Annals of Emerging Technologies in Computing (AETiC)
Vol. 3, No. 4, 2019

Research Article

Hardware Dynamic Memory Manager for Hard Real-Time Systems

Lukáš Kohútka*, Lukáš Nagy and Viera Stopjaková

Institute of electronics and photonics, Slovak university of technology in Bratislava, Slovakia
lukas.kohutka@stuba.sk; lukas.nagy@stuba.sk; viera.stopjakova@stuba.sk
*Correspondence: lukas.kohutka@stuba.sk

Received: 10th September 2019; Accepted: 24th September 2019; Published: 1st October 2019

Abstract: This paper presents novel hardware architecture of dynamic memory manager providing memory allocation and deallocation operations that are suitable for hard real-time and safety-critical systems due to very high determinism of these operations. The proposed memory manager implements Worst-Fit algorithm for selection of suitable free block of memory that can be used by the external environment, e.g. CPU. The deterministic timing of the memory allocation and deallocation operations is essential for hard real-time systems. The proposed memory manager performs these operations in nearly constant time thanks to the adoption of hardware-accelerated max queue, which is a data structure that continuously provides the largest free block of memory in two clock cycles regardless of actual number or constellation of existing free blocks of memory. In order to minimize the overhead caused by implementing the memory management in hardware, the max queue was optimized by developing a new sorting architecture, called Rocket-Queue. The Rocket-Queue architecture as well as the whole memory manager is described in this paper in detail. The memory manager and the Rocket-Queue architecture were verified using simplified version of UVM and applying billions of randomly generated instructions as testing inputs. The Rocket-Queue architecture was synthesized into Intel FPGA Cyclone V with 100 MHz clock frequency and the results show that it consumes from 17,06% to 38,67% less LUTs than the existing architecture, called Systolic Array. The memory manager implemented in a form of a coprocessor that provides four custom instructions was synthesized into 28nm TSMC HPM technology with 1 GHz clock frequency and 0.9V power supply. The ASIC synthesis results show that the Rocket-Queue based memory manager can occupy up to 24,59% smaller chip area than the Systolic Array based manager. In terms of total power consumption, the Rocket-Queue based memory manager consumes from 15,16% to 42,95% less power.

Keywords: ***Hard Real-Time; Dynamic Memory Management; SRAM; ASIC; Worst-Fit***

---

1. Introduction

Hard real-time (RT) systems belong to a category of cyber-physical and embedded systems that contain real-time tasks. Success of hard real-time tasks depends not only on the task results but on

Lukáš Kohútka, Lukáš Nagy and Viera Stopjaková, "Hardware-Accelerated Dynamic Memory Manager for Hard Real-Time Systems", *Annals of Emerging Technologies in Computing (AETiC)*, Print ISSN: 2516-0281, Online ISSN: 2516-029X, pp. 48-70, Vol. 3, No. 4, 1st October 2019, Published by International Association of Educators and Researchers (IAER), DOI: 10.33166/AETiC.2019.04.005, Available: http://aetic.theiaer.org/archive/v3/v3n4/p5.html.

time when these tasks are finished too. Improper timing of hard real-time tasks completion (e.g. too late completion) generally represents a big failure as if the task results were incorrect, which may even cause a complete failure of the whole system. Therefore, it is critically important to consider determinism and predictability of hard RT systems when implementing algorithms for such systems. Ideally, all operations should be performed in constant or nearly constant time [1, 2].

Dynamic memory management algorithms are responsible for dynamic memory allocation and deallocation, which is intensively used in software development based on object-oriented programming. However, dynamic memory is usually not used in hard real-time systems at all. The reason for not using dynamic memory in hard real-time systems is the nondeterministic behavior of memory allocation and deallocation algorithms. The nondeterminism is caused by two facts. The first one is that a request for memory allocation can be rejected due to insufficient amount of free memory or due to memory fragmentation. The second source of nondeterminism is the variable response time of memory allocation/deallocation operations. For real-time systems, it is very important to perform such operations in constant time. Constant response time is especially important for hard real-time systems because they belong to safety-critical systems too. Even if a microprocessor with the highest performance was used, there is still no guarantee that the memory management operations meet the real-time requirements of the system. Therefore, a dedicated hardware-accelerated memory manager that provides implementation of Worst-Fit algorithm for memory allocation and deallocation of SRAM-based memory is proposed, which is suitable for usage in hard real-time systems due to its constant response time [3-7].

The constant latency of all operations within the system, including the Worst-Fit algorithm, is very important for more reliable and deterministic memory management in hard real-time systems. In such cases, software implementations do not meet these requirements because software implemented Worst-Fit algorithm does not operate in constant time [1-13].

Since the Worst-Fit algorithm contains data sorting operations, the research presented in this paper also includes a design of novel hardware-accelerated data sorting in a form of a min/max queue, which consumes less logic resources than existing min/max queues for its implementation. This was achieved by developing of a new sorting architecture, which was adopted for implementation of Earliest Deadline First (EDF) and Robust Earliest Deadline (RED) algorithms that are well-known as task scheduling algorithms suitable for real-time systems [8-32]. The new sorting architecture is called Rocket-Queue, which was developed to increase the scalability of data sorting realized by hardware in order to be able to sort more items for the same or similar resource cost [33-35].

The structure of this paper is following. Section 2 contains related work on dynamic memory management and worst-fit algorithm. Section 3 contains related work on sorting architectures that can be used for implementation of min/max queues with constant response time. In Section 4, a novel hardware-accelerated memory manager is proposed. In Section 5, new sorting architecture suitable for the proposed memory manager, called Rocket-Queue, is presented. Verification of the described memory manager is described in Section 6. Section 7 contains synthesis results including tables and graphs. These results are afterwards discussed in Section 8. The Section 9 sums up the whole research

presented by this paper. The last section describes the limitations of the proposed solution and future work.

2. Related Work on Dynamic Memory Management

From software point of view, there are two types of memory available to programmers for creation of program variables:

- **Static memory** – size and amount of program variables must be defined before the program is compiled and executed.
- **Dynamic memory** – size and amount of program variables can be specified and changed anytime during the execution of the program (i.e. run-time).

Although the dynamic memory is much more flexible, can waste less unused memory and is required for instantiation of classes in object-oriented programming, the management of dynamic memory is much more complicated when comparing to the static memory [36]. The dynamic memory management is responsible for memory allocation and deallocation, whenever it is requested from the computer program. From C language, which is relatively popular for implementation of embedded and real-time systems, the memory allocation is analogous to the "malloc" function and the memory deallocation is known as the "free" function.

For memory allocation, there exist several algorithms, such as:

- **First-Fit** – the algorithm searches for a sufficiently large empty block of memory and selects the first block that meets such requirement. The searching always begins from memory address 0 and continues forward until the first suitable block is found and selected for memory allocation.
- **Next-Fit** – very similar to the First-Fit algorithm. The only difference is that the searching does not begin from address 0, but the address of the last selected block is used as a starting point for the next searching.
- **Best-Fit** – the algorithm selects for allocation the smallest empty block among those blocks that are large enough to meet the memory size requirement. In order to selects such a block, the algorithm must filter out insufficiently large blocks and after that, the smallest block is selected, which requires sorting of these blocks afterwards.
- **Worst-Fit** – the largest empty block of memory is always selected for memory allocation. Even though this may appear to be a bad approach, whenever the selected block of memory is larger than the requested size, this block is split into two blocks: a block with requested size that is used for memory allocation and a block with remaining size that remains unused (i.e. free block). This algorithm does not require to filter out free blocks of memory with insufficient size before the blocks are sorted according to their size. Thus, the algorithm begins with sorting of all free blocks and continues with check, whether the selected largest block is sufficiently large. In terms of external fragmentation, the worst-fit and best-fit algorithms provide similar results. While best fit is statistically a bit more efficient in the case of allocating a mix of very small and very big blocks of memory, the worst fit algorithm is statistically a bit more efficient in the case of allocating very similar sizes of memory blocks
- **Buddy algorithm** – a completely different approach for allocating memory blocks dynamically is used in buddy algorithm. This algorithm restricts to provide only blocks of size that

is any power of number two (e.g. 2, 4, 8, 16, 32 and 64 bytes). The memory is being recursively divided in two halves. The dividing is being performed until the smallest possible memory block with still sufficient size is found. Two neighbouring blocks of free memory are combined together only if they have the same size. The advantage of buddy algorithm is that it is faster than the previous algorithms. The disadvantage is that the memory utilization is lower due to higher internal fragmentation. Even though this algorithm is faster, it is still not suitable for real-time systems because its execution time cannot be always precisely predicted.

