Dual port memory based heapsort implementation for FPGA

Wojciech M. Zabołotny[a]

[a]Institute of Electronic Systems, Warsaw University of Technology, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland

ABSTRACT

This document presents a proposal of implementation of the Heapsort algorithm, which utilizes hardware features of modern Field-Programmable Gate Array (FPGA) chips, such as dual port random access memories (DP RAM), to implement efficient sorting of a data stream.
The implemented sorter is able to sort one data record every two clock periods. This throughput does not depend on the capacity of the sorter (defined as number of storage cells in the sorter).
The mean latency (expressed in sorting cycles - each equal to two clock periods) when sorting the stream of data is equal to the capacity of the sorter.
Due to efficient use of FPGA resources (e.g. data are stored mainly in internal block RAMs), the complexity of the sorter is proportional to the logarithm of sorter capacity. Only the required RAM size is linearly proportional to the sorter capacity.
The proposed sorter has been tested in simulations and synthesized for real FPGA chips to verify its correctness.

Keywords: FPGA, sorting, Heapsort, parallel processing, dual port memory

1. INTRODUCTION

Sorting is often an essential part of data processing. In many applications it is possible to use software implementations which are widely available both in commercial and free versions. However in some high speed applications it is desirable to perform sorting in hardware, utilizing possibility of parallel operation. One of algorithms with optimal complexity $O(n\log(n))$ and minimal storage requirements is the Heapsort.[1] There exist some descriptions of hardware based sorters using the Heapsort algorithm. One example is presented in [2], but it is not fully parallel. The VHDL implementation of Heapsort is publicly available at least since 1996,[3] but this code is not suitable for synthesis.

It seems that no synthesizable implementations of Heapsort in FPGA are widely available. Additionally features available in modern FPGAs such as availability of multiple block RAMs, suitable for use as dual-port memories[4] allow to develop highly parallelized, efficient implementation of the Heapsort algorithm. Therefore I decided to develop such implementation well suited for sorting of stream of data – task often needed in multichannel data acquisition systems.

2. USE OF HEAPSORT FOR SORTING OF STREAM OF DATA

Let's assume, that we have a multichannel detector system, which during a long time period produces data records marked with the time-stamp. The data from different channels are serialized into a single data stream containing K records, but due to different occupancies of particular detector channels, and due to different link latencies, the data records in the serialized stream are out of order regarding their time-stamps. Let's denote the time-stamp of the record on position k in the data stream as t_k. We need a sorter to sort the data according to their time stamps before further processing, so that $\forall n \in \mathbb{Z} \cap [1, K-1] : t_{k+1} \geq t_k$.

From the sorting point of view the important parameter of our data stream is the maximum distance between unsorted data records, defined as:

$$D_{max} \stackrel{\text{def}}{=} \max\{d \in \mathbb{Z} \cap [1, K-1] : \exists k \in \mathbb{Z} \cap [1, K-d] : t_{k+d} < t_k\} \quad (1)$$

Further author information: (Send correspondence to W.M.Z.)
W.M.Z.: E-mail: wzab@ise.pw.edu.pl, Telephone: +48 22 234 7717

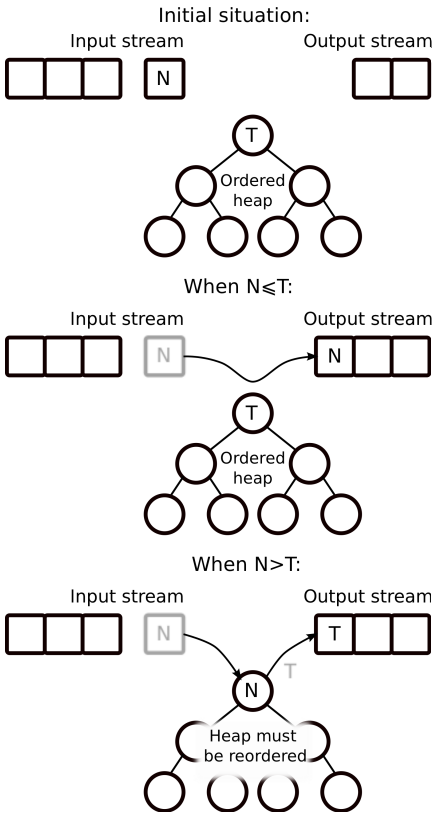

Figure 1. Use of heap sorter to sort stream of data. The record N is taken as the next record in the input stream, and is compared with the T record on the top of heap.

