UDC 004.383

DESIGN OF DATA BUFFERS IN FIELD PROGRAMMABLR GATE ARRAYS

**A. M. Sergiyenko, P. A. Serhiienko, I. V. Mozghovyi,
A. A. Molchanova**

Annotation: The design of the data buffers for the field programable gate array (FPGA) projects is considered. A new method of buffer design is proposed, which is based on the representation of the synchronous dataflow graph in the three-dimensional space, optimization of them, and description in VHDL. The method gives the optimized buffers which are based either on RAM or on the register pipeline. The derived pipeline buffer can be mapped into the shift register primitive of FPGA. The method is built in the experimental SDFCAD framework intended for the pipelined datapath synthesis.

***Keywords**: FPGA, VHDL, synchronous dataflow, datapath synthesis.*

Introduction

Field programmable gate arrays (FPGAs) are popular devices that provide both high-speed computations for any complex task and availability for many designers of application-specific computers. The FPGA design technology was expanded over the last decades, which is based on the register transfer level (RTL) description of the computational datapath using the hardware description language like Verilog or VHDL. In recent years, high-level synthesis tools become popular because they provide a compilation of the C programs into the hardware descriptions, inviting the firmware programmers for designing the FPGA applications [1].

In many cases, the FPGA project consists of a set of ready blocks and intellectual property cores (IP cores) that communicate with each other through the proper interfaces and data buffers. But the selection of these interfaces and designing these buffers are still uncertain. In most cases, the usual methods of the RTL design are used or the ready IP cores are selected to build the data buffers, which gives the increased hardware volume, insufficient throughput, or both.

Most FPGA projects are pipelined, application-specific processors. The FPGA architecture contains a lot of hardware resources like registers, FIFOs, pipelined DSP blocks, two-port blocked RAMs (BRAM), and pipelined input-output pads, which support the pipelined computations. But they are utilized in the data buffers using the old synthesis methods which don`t provide good results. In particular, the buffers are usually designed separately from the pipelined datapath to which they are connected [2].

In the article, a new method of data buffer design is proposed which provides effective FPGA resource utilization. The derived buffer solution is described by the VHDL language and can be used effectively in any hardware project.

Methods for the buffer design

The methods of the data buffer design evolved for decades. The memory bandwidth increase is the usual goal of the buffer design. The easiest way to increase the memory bandwidth is to have multiple memory blocks in parallel. Similarly, it is possible to implement a memory with an extra-large data word length that stores several adjacent data. But in these cases, when the memory is out of FPGA, in addition to several memory chips, it is necessary to have many separate outputs from FPGA for addresses and data, which is often unacceptable. The impact of this problem is somewhat reduced by organizing several blocks of cache memory in FPGA. By dividing the address space into multiple banks, using one memory bank for odd addresses and another one for even addresses, the adjacent addresses can be accessed simultaneously. For example, four banks can be used to access four pixels in a 2×2 block. In addition, for efficient access to the pixels in the aperture, the address can be coded as proposed in [3].

In the pipelined random access to RAM, one process can write results to one memory bank, and another can read data from the second bank. When the processing of the next data array is completed, the banks switch their roles. At the same time, a third memory bank is used for better synchronization [4]. But such switching of banks adds a long period of time to the latency of the algorithm and has the consequence of increasing the hardware costs of the system, and the use of more FPGA pads.

A more practical approach is to run the memory at a higher clock frequency than the rest of the system. Double data rate (DDR) memory is one example of memory that allows data to be transferred twice per clock. As a rule, modern high-capacity FPGAs have dedicated outputs and a built-in access controller for external dynamic DDR memory of recent generations [5]. At the same time, the project simulates multiport memory due to access time slots. In addition, blocks of the buffer memory are required for writing and reading, since dynamic memory has high throughput only when transferring rows of data from neighboring cells. Unfortunately, in many projects, DDR memory is also required to support the operating system of the processor embedded in the FPGA, and therefore the bandwidth of this memory drops when processing large data arrays.

Pipeline and FIFO first-in-first-out (FIFO) type buffers are two popular types of memory organization methods utilized in FPGA. They are distinguished in the following. The data stored in and retrieved out of a pipeline is also in the first-in-first-out category. But the steps of storage and retrieval are constant as in the serial-in-serial-out shift register. A FIFO buffer is a storage where the data can be pushed into and popped out with the same data order, but these operations can be uncorrelated. However, the implementation of both long pipeline buffer and FIFO is based on RAM which is operating as the circular buffer. The method of such buffers design is explained in [6]. But the designer must organize the proper order of data pushing and popping separately.