3. Related Work on Sorting Architectures

Unlike other memory management algorithms, the Worst-Fit algorithm was identified as a suitable candidate for hardware acceleration because the major source of nondeterminism within its software implementation is the data sorting that is required to keep track of the largest block among all free blocks of memory. Therefore, an important part of the selected memory management algorithm is the sorting logic that provides the largest free block in constant time. Since only the largest free block is important, the sorting architecture can be implemented in a form of a max queue. The max queue is a data structure that provides the item with maximum value and it is possible to insert a new item or remove an existing item from the queue. The item removal must be available for any item according to its identification number (ID), not only for the max item. This is required in order to be able to merge neighbouring free blocks into one larger free block.

Several architectures for implementation of min/max queues were already developed and can be used for dynamic memory management in hard real-time systems. Nevertheless, they suffer from scalability issues due to increasing critical path length and resource cost with regards to increasing capacity of the queues (i.e. the maximum number of items that can be sorted). The most popular architectures include FIFO with MUX Tree [10, 11, 15, 21], Shift Registers [18, 20, 22, 23], DP RAM Heapsort [19] and Systolic Array [24-28, 33].

The FIFO approach is the least scalable in terms of critical path length due to the complexity of the MUX Tree part, which contains too long critical path. It is also very inefficient in terms of chip area cost [10, 11, 15, 21].

The Shift Register architecture is more efficient approach than the previous one, but the critical path length is still not constant with respect to the increasing queue capacity. The architecture is composed of homogenous cells, where each cell is able to store one item for sorting. These cells communicate with the nearest neighbours. All cells are receiving input data simultaneously from one common bus. The more cells the queue contains, the longer the critical path is present [18, 20, 22, 23].

DP RAM Heapsort is relatively efficient sorting architecture. However, it is not designed for min/max queues and the items in such architecture can be deleted only from the beginning of the sorting architecture. Removing items according to their ID is impossible [19].

In Systolic Array architecture, the critical path length problem is solved with pipelining approach. The first cell is the only one that receives the input data. When a new instruction is inserted into the first cell, the instruction is gradually propagated from one cell to another one, with a speed of one cell per one clock cycle. This happens until the instruction reaches the last cell. The only

disadvantage of the Systolic Array architecture is that it still consumes quite high amount of logic resources [24-28, 33].

From among the architectures mentioned above, only Systolic Array architecture fulfils the requirements of constant and relatively low response time, which is two clock cycles. Furthermore, the Systolic Array architecture can remove any item according to its ID, which is an important requirement of the worst-fit based memory manager.

4. Proposed Hardware-Accelerated Memory Manager

The proposed memory manager is implementing the existing algorithm, worst-fit, in a form of a coprocessor unit. The coprocessor provides four custom instructions that can be called from external environment. These instructions are:

- **MALLOC** – instruction that finds and allocates suitable free block of memory and provides an address of the allocated block as the instruction output. Input data consists of required memory size only.
- **FREE** – instruction that deallocates an allocated block of memory and merges adjacent free blocks of memory to one bigger free block of memory. The input for the instruction is only address of the block that has to be deallocated.
- **WRITE** – standard memory write instruction. This instruction is provided to eliminate possible memory access conflicts.
- **READ** – standard memory read instruction. The read value is returned one clock cycle later. Again, this instruction is provided to eliminate possible memory access conflicts.

Since the coprocessor does not provide any NOP (no operation) instruction, it can be simply enabled/disabled by a 1-bit *enable* signal. Thus, the selected instruction among the four possible instruction opcodes is valid only when the *enable* signal is activated.

The whole coprocessor operates in a single clock domain and therefore, it uses only one clock and one synchronous reset signal, which is used for putting the coprocessor into initial state.

The top-level block diagram of worst-fit based dynamic memory manager is depicted in Figure 1. The input ports of the manager consist of clock signal *clk*, reset signal *rst*, *enable* signal, signal for selection of instruction that is called *instr* and has bit width of 2 bits, and *data_in* signal representing input data needed for the selected instruction. The output ports are: *ready* signal that informs whether the environment that the manager is ready to accept a new instruction, *data_out* signal that contains output data produced by the executed instruction, signal *valid* that is used for informing the environment when the *data_out* contains valid data, and *error* signal that notifies the environment about possible errors that may occur. The top-level module of the memory manager consists of three components: *Control_unit*, *Max_queue* and *Memory*.

The Control_unit is the main component responsible for instruction decoding and instruction execution. This component processes all inputs and produces all outputs of the manager. It is also responsible for control of the remaining two components and therefore, there is a communication interface between Control_unit and the other two components.

The Memory component is the memory, which has to be managed by the memory manager. In our case, the memory is implemented by SRAM because of relatively simple interface of such memory type but the proposed solution is in principle applicable to any other memory types as well.

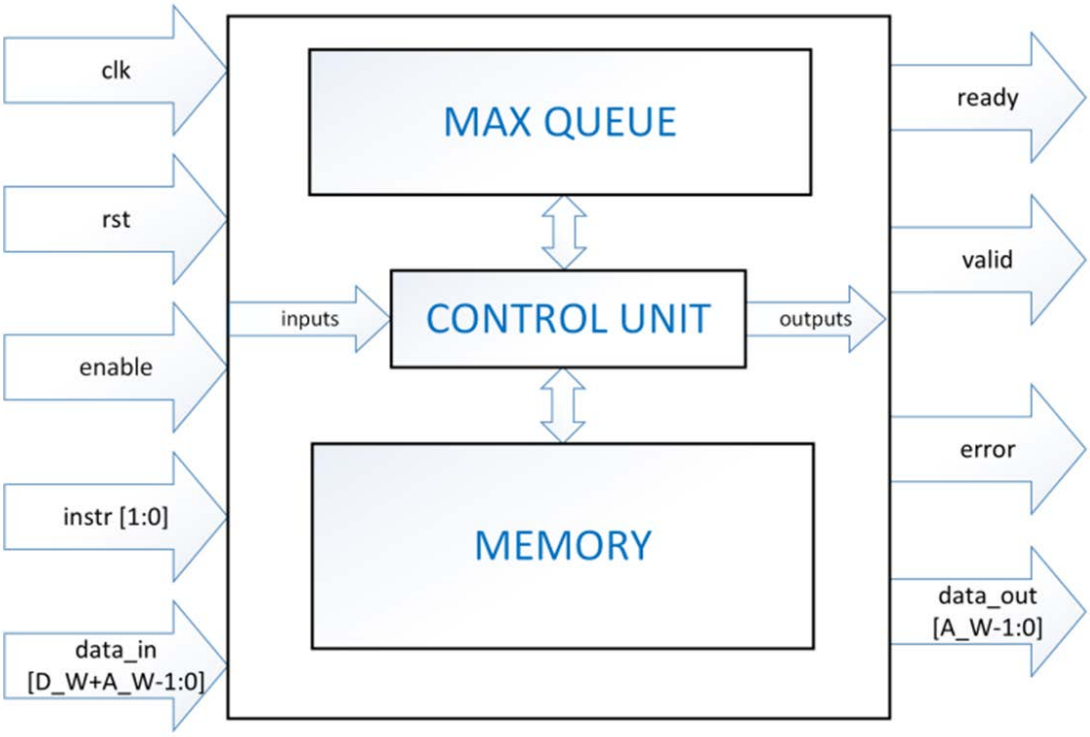

Figure 1. Block Diagram of Worst-Fit Memory Manager

The Max_queue component represents a data structure with own memory storage that is used for storing of information about all existing free blocks of memory that are available in the Memory component. The Max_queue is responsible for automatic sorting of the free blocks so that the largest free block (i.e. free block with maximum size) is continuously provided as an output of the Max_queue component. Due to this, the Control_unit component is always aware of which free block of memory is the largest one and where this block is located (i.e. what address belongs to this block).