The value of the D_{max} may be found analyzing the properties of the particular detector and link system. Such a data stream may be sorted using a heap sorter containing $M \geq (D_{max} - 1)$ storage elements. The number of storage elements M will be further denoted as *sorter capacity*.

To simplify further description let's denote the value of the time-stamp of the record as the *value of the record*. Record A is greater than record B, if the time-stamp of record A is greater than time-stamp of record B. In the sorter we use a binary-heap, organized so, that in the ordered state, the value of record stored in a node is not bigger, than values of records stored in its children.

To perform sorting of our data stream we should first fill the heap with *initial records* with the "smallest possible" value (it could be done e.g. automatically after reset of the sorter). We can for example reserve the time-stamp equal to 0 for this purpose. When sorted records are received from the output of the sorter, such records with time-stamp equal to zero will be quietly ignored. Of course the heap filled with those *initial records* is in ordered state.

Then we should sequentially process our data. In each sorting cycle we take one data record from the input data stream, and compare it with the record available on the top of the heap. If the input record is smaller or equal to the record on the heap's top, we output the input record. If the input record is greater than record on the heap's top, we output the record from the top of the heap, and put the input record on top of the heap. In the second case we need to reorder the heap (see Fig.1). In the next sections, I'll show how to implement in the FPGA a heap which may be quickly reordered using the parallel processing. With such approach, the mean latency (mean time spent by the data record inside of the sorter) expressed in sorting cycles is equal to the capacity of the sorter. When all data stream records are read, we yet need to extract the biggest records, which are still stored in the heap. For that purpose we should use the *end records* with "highest possible" value. E.g. if we use l bits to store the time-stamp, we may use the time-stamp equal to $2^l - 1$ as the value of the

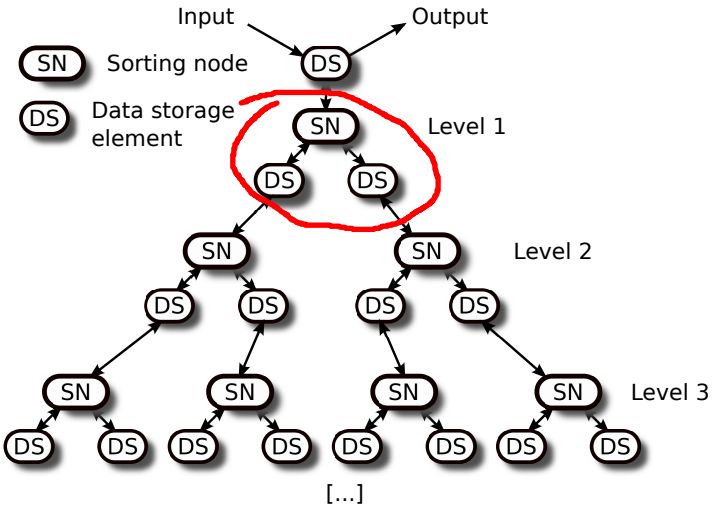

Figure 2. The simplest hardware architecture implementing parallel Heapsort in the hardware.

end record. Of course such time-stamp should never occur in our data stream.

After the last record from the data stream is processed, we keep providing the *end records* to the input, until the first *end record* is seen on the output. When first *end record* reaches the output, the whole stream is processed.

3. PARALLEL IMPLEMENTATION OF HEAPSORT IN HARDWARE

I must state here that the final concept developed and presented in this paper is very similar to the implementation presented in Ref. 5, even though I have developed my solution independently. Additionally due to use of features available in the new FPGAs, the sorter presented in this paper is able to sort one record per 2 clock periods, while solution described in Ref. 5 needs three "phases" additionally divided into 2 "subphases" in each sorting cycle.

Below I'll present how my concept of synthesizable parallel heap-sorter was developed.

