for set of N < 100

phase's clock cycle time is $CLK_{pivot\text{-}initialize} < 0.1$ ns and the median-evaluate phase's clock cycle time is $CLK_{median\text{-}evaluate} < 0.37$ ns. These timings result is an approximate conservative clock frequency of 2 GHz. Consequently, the total time for a scan matrix of 3 × 3, which has 9 numbers, will be 10–25 clock cycles (8 + 2 to 23 + 2), where 2 is added due to pivot-initialize phase, assuming the data are initialized in the serial shifter as parallel-in/parallel-out mechanism. For a 5 × 5 scan matrix with 25 numbers, the delay ranges from 26 to 266 clock cycles due to reported mathematical iterations Eqs. (1) and (2).

We have compared our efficient median-finding design with prior works with respect to hardware complexity, maximum operating frequency, switching activites and power consumption, and the total time needed to scan a 512 × 512 image with 8-bit pixel size. Table 7 shows the reported data indicating the ranges between the best and worst case values (best/worst), attributable to 3 × 3 and 5 × 5 scan matrices for the whole image. Designs in Refs. [15, 16] are based on conventional sorting algorithms with classical cell circuit structures array and pipelining with merging memories. These design choices increase the circuit costs, power, and latencies; besides, memory time bottleneck limits the maximum running frequency. References [17, 18] are based on rank sort of the numbers within the sequence, which actually store the numbers based on their weighted address index. Frequent access to registers and memory is required, thus limiting the clock frequency and increasing power consumption. In a variation, Ref. [18] attempts to reduce the power consumptions by tapping the index of input numbers instead of working on the values by using token-ring circuitry. However, the design still uses

Table 8 Our median hardware against quick sort on high performance CPU computations

Algorithm	Platform	Average median 3 × 3 scan time (msec) for image frame 800 × 600	Name
Ref [24]	Multi-core CPU	200	Quick sort
Proposed	ASIC	0.66	Sorting-free median

large memory (registers) and a pipelined architecture that tend to increase the design complexity and power draw. Besides, the design uses accumulative operations and consecutive comparisons that reduces the maximum operating speed.

Our design alleviates the speed bottleneck components (i.e., memory) and power-intensive components (i.e., pipelining), leading both to higher clock frequency and lower power requirements. Additionally, the proposed design uses simple components with simple routing directions that minimizes the effect of loading and parasitics resistor–capacitor components. All components are gated with enable signals to restrict the activities to the needed components for the operation. We record the worst-case arrangemt of 3 × 3 and 5 × 5 scan matrices that give worst case iterations, assuming, pessimistically, that every scan matrix of the image is a worst-case arrangements. Even with such an unrealistic worst-case scenario, our total processing time for a 512 × 512 image is the fastest among the designs studied. We have for 3 × 3 scan matrix the total time $(512 \times 512)*(23 \ +2)*0.5 \ *10^{-9}/9 = 0.364$ ms and for 5 × 5 scan matrix $(512 \times 512)*(264 + 2)*0.5*10^{-9}/25 = 1.394$ ms.

It is also instructive to compare our design with data reported in the literature for median algorithms [24] that harness the resources of powerful CPUs. Table 8 reports the execution time using quick-sort3 for evaluating the median on image of size 800 × 600 pixels using a scan matrix of size 3 × 3 that harness the resources of a multi-core CPU, where the memory initialization time to load the frame is not counted. By comparison, our proposed hardware median-finding circuit gives a total time for the same image of 0.66 ms, assuming worst case iteration for every scan 3 × 3 matrix. This result shows the impressive speed advantage of our design (about 200x) running on custom SOCs ICs against other median algorithms that run on high-performance CPU platforms.

6 Conclusion

In this paper, we proposed a novel sorting-free median algorithm, along with the associated hardware implementation. Our median-finding design exhibits linear, $O(N)$, time complexity on average for sequences of length 100 or less. The worst-case quadratic, $O(N^2)$, complexity is only encountered in no more than 0.15 fraction of possible input data arrangements. This small fraction is further diminished to 0.035 or even 10^{-5} in the practically important cases when the sequence length exceeds 5. The elegant mathematical algorithm limits the design complexity to only a few registers, comparators, and a counter with simple basic repeated path operations, making the design reconfigurable and fast due to its short critical path.

Furthermore, the design has a small geometry that is targeted for DSP SOCs applications on hand-held devices and home entertainment image processing. A complete end-to-end design is presented along with control unit and all associated signal exchanges that facilitate the interfacing mechanism with neighboring components. A complete image can be scanned by 3 × 3 or 5 × 5 matrices of 8-bit pixel values, where the median is derived and written on every scan matrix of the complete image.

Our sorting-free median-finding circuit is compared with sorting-based software algorithm (i.e. Quick-Sort3) to scan regular image of size 800 × 600 8-bit pixel using 3 × 3 scan matrix, where our design was determined to outperform the software algorithm by about 200X. The design is implemented on ASIC standard CMOS library components of 90 nm TSMC technology with 1 V power supply for computing the median of 3 × 3 and 5 × 5 matrices, where the running frequency is 2 GHz, giving the relatively small transistor counts of 3202 and 5203, respectively. The small transistor counts imply low power consumption on the order of less than 2 mW for a complete scan image.