Thanks to the Max_queue component, the Control_unit is relatively simple to implement because it does not have to deal with finding the largest free block of memory. The Control_unit is implemented in a form of a finite-state machine (FSM) that consists of 15 states in total. This state machine is described by a state diagram that is shown in Figure 2. Each state is named with a prefix "S_", which implies that the identifier is used for naming a state. The initial state that is provided by activating reset signal is called S_RESET. This state is used only for initialization of Memory with a header at address 0, which marks that the whole memory consists of only one big free block of memory that occupies the whole memory address space. The S_RESET state is always followed by S_READY state, which means that the memory manager is ready for accepting and executing a new instruction. Thus, the output *ready* is active. The Control_unit remains in S_READY state until a new instruction is provided, which is observed by activating the input *enable*. Depending on which instruction is selected, the next state is decided. If instruction WRITE is used, the state machine remains in state S_READY because the WRITE instruction is executed in 1 clock cycle only. If instruction READ is used, the Control_unit performs memory read operation and the state S_READY is followed by state S_MEM_READ. This state is used for returning the value read from memory and

providing it to the output *data_out*. The S_MEM_READ is always followed by S_READY state. If instruction MALLOC is used, the state machine goes to state S_MALLOC and the Control_unit reads output from the Max_queue, i.e. the size and address of the largest free block of memory. The size of the largest free block is compared to the requested size. If these sizes are equal, then the selected free block of memory is allocated as it is and the following state is S_READY. The Control_unit performs also removing of the corresponding item from the Max_queue and updating header of the selected memory block so that the block is flagged as allocated. However, the requested size of memory is usually lower than the available size of the largest free block. In such cases, the S_MALLOC state is followed by S_MALLOC_BIGER because the selected free block of memory is split to two smaller blocks. One block has exactly the requested size and is allocated, while the second block keeps the remaining size and remains free. This state is followed by S_WAIT state, which does not perform any operations and is used only to wait for one clock cycle before the state machine returns to the S_READY state. The highest number of states is used for execution of the FREE instruction. If the FREE instruction is used, the Control_unit starts reading the header of the block identified by address provided from input *data_in* and the S_READY state is followed by S_FREE_BEGIN. In state S_FREE_BEGIN, the header provided from Memory is analysed. If the header says that the block is already free, then the state machine returns to S_READY state and output *error* is activated. The Control_unit also checks whether the previous block (i.e. the block that is located before the block to be freed) is empty or allocated. If the previous block is empty, then the state S_FREE_BEGIN is followed by state S_FREE_PREV_EMPTY, otherwise the state S_FREE_PREV_USED is the next state. The Control_unit also performs reading of header of the next block located behind the block to be freed, which is then analysed in the next state. State S_FREE_PREV_USED checks whether the next block behind the selected block is free or allocated. If it is allocated, then the selected block is occupied by allocated block from both sides and thus this block is freed without any merging operations. In such case, the S_FREE_PREV_USED state is followed by S_WAIT state. The actual freeing is performed by flagging the selected block as free by rewriting its header in Memory and by inserting the address and size of the freed block to the Max_queue, which automatically keeps the largest free block as the output of the queue. However, if the next block behind the selected block for freeing is empty, then these two blocks are merged in the next state, called S_FREE_POSTFIX_MERGE. This merging is performed by removing the free block from Max_queue, which is achieved by identifying the free block according to its address. The S_FREE_POSTFIX_MERGE state is followed by S_FREE_END state, which performs insertion of the merged block to the Max_queue.

For cases when the previous block is free, i.e. when the state machine gets to state S_FREE_PREV_FREE, there must be again performed the analysis of the next block behind the block selected for deallocation. The Control_unit already performs removing of the previous block from the Max_queue too. If the next block is allocated, then the state S_FREE_PREV_FREE moves to state S_FREE_PREFIX_MERGE, which is used for merging of the selected block with the previous free block into one bigger free block of memory. If the next block behind the selected block is also free, i.e. the selected block is wrapped by free blocks from both sides in the memory, then all three blocks are merged together into one bigger free block. This is performed by states S_FREE_MERGES_1, S_FREE_MERGES_2 and S_FREE_MERGES_3. Such merging realizes sequential removal of the previous free block and the next free block from the Max_queue and insertion of the merged free block back to the Max_queue during the S_FREE_END state.

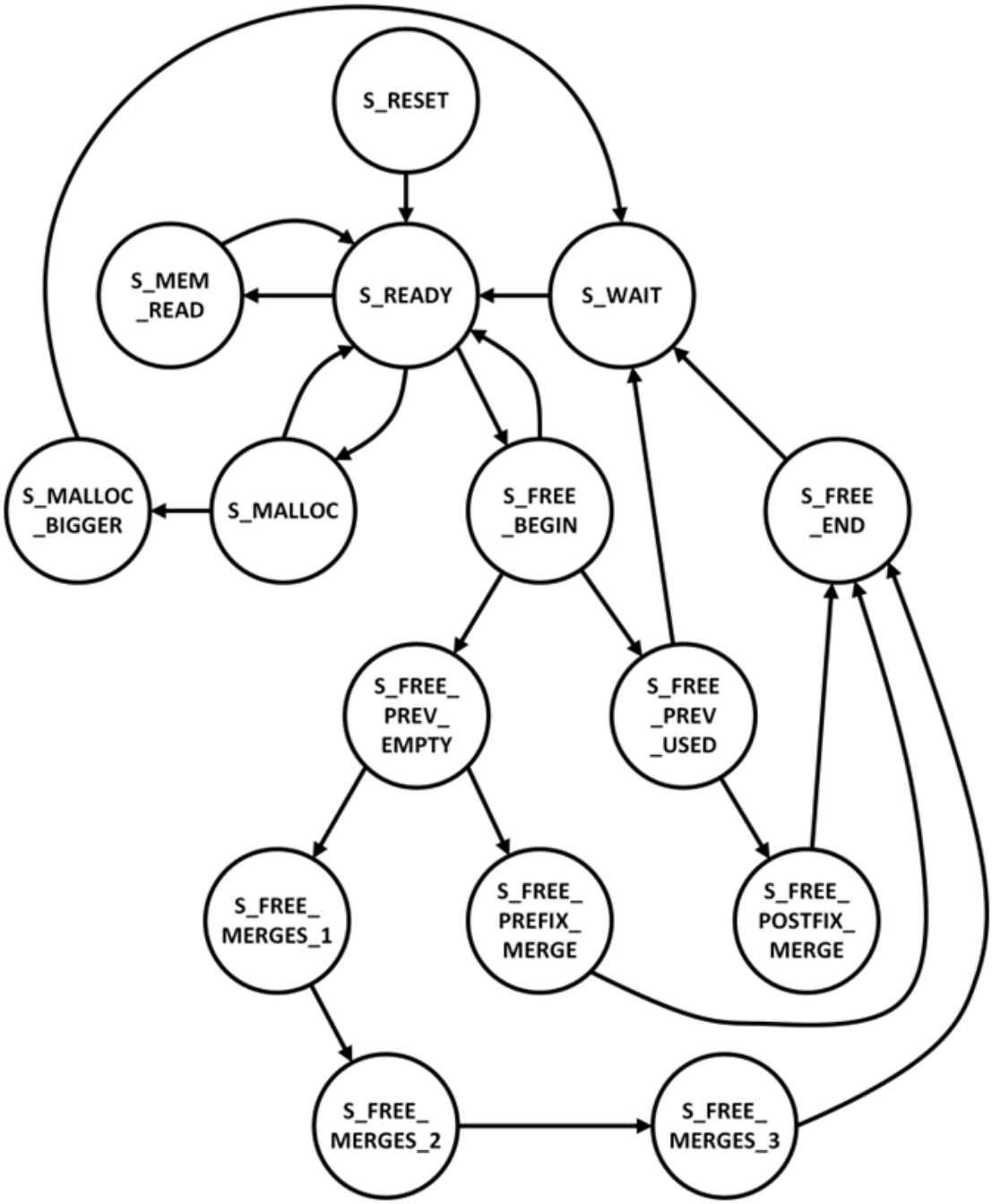

Figure 2. State Diagram of FSM Implementation of Control_unit

In terms of timing, the instructions take the following number of clock cycles depending on the decisions made by the state machine:

- **MALLOC**
  - 1 clock cycle – if no splitting is needed
  - 2 clock cycles – if block splitting is performed
- **FREE**
  - 4 clock cycles – if no merging is needed
  - 6 clock cycles – if merging with one neighbouring block is performed (i.e. either the previous or the next block only)

  - 8 clock cycles – if merging with both, i.e. the previous and the next block, are performed
- **WRITE** – 1 clock cycle
- **READ** – 2 clock cycles

5. Proposed Rocket-Queue Architecture

Even though the Systolic Array architecture can be used for implementation of the Max_queue, the sorting architecture represents a significant part of the whole coprocessor in terms of resource cost and power consumption. Systolic Array architecture is not very scalable with regards to the number of queue capacity (i.e. maximum number of items that can be stored and sorted). Although the functionality and timing attributes of operations implemented in Systolic Array and Rocket-Queue are the same (i.e. item inserting including the sorting and item removing based on item ID) because both architectures perform these operations in 2 clock cycles regardless of the queue capacity, these architectures consuming too many resources when higher queue capacity is used [34, 35].

One of the most intensively resource consuming parts of min/max queue architectures is a comparator that is used for data sorting. Such comparison is done in each cell of Systolic Array and in Shift Registers too. Reducing the number of comparators that are used in the queue can significantly reduce resource consumption. For this reason, a new architecture for min/max queues that is called Rocket-Queue was designed [34, 35].