If the data are executed sequentially, then it is worth using the buffers of the FIFO type, which cell groups store blocks of data, and the output data are selected by the local addresses [7].

When the algorithm can be represented by some Petri net, then the stream processing computational model can be used. In this model, the computational node or processing kernel consists of the stencil buffer and computing module connected to the buffer inputs. During the computational process, the input data are loaded into the buffer asynchronously and just when they form the proper stencil the computations start [8]. So, the buffer is really the register pipeline with a large set of outputs, which is often the inefficient solution.

When the usual serial program is mapped in the hardware, then the data buffer with the last-in-first-out discipline is needed. The method of such stack buffer design as well as the respective finite state machine development named Hierarchical Finite State Machines (HFSM) method is described in [9].

The von Neumann architecture paradigm is widely used in which each datum has its own robust address in the common address space. The data buffers are implemented as the cache memory blocks in this paradigm. Note, that in particular, when the data lose their addresses in the moment before their execution, then this cache memory can be represented as a usual FIFO buffer. Therefore, the usual data buffer is often called the cache [10]. The method of the cache buffer design for FPGA based on the optimized data throughput is described in [11]. When the FPGA application deals with dynamic memory allocation, then the cache buffers can be designed using the method of algorithm analysis which selects the independent and shared memory fields [12].

The dataflow processing is the kind of algorithm that is usually implemented in FPGA because the FPGA architecture provides the effective implementation of such algorithms. The most common model for the dataflow algorithm representation is the Kahn processing network (KPN). The nodes of this network represent the operations or actors, and the edges represent the dataflows. The edges contain the FIFO buffers of the proper length. Usually, KPN is mapped into FPGA by one-to-one mapping. So, the FIFO buffers serve as the proper data buffers [13]. Note, that this model considers that the data are retrieved from FIFO in arbitrary order, i. e., the buffer can contain several outputs from its head registers.

The unified modeling language or UML provides an effective KPN representation. Many tools like IBM Rational Rhapsody provide translation of the UML description into hardware [14]. The Matlab Real-Time Workshop (RTW) tool offers code generation capabilities directly from Simulink graphical system descriptions which is a kind of KPN [15]. These tools implement the FIFO buffers as they are foreseen in the given KPN. But these buffers must obey the rules of the asynchronous reading and writing data in them in the respective order.

The synchronous dataflow (SDF) graph is the abridged KPN model, in which all dataflows are synchronous. Note, that two dataflows are synchronous if the data in one flow are correlated with the data in the other one, for example, both data samples have the same index sets. The FIFO buffers in the SDF model are always synchronous ones, and this model is usually free of deadlocks. This model gives simple mapping to hardware, providing effective methods of structure optimization like pipelining, retiming, folding, and resource sharing [16]. This idea is expanded and fulfilled in the SDF modeling framework Ptolemy [17]. By this method, the optimized data buffers are synthesized as well. But the synthesis results can be far from excellent because the optimization is performed by hand or automatically. Through this process, the effective schedule is searched which disagrees with the hardware minimization.

When the algorithm given by SDF has no loops and feedback then it is usually represented by the dataflow graph (DFG). Then, the data buffers with the minimum volume can be synthesized using the method proposed in [18]. This method combines the register allocation by the left-edge scheduling and the SDF folding.

When the 2D signals or images are processed, then the problem of the buffer design becomes more complex. In this situation, the multidimensional SDF can be used, in which the data have the vectors of indexes which can be considered as the pixel coordinates in the image frame [19]. But the buffer design remains a complex task.

Many algorithms including ones of image processing are represented by the loop nest. The index vectors of the loop nest iterations and the data themselves form the multidimensional grid, and the algorithm does the respective lattice-like DFG. The method of the systolic processor design is widely used for mapping these algorithms both into the processor structure and into the timetable of the operator execution [16]. The pipelined data buffers are the obligatory result of such mapping. Therefore, this method is widely used now to design data buffers in many synthesis methods and automatic design frameworks.

Placing the operators in the iteration space and mapping them in the structure and timetable is used in [20] as well. To optimize the data buffers, the system of linear inequalities which takes into account the operator data dependencies, data moving delays, and time limitations. This system is solved using the usual integer linear problem solver. As a result, the throughput is optimized and the pipelined data buffers are synthesized. But the synthesis process becomes very complex when the problem dimensions increase.