The parallelly operating heap may be relatively easy implemented in the hardware. Because we always may use any $M \geq (D_{max} - 1)$, let's concentrate on heaps containing $M = 2^n - 1$ storage elements, which have the most regular structure. Such a heap may be presented as a storage elements connected with the "sorting nodes", organized in $J = n - 1$ "levels" as shown in Fig.2. Each *sorting node* is connected to three *data storage elements* – the *upper* one (U), and two *lower*: the *left* (L), and the *right* (R). The relationship between the number of levels J and capacity of the sorter M is the following: $M = 2^{J+1} - 1$.

When (according to the algorithm described in the previous section) the new record replaces the value stored in the upper storage element of the sorting node (the value is *updated*), the node gets *activated* and performs operations shown in the Fig.3.

At first glance, it may seem, that in the worst case (when the new input record finally should be stored in the last level), we need to activate consecutively sorting nodes in all levels. So the sorter with J levels, with capacity $M = 2^{J+1} - 1$ would require J steps to fully reorder the heap. (It is not defined yet, how many clock pulses do we need to complete one step. This will be discussed later.)

However the more thorough analysis of operations shown in the Fig.3 reveals, that the continuous operation of the sorter requires operation in sorting cycles containing only two steps:

1. In the first step the level is inactive, and data storage elements it is connected to may get updated by sorting nodes from the previous and from the next levels.

2. In the second step, the sorting node (SN) compares data and swaps them if it is necessary, notifying the next level node about the update.

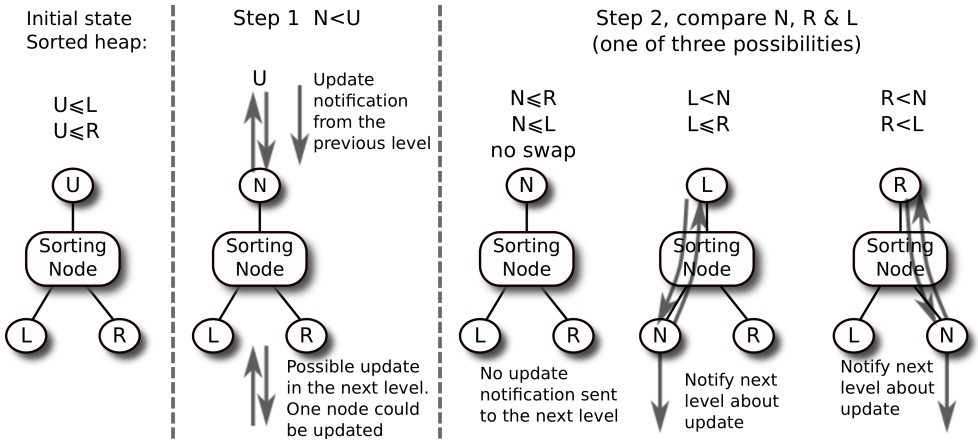

Figure 3. Operations performed by each sorting node. Initially the sorting node is part or the sorted heap ($U \leq L$ and $U \leq R$). The figure shows flow of data for different possible relations between the value of the new record (N) and values stored in the storage elements connected to the sorting node (U, L, R). When $N \geq U$ nothing happens.

Please note, that reordering of data in the level j does not disturb heap ordering on higher (previous) levels. When the sorting node in the level j swaps the data, the new, *bigger* record is transferred to the storage element below this level, and the record from this storage element is transferred to the storage element above this level. However, if the heap was previously ordered, then this upward transferred record is granted to be *lower* than the U record of the parent node in the level $j-1$. Therefore in the Fig.3 only lower (next) levels must be notified about the update.

The only requirement to avoid conflicts is that two neighboring levels do not try to swap data simultaneously. Therefore when the "odd" levels perform step 1, the "even" levels should perform step 2, and when the "odd" levels perform step 2, the "even" levels should perform step 1. Therefore the structure shown in the Fig.2 is able to accept one input record and provide one sorted output record every two steps.

Unfortunately such implementation is very resource hungry. For sorter with capacity of $M = 2^{J+1} - 1$ we need $(M-1)/2$ sorting nodes and M data storage elements. Additionally the resources are used very inefficiently – as the previous analysis has shown, in the particular level only one sorting node may be notified about the update and therefore only one sorting node in each level may be activated in each sorting cycle. This allows us to use only one sorting node per level, and to use RAM memory instead of independent registers as storage elements. The transformation of our simplest solution into RAM based sorter is shown in the Fig.4.