The Rocket-Queue architecture does not use one comparator in each cell unlike the other existing min/max queue architectures. This is achieved by sharing the same comparator within a set of cells that are organized into one level. Since one comparator is shared by multiple cells, multiplexers are needed to access the shared comparator. The architecture consists of several levels and each level is composed of several cells that share the same comparator. There are two types of levels: duplicating levels and merged levels. The beginning of the queue is formed by duplicating levels that are called duplicating because each level below a duplicating level is composed of doubled number of cells. The duplication of cells per level is performed only for the first few levels, i.e. the number of duplicating levels is finite and parameterized. The levels below the last duplicating level are the merged levels. Each merged level keeps the same amount of cells [34, 35]. Figure 3 shows an example of Rocket-Queue, which consists of three duplicating levels and eleven merged levels. Each cell represents memory storage for one item to be sorted and the connections between these nodes represent possible movements of items between these nodes, which may occur whenever an item is being added or removed.

There are two drawbacks caused by merging comparators into one shared comparator within a level. The first disadvantage is that the multiplexers used for accessing the shared comparator have to be introduced, which increases the resource cost and critical path length. Therefore, the amount of duplicating levels is limited to five duplicating levels at most. The second disadvantage is the fact that the Rocket-Queue architecture must insert a counter into each cell to make decision, in which direction to further continue with the instructions that are inserting a new item into the min/max queue. Each instruction starts at the first level, which is located at the top of the min/max queue and then the instruction moves down, one level per one clock cycle. Only one cell within the same level

is actively used by the item insertion. The remove instructions check identification numbers (i.e. ID) of all cells within the same level simultaneously [34, 35].

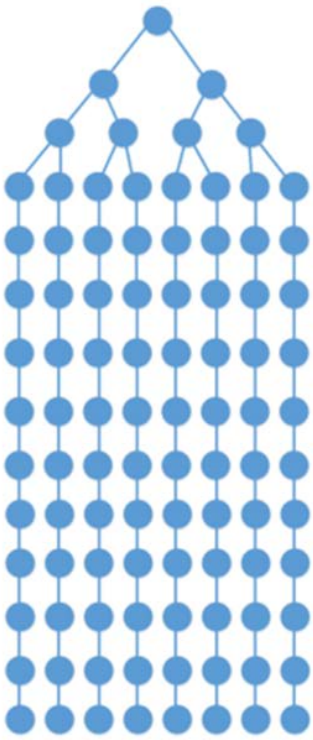

Figure 3. Rocket-Queue Architecture [34]

The organization of the Rocket-Queue architecture into levels with interfaces between these levels is described by Figure 4. The example depicted in this figure contains one duplicating level and two merged levels. It is noticeable that the interface of all levels, regardless of if it is a duplicating level or a merged level, is the same with exception of the first level, which provides the capacity of the one item only. Therefore, the first level does not need the ADDR input. The ADDR input is used to select a cell within the particular level. The first level is directly connected to the interface of Rocket-Queue too.

The ITEM_TOP is the input item that is provided for the queue, which has to be either added or removed. The ITEM_DOWN has the same purpose as the ITEM_TOP but it is used for the other levels below.

One can notice that all instructions are being propagated indirectly by the levels, thus, every level communicates with the nearest neighbours only.

The ADD_ITEM signals are one-bit control signals, which are used for distinction between item inserting and item removing. If ADD_ITEM is logic one, then the provided item (i.e. ITEM_TOP) has to be inserted into the queue. If it is logic zero, then the ID of the provided item is used for removing of an existing item from the min/max queue.

The PUSH signal is used to notify whether the provided instruction was already successfully executed in another level above the current level. Therefore, the first level has PUSH signal constantly driven to logic zero.

The ITEM_UP signals are used for providing items up to the higher levels. This is very especially important whenever an existing item is removed from the queue. The first level provides an item with the minimum/maximum sorting value as the output of the Rocket-Queue.

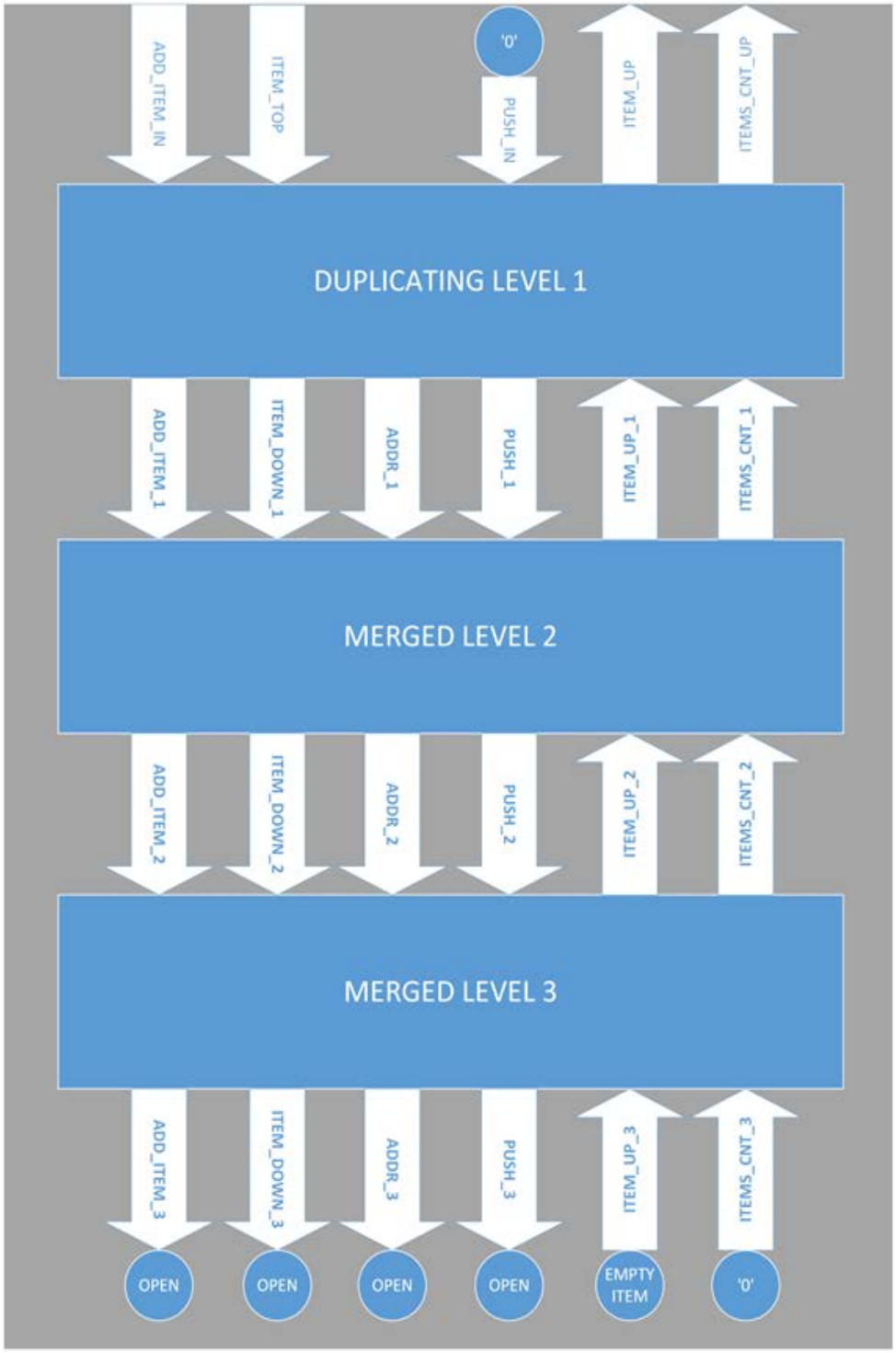

Figure 4. Block Diagram of Rocket-Queue

The min/max queue can provide the actual number of items that are present in the queue as well, which is handled by the ITEMS_CNT_UP. This is used for "tree balancing" too, i.e. to ensure that the new items are inserted into the highest possible level so that no data overflow occurs.

The lowest level (i.e. level 3 in Figure 4) represents also an end of the whole queue and thus, the outputs that would continue further below the queue are not used (i.e. they are opened). Since the

last level of the Rocket-Queue architecture has these outputs opened, data loss (i.e. data overflow) could occur if the tree balancing based on counting of the items was not used. Also, if number of items inserted into the queue is higher than the real capacity of the queue, then some items would be lost through the opened outputs of the lowest level, however, this can is caused only if the queue is either improperly used or incorrectly configured. An empty item is represented by item ID equal to binary zero and either the sorting value of binary zero for max queue configuration or the maximum possible binary value for min queue configuration. The empty items are used as an input for the Rocket-Queue, whenever no change has to be applied to the queue (i.e. NOP pseudo instruction).
Since the Rocket-Queue is parametrizable, it is very simple to change the capacity of the Rocket-Queue architecture. The only thing that must be specified is the number of duplicating levels and the number of merged levels. Both types of levels are using the same interface and they can be simply cascaded. The critical path length of the whole Rocket-Queue architecture stays constant regardless of how many merged levels are used. Only the number of duplicating levels affects the critical path length due to the increasing depth of multiplexer trees.