This method is expanded using the polyhedral model of the parallel algorithm DFG representation and its mapping [21]. Due to this method, the executed iterations of the algorithm and their data form the polyhedron in the multidimensional iteration space which limits the volume of the lattice-like DFG. Each iteration in it occupies a particular integer vector in the space. This polyhedron is mapped into the systolic structure of the computer and the timetable using the optimized affine transformations of this space. When the loop nest describes the data array behavior then the result of the mapping is a set of pipelined data buffers. A similar method is proposed in [22]. The method named lattice-based partitioning is based on the same principle and performs the selection of a set of distributed buffers [23].

FPGA hardware is utilized very well providing high throughput when the data are reused frequently. The method of the buffer design described in [24] provides the data reusing when the algorithm performs the sequential array processing using the modulo addressing. A more sophisticated method utilizing data reuse is proposed in [25]. The approach of the systolic processor design is implemented in it and the data which are fetched from the one- or two-dimensional array are reused in the algorithm.

The buffers of different lengths should be designed for different data array sizes. It is proposed to use a universal buffer, which is adjusted to the array size and the computed frame in it with the possibility of dynamic reconfiguration [26]. A similar method for image processing is described in [27], which is capable of transposing the position of pixels in the frame, as well as performing image correction at the frame edges.

The works [28, 29] present general methods of designing a pipelined structure for image processing with a sliding aperture selected for processing. At the same time, the functions that are sequentially performed in the algorithm are mapped in the corresponding processing blocks, which are separated from each other by buffer blocks that store several adjacent lines. The interconnections between processing blocks and buffer blocks are buses that correspond to the edges of DFG.

The smart buffer is a compiler-generated data buffer that provides re-using the fetched data in the sliding aperture. The structure of the buffer is determined by the window size, array size, and the stride of the reuse in each dimension [30]. This method is effectively utilized in the Riverside optimizing compiler for configurable computing (ROCCC) approach and compiler [31].

Goals of the investigation

The analysis of different methods of the data buffer design makes it possible to conclude the following.

KPN mapping gives a set of pipelined data buffers in a natural manner. However, the resulting buffers have several output ports in many cases and the deadlock problem is solved hard.

SDF is the abridged model of KPN, but it is a rather impressive one and it is free of deadlocks. Many dataflow algorithms like digital signal processing are represented as SDF and are effectively mapped into hardware structures including pipelines and FIFOs.

The most sophisticated and formalized methods are ones that are based on the representation of the algorithm as DFG in the multidimensional grid and mapping it into the systolic-like processor structures. Many of them are implemented in high-level synthesis frameworks. But these methods are limited by the algorithms which are represented by the loop nests and do not take into account the features of the hardware technology.

The goal of the investigation is to develop a new method of data buffer design that is more sophisticated and is able to take into account the features of the FPGA architecture. The method is intended for the pipeline buffer design however it is fitted for the buffers based on RAM. These buffers are designed in general for the streaming algorithms like DSP, image processing, or others that can be represented by SDF.

The derived buffers must be optimized both in the clock frequency and in hardware. Therefore, first of all, the FPGA features are considered. Then, the method of the pipelined datapath design is selected which involves the better features of the methods considered above. And next this method is adapted to the data buffer design.

FPGA resources for the buffer design

The FPGA chip usually contains sufficient volume of different memory resources. Usually, the basic building block is the Look-Up Table (LUT) in Xilinx FPGAs or Adaptive Logic Module (ALM) in Intel FPGAs. Each of them is accompanied by one or two 1-bit registers. These registers usually form the storage elements of the pipeline stages including the pipeline buffers. LUT by itself is configured as the buffer RAM with a volume of up to 64 bits, and with several possible reading ports. Moreover, it can be configured as the pipeline buffer of the variable length. Fig. 1 illustrates the structure of such an SRL16 primitive which contains the 16-bit shift register, and each of its taps is selected statically or dynamically by the output multiplexor.

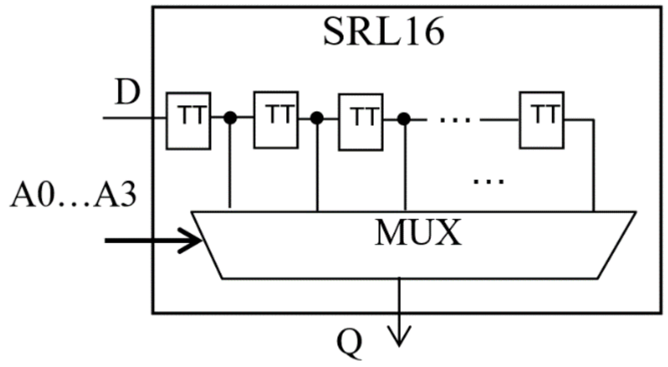

Fig. 1. Pipeline buffer SRL16 structure

FPGA contains from tenths to thousands of two-port blocked RAMs (BRAMs). Each of them contains kilobytes of memory of programable bit width. The ratio of BRAM number to LUT number in FPGA is equal from 60 to 200. Usually, they can be configured as FIFO buffers [32, 33].