However, because the storage elements are accessed by two sorting nodes (belonging to two neighboring levels), instead of standard RAM, it is better to use the dual port RAM (DP RAM) (see Fig.5) as the storage elements.

4. IMPLEMENTATION OF HEAPSORT WITH DP RAM MEMORIES

In this section I'll present possible DP RAM based implementations of the Heapsort algorithm, starting from the simplest one, which is also the easiest to understand.

4.1 Basic implementation of Heapsort with DP RAM memories

The simplest version of Heapsort sorter using the DP RAMs is shown in the Fig.6. In this structure below the level j we have the DP RAM with capacity of $P_j = 2^j$ data records. The lower half of this DP RAM is considered to store the *left* (L) records, while the upper half stores the *right* (R) records. The same DP RAM is also connected to the next level sorting node, providing it with the *upper* (U) records. When the sorting node from level j, due to data swapping, updates the record at address a_j, it notifies the lower level node about it, sending this address. Then the next node compares this new record with its *lower* memory content at addresses a_j (L) and $a_j + P_j/2$ (R), and takes a decision, whether swapping of data is needed.

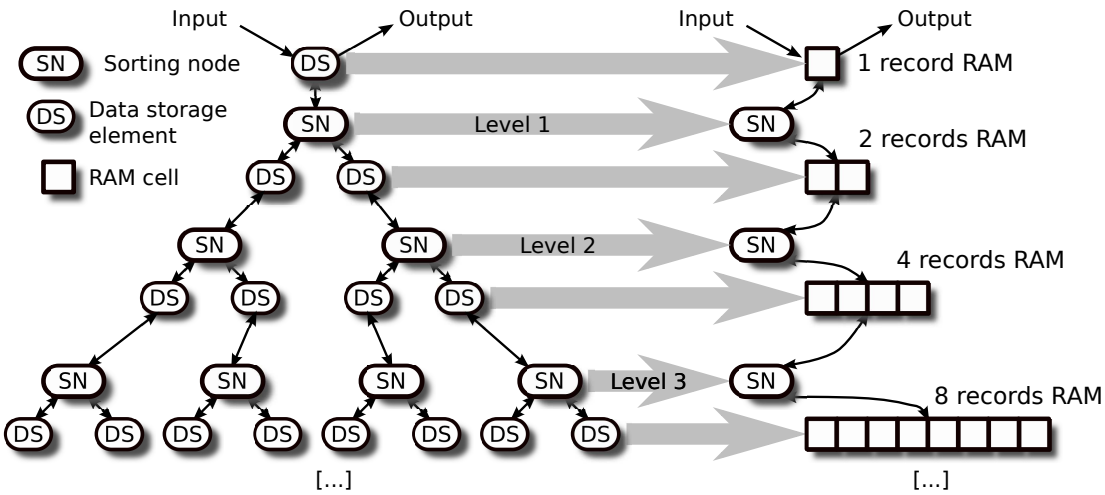

Figure 4. Transformation of heap-sorter based on independent storage elements into heap-sorter based on RAM. In fact the first level sorting node handles also input and output from the sorter, and therefore it has slightly different structure, incorporating it's upper RAM implemented as internal register. Due to optimizations during synthesis also other small RAMs may be synthesized as registers.

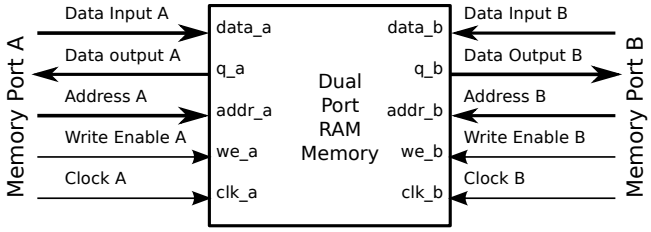

Figure 5. Dual port random access memory - DP RAM.