The pseudo-code below describes the algorithmic behaviour of the top-level of the Rocket-Queue architecture, whenever a new instruction to the min/max queue is sent. Although the pseudo-code is written as sequential, all the steps in the while loops are executed in parallel and iterations of these while loops use pipelining approach.

```
function run_rocket_queue_instruction()
{
  int i = 1;
  while (i <= DUPLICATING_LEVELS)
  {
    fill_inputs_of_level(i);
    run_duplicating_level(i);
    update_output_of_level(i);
    i++;
  }
  while (i <= DUPLICATING_LEVELS + MERGED_LEVELS)
  {
    fill_inputs_of_level(i);
    run_merged_level(i);
    update_output_of_level(i);
    i++;
  }
}
```

Figure 5 depicts a block diagram that describes the internal organization of one duplicating level. At first, there is several registers used to store items. However, these items share the common comparator to compare item values, which is achieved by using of the multiplexer tree to select, which two item values are compared. These two values are called A and B for simplicity. In addition to this, every item register uses own "ARE IDs EQUAL?" submodule, which serves to compare the

item IDs to the ID of the INPUT_TOP item. This way, the item removal instruction executed for all items within the level simultaneously.

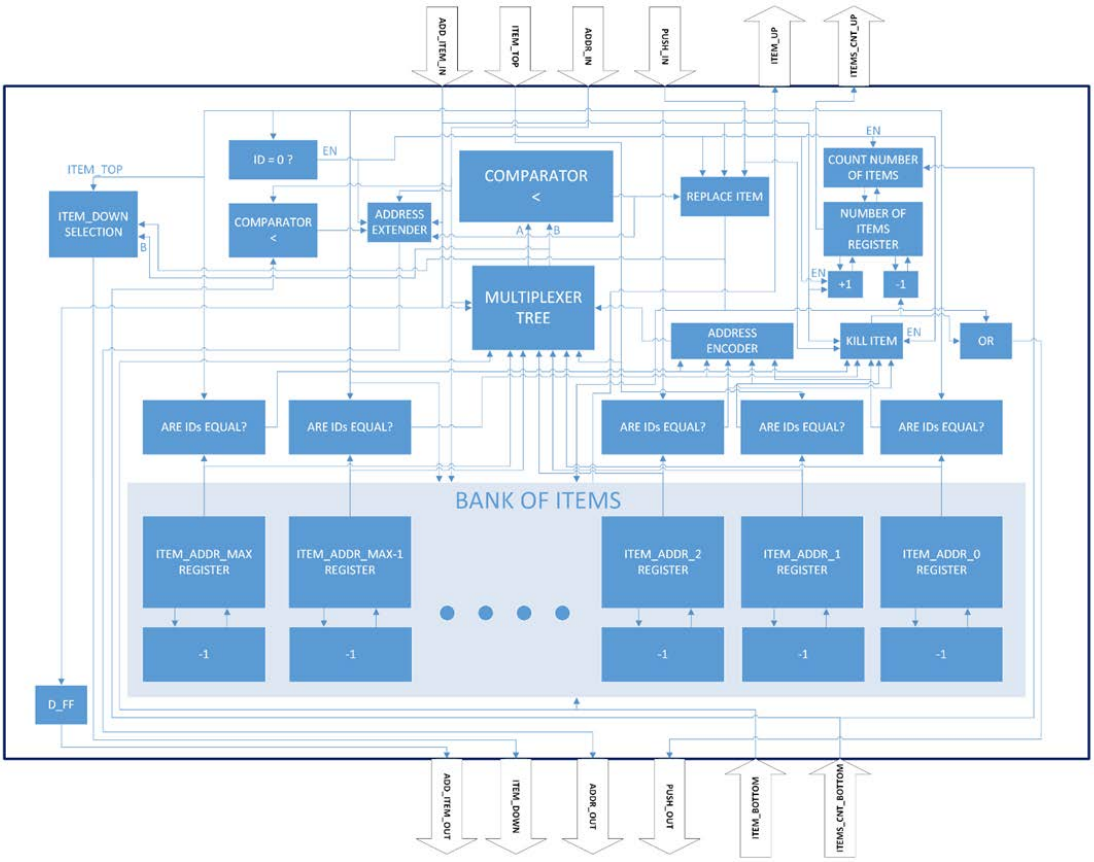

Figure 5. Block Diagram of One Duplicating Level of the Rocket-Queue Architecture

The REPLACE ITEM represents a control logic that performs decisions, whether the INPUT_TOP item has to replace an existing item that is localized by the address ADDR_IN. Replacement occurs if and only if the instruction is the inserting instruction (i.e. ADD_ITEM_IN is logic one), the input ITEM_TOP is not an empty item and either the PUSH_IN is logic one or the ITEM_TOP contains better sorting value (i.e. it is lower for min queue and higher for max queue) than the existing item stored in BANK OF ITEMS that is identified by the ADDR_IN.

For the killing instruction, ARE IDs EQUAL? modules perform the comparisons of the IDs with the input ID. Then results of these comparisons are sent to the ADDRESS ENCODER, which sets up an address of the item that has the same ID as the input ID. The address is further sent to the MULTIPLEXER TREE to select the item that has to be removed by moving an existing item from the level below the current level to the selected item register. Thanks to the MULTIPLEXER TREE, it is the COMPARATOR can be shared for all the items within the same level. The COMPARATOR serves for comparison of sorting values from items. By sharing one comparator for multiple items, one can save logic resources and power consumption despite the fact that the multiplexers were added to implement the sharing feature.

The KILL ITEM module is a combinational logic circuit that performs the decision, when the moving of an existing item from below to the current level is executed.

The NUMBER OF ITEMS REGISTER keeps the number of items for each cell within the actual level so that this value is equal to the total number of items of a sub-tree that is represented by the

actual cell as the root node. The ADD_ITEM_IN signal is delayed by one clock cycle, which is then further propagated as the output ADD_ITEM_OUT.

Every instruction spends always one clock cycle per level. Thus, pipelining is applied. However, every instruction must be followed by one NOP pseudo-instruction. For this reason, the throughput of the Rocket-Queue is 2 clock cycles per instruction, which is actually the same throughput as for the Systolic Array architecture.

The ADDRESS EXTENDER block is used to extend the ADDR_IN by one new bit that decides the further continuation of the instructions down the levels. The output of the ADDRESS EXTENDER propagated to ADDR_OUT. Such address extension is implemented in duplicating levels only.

The following pseudo-code describes the behavioural functionality of one duplicating level. The first part of the pseudo-code is used for item insertion and the second part is represents the item removing.

```
function run_duplicating_level()
{
  if (ADD_ITEM_IN == 1)
  {
    if ((ITEM_TOP.value < my_items[ADDR_IN].value) or (PUSH_IN == 1))
    {
      ITEM_DOWN = ITEM_UP[ADDR_IN];
      ITEM_UP[ADDR_IN] = ITEM_TOP;
      PUSH_OUT = 1;
    }
    else
    {
      ITEM_DOWN = ITEM_TOP;
      PUSH_OUT = 0;
    }
    number_of_items[ADDR_IN]++;
    ADDR_OUT = {ADDR_IN, (NUMBER_OF_ITEMS_BOTTOM[{ADDR_IN, 1}] < NUMBER_OF_ITEMS_BOTTOM[{ADDR_IN, 0}])};
  }
  else
  {
    foreach item item with index i in ITEM_UP do
    {
      if ((ITEM_UP[i].ID == ITEM_TOP.ID) or (PUSH_IN == 1))
      {
        bit go_left = (ITEM_BOTTOM[{ADDR_IN, 1}].value < ITEM_BOTTOM[{ADDR_IN, 1}].value);
        number_of_items[ADDR_IN]--;
        PUSH_OUT = 1;
        if (PUSH_IN == 1)
        {
```

```
            ADDR_OUT = {ADDR_IN, go_left};
            ITEM_UP[ADDR_IN] = ITEM_BOTTOM[{ADDR_IN, go_left}];
          }
          else
          {
            ADDR_OUT = {i, go_left};
            ITEM_UP[i] = ITEM_BOTTOM[{i, go_left}];
          }
        }
      }
    }
}
```

The Figure 6 shows a block diagram of a merged level. One can notice that it is almost exactly the same as the block diagram of duplicating level depicted in Figure 5. The only difference is that the number of cells is not duplicated and therefore, the addresses are not extended anymore, which causes that the ADDRESS EXTENDER and the items count comparator are removed. The other parts of the merged level are exactly identical to the duplicating level.