References

1. Ross, J. C. (2011). *The image processing handbook* (6th ed.). Boca Raton, FL: CRC Press.
2. Nair, M. S., & Mol, P. M. A. (2013). Direction based adaptive weighted switching median filter for removing high density impulse noise. *Computers and Electrical Engineering, 39,* 663–689.
3. Monajati, M., Fakhraie, S. M., & Kabir, E. (2015). Approximate arithmetic for low-power image median filtering. *Circuits Systems and signal Process, Springer, 34*(10), 3191–3219.
4. Martinez-Hinarejos, C. D., Juan, A., & Casacuberta, F. (2000). Use of Median String for Classification. In *Proceedings 15th International Conference on Pattern Recognition*, (vol. 2, pp. 903–906).
5. Battiato, S., Cantone, D., Catalano, D., Cincotti, G., & Hofri, M. (2000). An efficient algorithm for the approximate median

selection problem. *Algorithms and complexity, 1767*, 226–238. **Springer, LNCS**.
6. Musser, D. R. (1997). Introspective sorting and selection algorithms. *Software practice and experience, 27*(8), 983–993.
7. Bandala-Hernandez, H. C., Rocha-Perez, J. M., Diaz-Sanchez, A., Lemus-Lopez, J., Vazquez-Leal, H., Diaz-Armendariz, A., et al. (2016). Weighted median filters: An analog implementation. *Integration, the VLSI Journal, 55,* 227–231.
8. Siskos, S. (2010). Low voltage analog median filters implementation. In *IEEE International Conference on Imaging Systems and Techniques*, (pp. 166–170).
9. Jendernalik, W., Blakiewicz, G., Jakusz, J., & Szczepanski, S. (2013). A nine-input 1.25 mW, 34 ns CMOS analog median filter for image processing in real time. *Analog Integrated Circuits and Signal Process, 76,* 233–243.
10. Chen, C. T., Chen, L. G., Hsiao, J. H. (1995). A hardware-oriented design for weighted median filter. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/ VLSI '95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scale*, (pp. 441–445).
11. Fuguo, D., Hui, F., Da, Y. (2010). A novel image median filtering algorithm based on incomplete quick sort algorithm. *International Journal of Digital Content Technology and its Applications, 4*(6), 79–84.
12. Satyanarayana, V., Srividya, S., & Yedukondalu, U. (2013). High throughput two-dimensional median filters on FPGA for image processing applications. *International Journal of Engineering research and Applications (IJERA), 3*(4), 1946–1949.
13. Wei, P., Zhang, L., Ma, C. Yeo T. S. (2010). Fast median filtering algorithm based on FPGA. In *IEEE 10th International Conference on Signal Processing Proceedings (ICSP),* (pp. 426–429).
14. Smith, J. L. (1996). Implementing median filters in XC4000E FPGAs. *Xilinx XCELL Journal, issue23,* 16.
15. Matsubara, T., Moshnyaga, V. G., Hashimoto K. (2010). A low-complexity and low power median filter design. In *International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS 2010)*, (pp. 1–4).
16. Jerose, G. I., & Selvi, R. S. (2013). Novel highspeed architecture for median filter. *International Journal of Science and Research (IJSR), 2*(4), 57–61.
17. Cadenas, J. O., Megson, G. M., & Sherratt, R. S. (2015). Median filter architecture by accumulative parallel counters. *IEEE Transactions on Circuits and Systems II: Express Briefs, 62*(7), 661–665.
18. Chen, Ren-Der, Chen, Pei-Yin, & Yeh, Chun-Hsien. (2015). A low-power architecture for the design of a one-dimensional median filter. *IEEE Transactions on Circuits and Systems II: Express Briefs IEEE, 62*(3), 266–270.
19. Hayes, J. P. (1994). *Computer architecture and organization* (2nd ed.). New york: McGraw-Hill.
20. Taiwan Semiconductor Manufacturing Corp. (2005). 90 nm CMOS ASIC Process Digests.
21. HSPICE, Synopsys, (2010) http://www.synopsys.com.
22. Abdel-Hafeez, S., Gordon-Ross, A., & Parhami, B. (2013). Scalable digital CMOS comparator using a parallel prefix tree. Journal of. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 21*(11), 1989–1998.
23. Abdel-Hafeez, S., & Gordon-Ross, A. (2011). A gigahertz digital CMOS divide-by-N frequency divider based on a state look-ahead structure. *Journal of Circuits, Systems, and Signal Processing, 30*(6), 1594–1572.
24. H. Inoue, T. Moriyama, H. Komatsu &T. Nakatani (2007) AA-SORT: A new parallel sorting algorithm for multi-core SIMD processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, (pp. 189–198).

Saleh Abdel-hafeez received a Ph.D. in computer engineering from the University of Texas at El Paso and M.S. from New Mexico State University. He was a senior member of technical staff at S3 Inc. and Viatechnologies.com at the area of mixed-signal IC design. He also was Adjunct Professor of computer engineering at Santa Clara University from 1998 to 2002. He has three US patents, numbered 6265,509 and 6356,509, with S3 Inc. His current research interests are in the areas of high speed ICs, computer arithmetic algorithms, and mixed-signal design. Dr. Abdel-hafeez is currently the Chairman of Computer Engineering Department at Jordan University of Science and Technology.

Behrooz Parhami (Ph.D., 1973, University of California, Los Angeles) is Professor of Electrical and Computer Engineering, and former Associate Dean for Academic Personnel, College of Engineering, at University of California, Santa Barbara, where he teaches and does research in computer arithmetic, parallel processing, and dependable computing. A Fellow of IEEE, IET, and British Computer Society, and recipient of several other awards (including a most-cited paper award from *J. Parallel & Distributed Computing*), he has written six textbooks and more than 280 peer-reviewed technical papers. Professionally, he serves on journal editorial boards and conference program committees and is also active in technical consulting.

Arwa Damir received the B.Sc. Degree of Computer Engineering in 2001 from Jordan University of Science and Technology, Irbid, Jordan. He is currently Full-time lab lecturer at the same University. His research interests include VLSI components design and Computer Architecture.