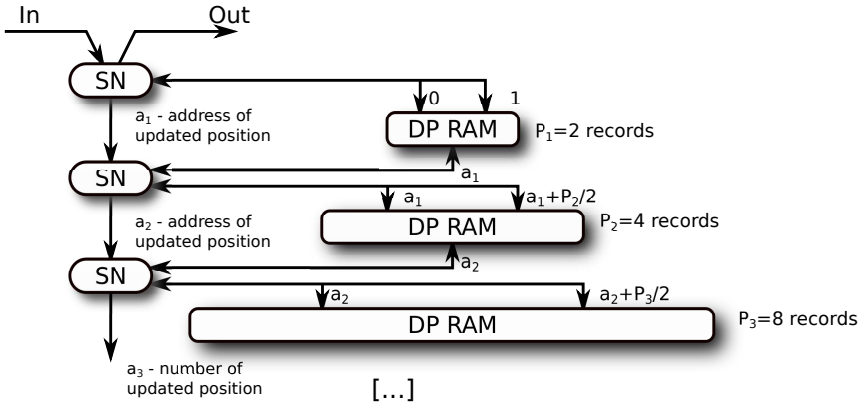

Figure 6. The simplest structure of Heapsort sorter with DP RAM memories.

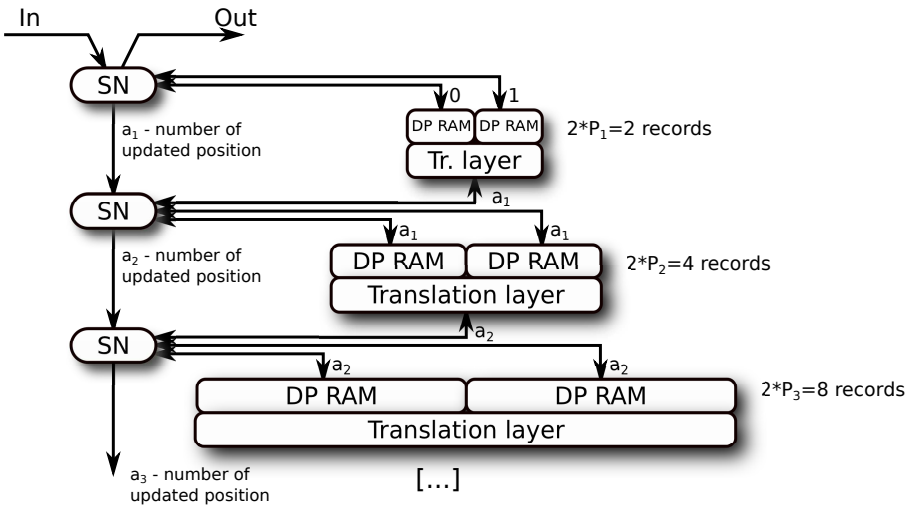

Figure 7. Structure of Heapsort sorter with duplicated "lower" DP RAM memories. This structure allows to read both "L" and "R" records in the same clock cycle.

In this implementation for sorter with capacity of $M = 2^{J+1} - 1$ we need only J sorting nodes and J DP RAMs with different sizes. Such solution makes reasonable use of resources available in the FPGA, but requires too many clock pulses per step. Let's analyze it more thoroughly.

In "step 2" (see Fig.3) the sorting node must compare N, L and R records. In a single clock period we can read either records N and R, or N and L, as they are read from different memories, but it is impossible to read in the same clock period records L and R, as they are read from the same memory. This problem alone forces us to use at least two clock periods per algorithm step. In fact there is yet another problem associated with the DP RAM timing, which also forces us to use at least two clock periods per algorithm step, but this will be analyzed and solved later.

4.2 Implementation of Heapsort with dual DP RAM memories

The only method to allow simultaneous reading of both L and R records, is to have separate *left* and *right* DP RAM memories. The resulting structure is shown in the Fig.7.

Of course in this implementation we need to provide an additional logic (*Translation layer*) allowing to present both memories to the next level as a single DP RAM with two times bigger capacity.

In this structure we can read all three N, L and R records in the same clock period, so theoretically we could work with single clock period per step (sorting one record per two clock pulses). However one problem still remains - the DP RAM timing. I will address this problem in the next subsection.