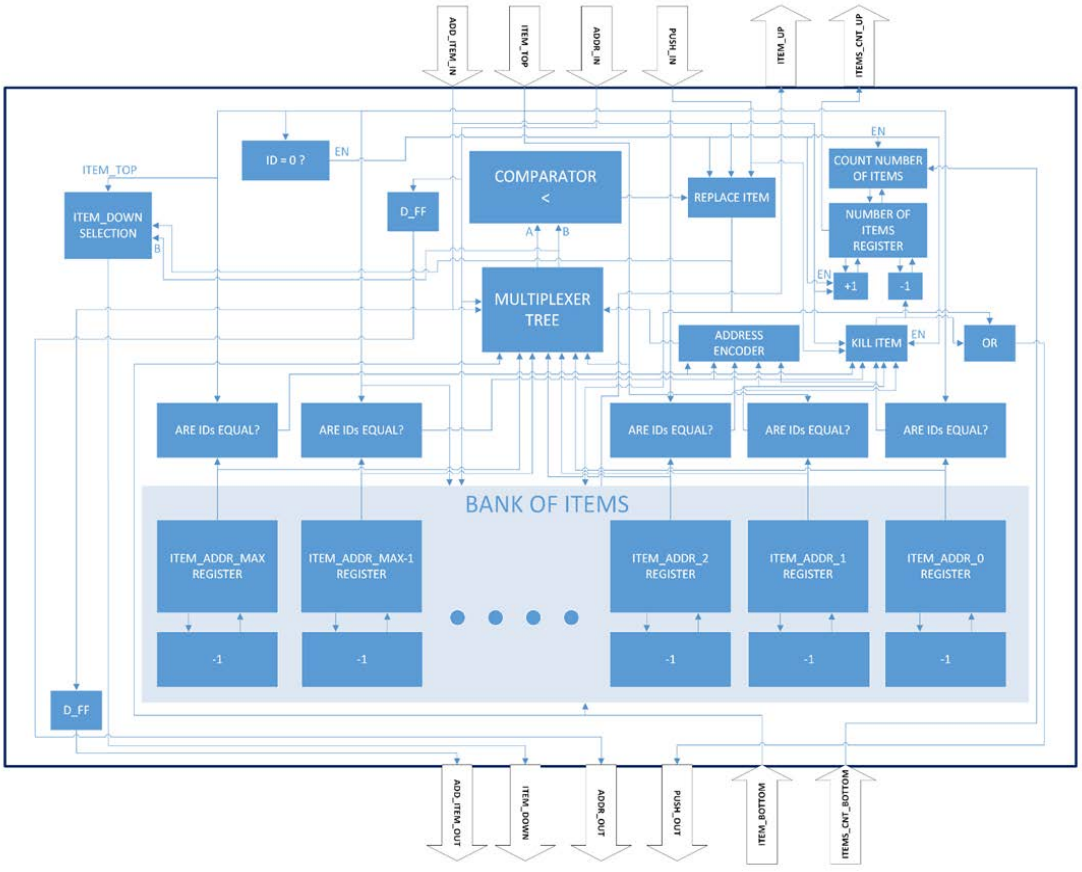

Figure 6. Block Diagram of One Merged Level of the Rocket-Queue Architecture

The following pseudo-code describes the behavioural functionality of one merged level. It is very similar to the pseudo-code of duplicating levels. Only addresses are not extended in this case anymore.

```
function run_merged_level()
{
```

```
if (ADD_ITEM_IN == 1)
{
  if ((ITEM_TOP.value < my_items[ADDR_IN].value) or (PUSH_IN == 1))
  {
    ITEM_DOWN = ITEM_UP[ADDR_IN];
    ITEM_UP[ADDR_IN] = ITEM_TOP;
    PUSH_OUT = 1;
  }
  else
  {
    ITEM_DOWN = ITEM_TOP;
    PUSH_OUT = 0;
  }
  number_of_items[ADDR_IN]++;
  ADDR_OUT = ADDR_IN;
}
else
{
  foreach item with index i in ITEM_UP do
  {
    if ((ITEM_UP[i].ID == ITEM_TOP.ID) or (PUSH_IN == 1))
    {
      number_of_items[ADDR_IN]--;
      PUSH_OUT = 1;
      if (PUSH_IN == 1) {
        ADDR_OUT = ADDR_IN;
        ITEM_UP[ADDR_IN] = ITEM_BOTTOM[ADDR_IN];
    }
      else
      {
        ADDR_OUT = i;
        ITEM_UP[i] = ITEM_BOTTOM[i];
      }
    }
  }
}
```

6. Verification of Proposed Solution

Two versions of memory manager in a form of coprocessor were developed:

- Memory manager that uses the existing Systolic Array architecture for implementation of the Max_queue component.

- Memory manager that uses the novel Rocket-Queue architecture for implementation of the Max_queue component.

Both these versions were verified as well as the Rocket-Queue architecture alone. All modules were described in SystemVerilog language and verified by simulations in ModelSim.

In addition to SystemVerilog language a simplified version of Universal Verification Methodology [37], also known as UVM, was applied during the verification phase in order to increase the strength of the verification and minimize the chance that the designed modules could not working as expected. Since the interface of the coprocessor is simple, the UVM was able to be simplified too. In our case, one transaction of standard UVM is implemented as a single instruction that is performed in two clock cycles. Thus, it is not needed to implement UVM agents to interface the design under test (DUT). Only one test procedure that generates constrained random instructions, one predictor and one scoreboard were used. The test procedure generates millions of random instructions that contain predefined instruction opcode but randomized instruction data. The predictor is a verification module that is responsible for prediction of DUT outputs according to the inputs provided from the test procedure. The description of the predictor is purely sequential and high-level, similarly to software. For example, the predictor is using standard SystemVerilog queue structure and corresponding sort() function that is used for software implementation of the Max_queue. A block diagram of the testbench architecture that is used for verification is depicted in Figure 7.

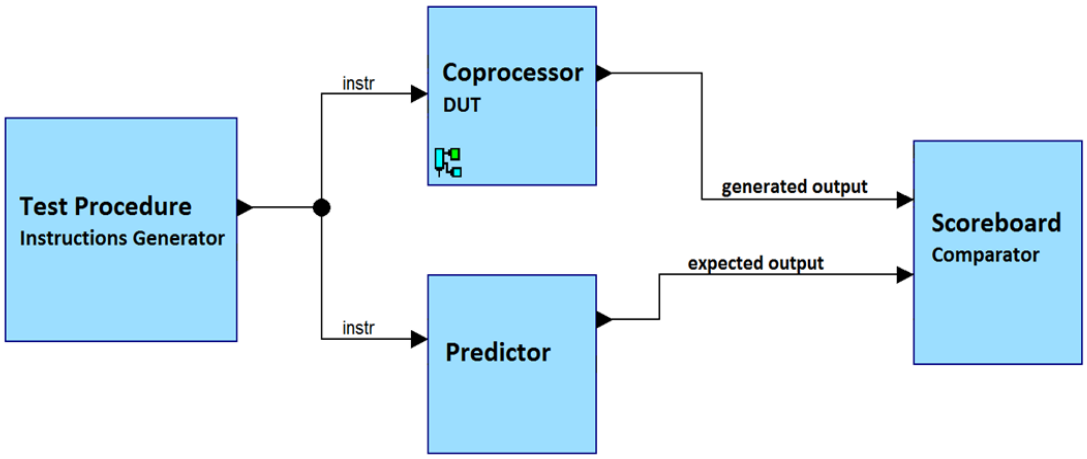

Figure 7. Block Diagram of Simplified UVM Testbench

More than a million of test iterations, where one such iteration consists of 1024 random instructions, were used to verify the designed modules. All instruction types were used during the testing. Full capacity of the Memory and Max_queue was used in these tests. Various configuration parameters were used for the coprocessor verification, i.e. the Memory depth (D_W) and word length (A_W) were changing.

7. Synthesis Results

An FPGA synthesis of two sorting architectures for min/max queues, Systolic Array ad Rocket-Queue architectures, was performed to compare the resource costs of both architectures in terms of LUT and registers consumptions. The target device for the synthesis is Intel FPGA Cyclone V

(5CSEBA6U23I7) and the target clock frequency is 100 MHz, which is relatively common clock frequency for current FPGAs. Two comparisons were performed: one for Adaptive Logic Module (ALM) consumption that represents the consumption of LUTs and one for registers consumption. These synthesis results are presented in Table 1. The bit width of the sorted data is 60 bits. The queue capacity, i.e. number of cells, is varying from 31 to 255 and the item ID width is always the lowest possible (e.g. 7 bits for 127 items and 8 bits for 255 items). The number of duplicating levels used in the Rocket-Queue architecture is 4. The number of merged levels depends on the total cells count (e.g. 7 merged level for 127 items or 15 merged levels for 255 items), where every merged level is composed of 16 cells.

Table 1. FPGA Synthesis Results of Systolic Array and Rocket-Queue architectures

Cells Count	*Item ID Width*	*Systolic Array ALMs*	*Systolic Array Regs*	*Rocket-Queue ALMs*	*Rocket-Queue Regs*
31	5	3.494	4.025	2.898	2.527
63	6	7.981	8.374	5.991	4.951
95	7	13.782	12.851	9.256	7.501
127	7	17.953	17.203	12.496	9.920
159	8	23.786	21.872	15.307	12.678
191	8	28.682	26.288	18.011	15.156
223	8	33389	30704	20850	17648
255	8	38992	35120	23915	20125

According to data in Table 2, Figure 8 shows the comparison of ALM and registers consumption between Systolic Array architecture and Rocket-Queue architecture. One can notice that the proposed Rocket-Queue architecture consumes significantly less resources than Systolic Array for the same queue capacity and when the queue is implemented in FPGA.

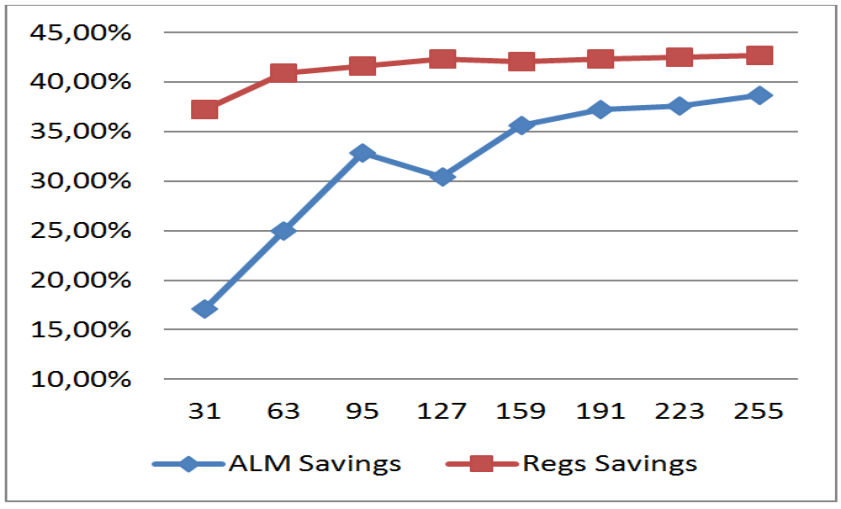

Figure 8. Relative FPGA Resource Cost Savings

Another synthesis was performed for the whole memory managers, one that uses Systolic Array architecture for implementation of the Max_queue and the second manager that is based on the Rocket-Queue architecture. These memory managers were synthesized for ASIC technology, specifically 28nm TSMC HPM. Target frequency used for the synthesis is 1 GHz and the voltage level used for powering the integrated circuit is 0.9 V. The chip area costs of the Memory itself, Systolic

Array based manager without the Memory, and the Rocket-Queue based manager without the Memory are presented in Table 2. The A_W parameter defines the bit width of memory addresses, which also indirectly specifies the number memory depth (i.e. number of memory words and possible addresses). The D_W parameter defines the bit width of one memory word. The chip area cost results are presented in μm².

Table 2. ASIC Chip Area Costs of Systolic Array based and Rocket-Queue based Memory Managers

A_W	*D_W*	*Memory*	*Systolic Array based manager*	*Rocket-Queue based manager*
4	16	1.033	725	625
5	16	2.091	1.260	1.217
6	16	4.196	2.499	2.566
7	16	8.415	5.077	5.226
8	32	33.473	10.959	10.611
9	32	67.009	23.314	17.582

According to data in Table 2, Figure 9 shows the comparison of Systolic Array based memory manager and Rocket-Queue based memory manager with respect to the chip area cost. The results are in μm².

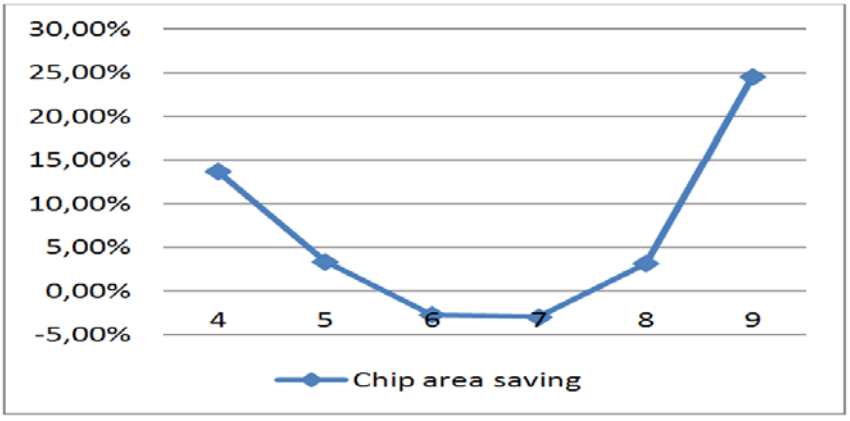

Figure 9. Relative Chip Area Cost Savings

The power consumption results are depicted in Table 3 and presented in μW. These results represent total power consumption that consists of the leakage power and dynamic power.

Table 3. ASIC Power Consumptions of Systolic Array based and Rocket-Queue based Memory Managers

A_W	*D_W*	*Memory*	*Systolic Array based manager*	*Rocket-Queue based manager*
4	16	157,34	152,95	129,76
5	16	319,28	313,50	260,58
6	16	640,66	660,65	528,92
7	16	1.283,16	1.451,75	1.202,46
8	32	5.083,49	3.463,39	2.578,34
9	32	10.173,68	7.552,76	4.308,83

According to data in Table 3, Figure 10 shows the comparison of Systolic Array based memory manager and Rocket-Queue based memory manager with respect to the total power consumption. The results are in μW.

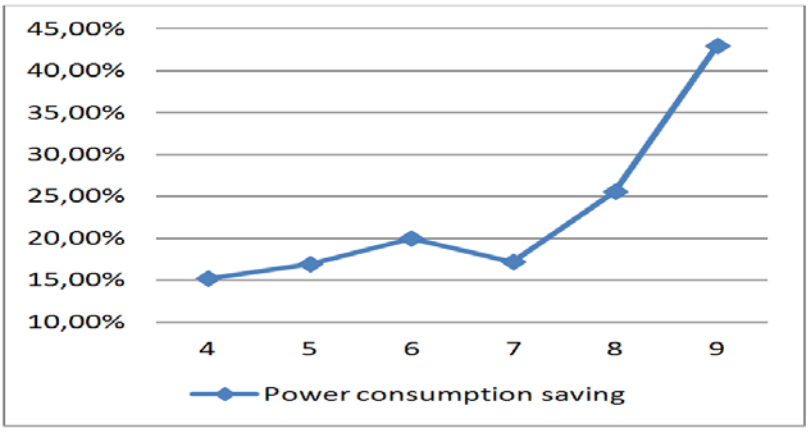

Figure 10. Relative Power Consumption Savings

8. Discussion

The experimental results show that the proposed Rocket-Queue-based dynamic memory manager implementing worst-fit algorithm is scalable regardless of whether it is implemented in FPGA or ASIC technology. The memory manager performs its memory allocation and memory free operations in few clock cycles regardless of memory size and regardless of the number of memory blocks that are present in the memory.

If we compare the proposed hardware implementation of worst-fit algorithm to the existing software implementations, it is clear that the performance and determinism are significantly improved if the hardware implementation is used. Software implementations require typically thousands of CPU clock cycles (or even more) for every memory allocation and every memory free operation. In addition to that, this timing can vary a lot due to memory fragmentation, which greatly reduces determinism of such a system.

9. Conclusion

Novel hardware architecture of the min/max queue, called Rocket-Queue, and hardware architecture of worst-fit based dynamic memory manager was presented in this paper. The proposed Rocket-Queue architecture is based on existing architectures - Shift Registers, Systolic Array and DP RAM Heapsort. The Rocket-Queue architecture provides more efficient sorting of items that can be used for implementation of min/max queues. Since max queue is needed for implementation of worst-fit algorithm, the worst-fit based memory manager also adopted the proposed Rocket-Queue architecture. The designed modules were described with SystemVerilog language and verified by UVM and simulations that contained random testing inputs. The presented modules were synthesized and tested on Intel FPGA Cyclone V and LUT consumptions were analyzed. In addition to this, a synthesis into 28 nm ASIC was performed too. The ASIC synthesis results were compared to evaluate chip area cost and power consumption of all presented modules. The comparison shows that the proposed Rocket-Queue architecture is significantly more efficient than the Systolic Array

architecture and therefore, the Rocket-Queue based memory manager is significantly more efficient than the Systolic Array based version too.