4.3 Implementation of Heapsort with dual DP RAM memories and bypass channels

The DP RAMs available in FPGAs work synchronously. To write a new record we have to provide the address and data in one clock period, and the next active clock edge will store the data. To read the data we have to provide the read address in one clock period, and after the active clock edge (i.e. in the *next* clock period) the read data will be available (see cycles 6,7 and 8 in Fig.8). If we want to achieve operation with one clock period per step, whenever the upper level sorting node decides to update (swap) the record at address a_j, we should immediately use this address to read the N record from upper memory and L and R records from lower memories. Unfortunately the DP RAMs available in contemporary FPGAs do not allow to read via one port the memory cell which has just been written via the second port (see cycle 3 in Fig.8). Similar conflict may also occur, when the updated U record in level j must be immediately read as an R or L record in level $j-1$. To avoid such conflicts, we need to provide alternative communication channels (let's denote them as *bypass channels*), to pass between levels the information about the updated record and its address.

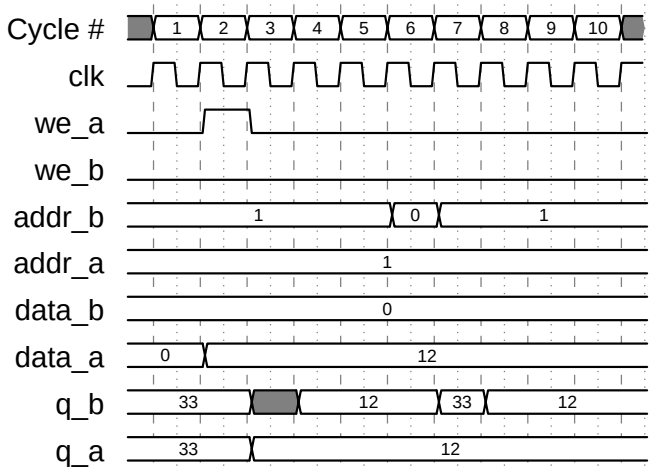

Figure 8. Simulation showing the operation of DP RAM. All memory cells were initialized to “33”. In pulse 3 the “12” byte was written through port A to cell 0. Please note, that port B was continuously reading this cell, and the newly written byte is accessible only in pulse 5. In pulse 4 the output of port B was undefined. The written byte is available on the port A output right in the next (4^th) pulse (“read after write” behavior of DP RAM).
In clock pulses 6 and 7 the port B address was changed. Please note, that the byte read from the newly selected memory cell is available on port B output in the next clock cycle.

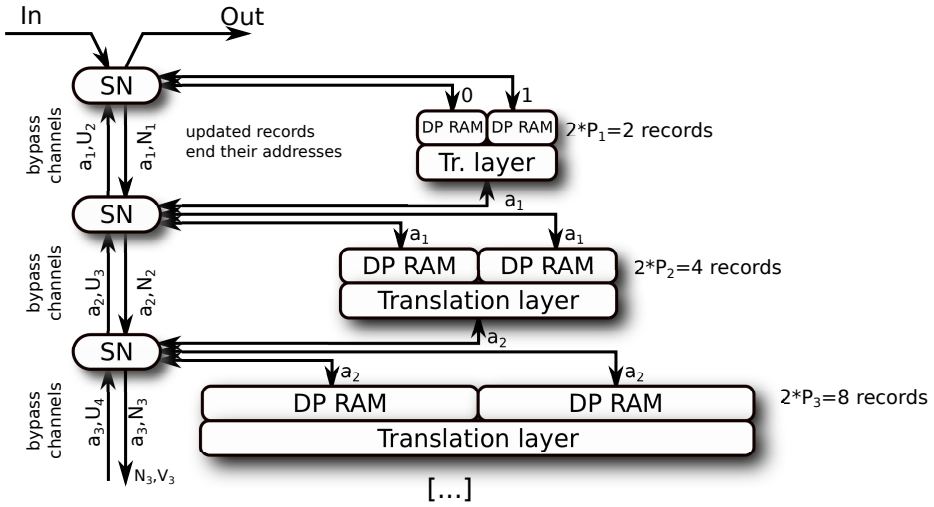

Figure 9. Structure of Heapsort sorter with added "bypass channels" allowing to resolve conflicts, when the same memory cell should be written in one clock cycle from one port and read from another port.

The resulting structure is shown in the Fig.9. The operation of sorting nodes is modified in the following way. Whenever the sorting node is activated due to update of its *U* record, it doesn’t read this record from DP RAM, receiving it instead via the *bypass channel* from the upper level sorting node.
Similarly when the sorting node reads its *R* and *L* records, it checks if the address to be read is equal to the address of just updated record, transmitted via the *bypass channel* from the lower level sorting node. If yes, then the appropriate record is read from that *bypass channel* instead of the *left* or *right* DP RAM.