All the presented solutions are especially suitable for hard real-time systems because these systems are very sensitive to any sources of non-determinism. Dynamic memory management is usually not allowed to be used in hard real-time systems due to the unpredictable time needed for allocation and deallocation of memory. The proposed hardware-accelerated memory manager eliminates this problem thanks to the fact that the memory allocation and memory free operations take constant time with respect to the actual number and position of free blocks of memory. The memory allocation operation consumes either 1 or 2 clock cycles. The deallocation consumes either 4, 6 or 8 clock cycles, depending on how many merges are required.

One of the main benefits caused by using the proposed memory manager is that real-time systems could start using dynamic memory in the embedded software, which would lead to more dynamic programming with object-oriented style. Thus, the gap between real-time systems programming and ordinary programming could be reduced, resulting in shorter time to market (TTM) and lower development costs of real-time systems. Another possible benefit is that memory allocation that allocates blocks of memory for operating system tasks (i.e. processes and threads) can be accelerated, which would increase the overall system performance and determinism.

10. Limitations and Future Work

There are two limitations related to the proposed solution: hardware size and memory fragmentation.

Since the proposed solution is implemented in ASIC or FPGA, in both cases, there is always a limitation for the maximum acceptable size of such a hardware, i.e. LUTs in FPGA or chip area in ASIC. This limitation affects the maximum number of memory blocks that can be managed using the proposed solution, which further limits the maximum allowed size of the memory that can be managed. For example, if the memory manager can manage up to 1000 blocks of memory, the memory can be split into 2000 blocks of memory at most. If the minimum size of one block is 1 kB, then such a memory manager can be used for a 2 MB memory. In order to manage a bigger memory, either the minimum size of one block or the size of the memory manager must be increased.

The second limitation is related to the memory fragmentation, which comes from the characteristics of the worst-fit algorithm. This algorithm causes no internal fragmentation, but the external fragmentation can be an issue, depending on the actual application and its demands for the memory. If the memory is fragmented too much, there is a possibility that the request for memory allocation fails, which may be critical for many real-time systems. This issue can be solved by performing many simulations and tests to see, if this issue ever happens and by increasing the memory size if needed. Additionally, the system can have a plan B that could be based on static memory or could request a smaller block of dynamic memory. Another option how to solve memory fragmentation is to perform re-fragmentation of the memory periodically, if there is time for it in the system.

The future work will be focused on improving the proposed solution with respect to the limitations mentioned above. Furthermore, it is planned to combine the proposed memory manager

with task scheduling and ideally, to combine the proposed solution with an existing Linux-based operating system.

Acknowledgement

This work was supported in part by the Slovak Research and Development Agency under grant APVV-15-0254 and by the Slovak Republic under grant VEGA 1/0905/17.

References

[1] R. Mall, "Real-Time Systems: Theory and Practice," 2nd edition, 2008, ISBN 978-81-317-0069-3.

[2] C.A. O'Reilly, A.S. Cromarty, "Fast" is not "Real-time" in designing effective real-time AI systems," SPIE Vol. 5~8 Application of Artificial Intelligence II, pp. 249-257, 1985.

[3] J. A. Stankovic, K. Ramamritham, Tutorial hard real-time systems, Computer Society Press, 1988.

[4] G.C. Buttazzo, "Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications," 2011.

[5] S. Heath, Embedded Systems Design, Newnes, 2003, ISBN: 0750655461.

[6] Lee, I.; Leung, J. Y.-T. & Son, S. H., Handbook of Real-Time and Embedded Systems, Chapman & Hall/CRC, 2007.

[7] M. Joseph, „Real-time Systems Specification, Verification and Analysis," Prentice Hall International, London, 2001.

[8] P. Marwedel: Embedded System Design: Embedded Systems Foundations of Cyber-physical Systems, 2010, ISBN 9400702566.

[9] M. Pohronská, "Utilization of FPGAs in Real-Time and Embedded Systems," in M. Bielikova, ed.,'Proceedings in Informatics and Information Technologies Student Research Conference', Vydavateľstvo STU, 2009.

[10] C. Ferreira, and A.S.R. Oliveira, "Hardware Co-Processor for the OReK Real-Time Executive," 2010.

[11] C. Ferreira, A. S. R. Oliveira, "RTOS Hardware Coprocessor Implementation in VHDL," 2009.

[12] A. B. Lange, K. H. Andersen, U. P. Schultz, A. S. Sorensen: HartOS - a Hardware Implemented RTOS for Hard Real-time Applications, 2012, s. 207-213.

[13] S.E. Ong, and S.C. Lee, "SEOS: Hardware Implementation of Real-Time Operating System for Adaptability," Computing and Networking (CANDAR), 2013 First International Symposium, 2013.

[14] S. Liu, Y. Ding, G. Zhu, Y. Li: Hardware scheduler of Real-time Operating. In: Advanced Science and Technology Letters Vol.31, 2013, s. 159-160.

[15] G. Bloom, G. Parmer, B. Narahari, R. Simha: Real-Time Scheduling with Hardware Data Structures, 2010.

[16] M. Varela, R. Cayssials, E. Ferro, E. Boemo, "Real-time scheduling coprocessor for NIOS II processor", *Proc. VIII Southern Conf. Programmable Logic*, pp. 1-6, 2012.

[17] R. Chandra, O. Sinnen: Improving Application Performance with Hardware Data Structures, 2010, ISSN 11783680.

[18] S.W. Moon, "Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches," IEEE Transactions on Computers, 2000.

[19] W. M. Zabołotny, "Dual port memory based heapsort implementation for fpga," Proceedings of SPIE, 2011.

[20] Y. Tang, and N.W. Bergmann, "A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems," IEEE Transactions on Computers, 2015.

[21] J. Starner, J. Adomat, J. Furunas, and L. Lindh, "Real-Time Scheduling Co-Processor in Hardware for Single and Multiprocessor Systems," Proceedings of the EUROMICRO Conference, 1996.

[22] K. Kim, D. Kim, and Ch. Park, "Real-Time Scheduling in Heterogeneous Dual-core Architectures," Proceedings of the 12th International Conference on Parallel and Distributed Systems, 2006.

[23] L. Kohutka, "Hardware task scheduling in real-time systems" in IIT.SRC 2015, Student Research Conference, 2015.

[24] L. Kohutka, M. Vojtko, and T. Krajcovic, "Hardware Accelerated Scheduling in Real-Time Systems," Engineering of Computer Based Systems Eastern European Regional Conference, 2015.

[25] L. Kohutka, V. Stopjakova, "Hardware Accelerated Task Scheduling in Real-Time Systems", Adept, 2016.

[26] L. Kohutka, V. Stopjakova, "Hardware Accelerated Task Scheduling in Real-Time Systems: Deadline Based Coprocessor for Dual-Core CPUs", DDECS, 2016.

[27] L. Kohutka and V. Stopjakova, "Task scheduler for dual-core real-time systems," 23rd International Conference Mixed Design of Integrated Circuits and Systems, 2016.

[28] L. Kohutka and V. Stopjakova, "Improved Task Scheduler for Dual-Core Real-Time Systems," Euromicro Conference on Digital System Design (DSD), 2016.

[29] L. Kohutka and V. Stopjakova, "A Novel Hardware-Accelerated Real-Time Task Scheduler based on Robust Earliest Deadline Algorithm," 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS), 2018.

[30] L. Kohutka and V. Stopjakova, "A New Hardware-Accelerated Scheduler for Soft Real-Time Tasks," 8th Mediterranean Conference on Embedded Computing (MECO), 2019.

[31] K. Churnetski, „Real-time scheduling algorithms, task visualization," Computer Science Department Rochester Institute of Technology, 2006.

[32] A. Mohammadi, S. G. Akl, „Scheduling Algorithms for Real-Time Systems," School of Computing, Kingston, Ontario, 2005.

[33] F. Klass and U Weiser, "Efficient systolic arrays for matrix multiplication," in Proc. Int. Conf. Parallel Processing, Austin, Tex., Aug. 1991, vol. III, pp. 21-25.

[34] L. Kohutka and V. Stopjakova, "Rocket-Queue: New Data Sorting Architecture for Real-Time Systems," 20th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2017.

[35] L. Kohutka and V. Stopjakova, "A New Efficient Sorting Architecture for Real-Time Systems," 6th Mediterranean Conference on Embedded Computing (MECO), 2017.

[36] H.-K. Choi, Y.C. Chung, S.-M. Moon, "Java Memory Allocation with Lazy Worst Fits for Small Objects", The Computer J., vol. 48, no. 4, July 2005.

[37] IEEE Standard for Universal Verification Methodology Language Reference Manual, IEEE 1800.2-2017, 2017.