With this modification the sorter is able to operate with one clock cycle per algorithm step, sorting one data record per two clock cycles.

Table 1. Results of synthesis of the 15 level heap sorter in the Virtex-6 XC6VLX75T-3FF484 chip.

Device Utilization Summary		
Number of occupied Slices:	2,484 out of 11,640	21%
Number of RAMB36E1/FIFO36E1s:	92 out of 156	58%
Number of RAMB18E1/FIFO18E1s:	24 out of 312	7%
Speed parameters		
Maximum clock frequency		100.05 MHz

Table 2. Results of synthesis of the 10 level heap sorter in the Virtex-6 XC6VLX75T-3FF484 chip.

Device Utilization Summary		
Number of occupied Slices:	1,860 out of 11,640	15%
Number of RAMB36E1/FIFO36E1s:	18 out of 156	11%
Number of RAMB18E1/FIFO18E1s:	18 out of 312	5%
Speed parameters		
Maximum clock frequency		125.2 MHz

5. TESTS OF THE SORTER

The described structure of the sorter has been implemented in a synthesizable behavioral VHDL code, and verified in functional simulations, using the GHDL[6] simulator. The input data stream with desired maximum distance between unsorted records (the D_{max} parameter defined in equation 1) were generated randomly with a program written in Python.[7] The simulations have proved the correct operation of the sorter.

Additionally the sorter was synthesized for implementation in Virtex 6[8] FPGAs. For synthesis we assumed, that the data record contains 18 bits of time-stamp and 20 bits of payload. The synthesis was performed for the smallest XC6VLX75T-3FF484 chip. However even in this, relatively small chip it was possible to implement sorter with 15 levels, and with capacity $2^{15+1}-1 = 65535$ records, capable to sort data streams with maximum distance between unordered records equal to 65536. The achieved maximum clock frequency was equal to 100 MHz, so the sorter may sort up to 50 million of records per second. When number of levels was decreased to 10 (capacity of 2047 records), the maximum clock frequency increased to 125 MHz. The detailed results of resource consumption are summarized in the Tables 1 and 2. As we can see the occupancy of slices increases similarly to the number of levels ($15/10 = 1.5$ and $2484/1860 = 1.335$), while the occupancy of memories increases significantly faster (in theory occupancy of memories should be proportional to the sorter's capacity, but the fact that not all memory blocks are fully utilized disturbs that relationship).

6. CONCLUSIONS

Presented implementation of the Heapsort algorithm in FPGA chips allows to achieve sorting of long data stream with high throughput and minimal possible mean latency. Use of DP RAM memories allows to store data records with reasonable consumption of internal routing resources, which allows to maintain short critical paths and relatively high clock frequency. Use of additional "bypass channels" allows to compensate deficiencies of current DP RAM implementations allowing to achieve high throughput of 1 record per 2 clock pulses. The mean latency (expressed in sorting cycles, each requiring 2 clock pulses) is equal to the capacity of the sorter increased by one, and therefore equal to the maximum allowable distance between unordered records in the input stream.

The source code of the described implementation will be available at `http://www.ise.pw.edu.pl/~wzab/fpga_heapsort`.

REFERENCES

[1] Williams, J., "Algorithm 232 - heapsort," *Communications of the ACM 7* **6**, 347–348 (1964).

[2] Machida, H., “US patent no 5603023: Processor circuit for heapsorting,” (February 1997).
[3] “Heap sort parallel.” http://www.doulos.com/knowhow/vhdl_designers_guide/models/heap_sort_parallel/.
[4] “System Generator for DSP Reference Guide.” http://www.xilinx.com/support/documentation/ip_documentation/dp_block_mem.pdf.
[5] Ha-duong, T. and Moreira, S., “The heap-sort based atm cells spacer,” in [*ATM, 1999. ICATM ’99. 1999 2nd International Conference on*], 346 –350 (1999).
[6] “GHDL - where VHDL meets gcc.” http://ghdl.free.fr.
[7] “Python programming language – official website.” http://www.python.org.
[8] “Virtex-6 FPGA family overview.” http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.