The Intel Hyperflex FPGA architecture provides the pipeline buffers of the arbitrary length in the routing segments in the inter ALM communications. These buffers enable the highest clock frequencies in Intel Stratix® 10 and Intel Agilex™ devices [34].

Usually, the most effective structure solutions are derived from the register transfer level (RTL) design. But in such a design, the buffer selection, and its dynamic control, which depends on the modules attached to it, is a hard design task. Therefore, the usual solution is selection the FIFO buffer based on BRAM, which takes increased hardware volume. The SRL16 buffers are utilized rarely in some specific finite state machines (FSMs), filters, or encryptors [35]. The Hyperflex register utilization in the projects takes specific knowledge about the SDF optimization and is not fulfilled in most cases when SDF contains the loops [34].

Spatial SDF method

A method of designing the pipelined datapaths by mapping SDF is proposed in [36, 37]. The feature of the method is that SDF is represented in the resource-time space in the form of an algorithm configuration (AC). The method makes it possible to search for a schedule, minimize the number of processor units (PUs), and search for effective interprocessor connections simultaneously. Here, PU means an elementary computing element with or without result registers, for example, an adder, a multiplier with a register, a pipeline buffer, etc. Therefore, it makes sense to create a method for the data buffers development based on this method. It is described below in short.

At the first stage of the synthesis, according to the specified method, operators-nodes of a homogeneous SDF together with the data dependency edges are located in three-dimensional space $\mathbb{Z}^3$ as sets of vectors K_i and D_j, respectively, taking into account the conditions, given in [36]. The coordinates of the vector $K_\mathrm{i} = (s,q,t)^\mathrm{T}$ mean the number s of the PU, where the operator is executed, the type q of this PU, and the time component t, which is equal to the clock number during the execution of the algorithm. Vectors K_i with equal time components form one row and are executed simultaneously. The time component $R(D_\mathrm{j})$ of the vector $D_j = K_i - K_l$ is equal to the delay between the executions of operators whose nodes K_i, K_l are adjacent. The number of PUs is minimized by fulfilling the requirements $|K_{s,q}|\rightarrow L$, i.e. the number of nodes mapped in the s-th PU approaches to L, where L is the algorithm execution period in clock cycles. In addition, when forming the effective algorithm configuration, it is desirable to build a perfect spanning tree of SDF, as suggested in [38].

In the second step, AC is balanced, which consists in adding delay nodes to the edges of SDF until the time components of all vectors D_j are equal to 0 or 1. After that, AC is optimized by permuting the node vectors from the same column in order to minimize the number of registers and the number of multiplexer inputs in the resulting structure and/or using other strategies, for example, retiming. Also, the number of registers is minimized by gluing delay nodes from the same column that store the same operand.

In the third step, the obtained optimized AC is mapped in the graph of the computer structure in the subspace $\mathbb{Z}^2$ named as the structure configuration. This is done by gluing the node vectors with the same coordinates $s,$ and q. AC is transformed into the schedule of operator execution, using the property that the time component of the vector K_i is equal to the moment of execution of the operator, regardless of the number of the execution period. At the same time, the resulting structure is not built and the schedule is not formed because the resulting structure is described in VHDL on the base of information in AC.

Method for the buffer design

Consider AC $C'_{\mathrm{A}v}$ which performs the iterative algorithm with the period of $L = 4$ clock cycles, and which consists only of input and output nodes. This AC is mapped into the data buffer. When placing the nodes of CA in the space $\mathbb{Z}^3$, one should use some strategies to minimize the number of connections between PUs. The location of the nodes of the delay operators according to the strategy of placing the edges $D_{i,j}$ in parallel to the axis Ot in the second step of the synthesis is shown in Fig. 2,

a. And the configuration C'_{Av} according to the strategy of placing the edges $D_{i,j}$ at an angle to the axis Ot is shown Fig. 2, b. The structure configurations corresponding to these CAs are shown in Fig. 2, c, and 2, d, respectively. Here, the bold points mean the nodes of input-output or some operator nodes, and circles mean the delay nodes mapped into the registers. The bars mean the multiplexers attached to the PU inputs, which perform the selection of the operand when it is read from the respective register.

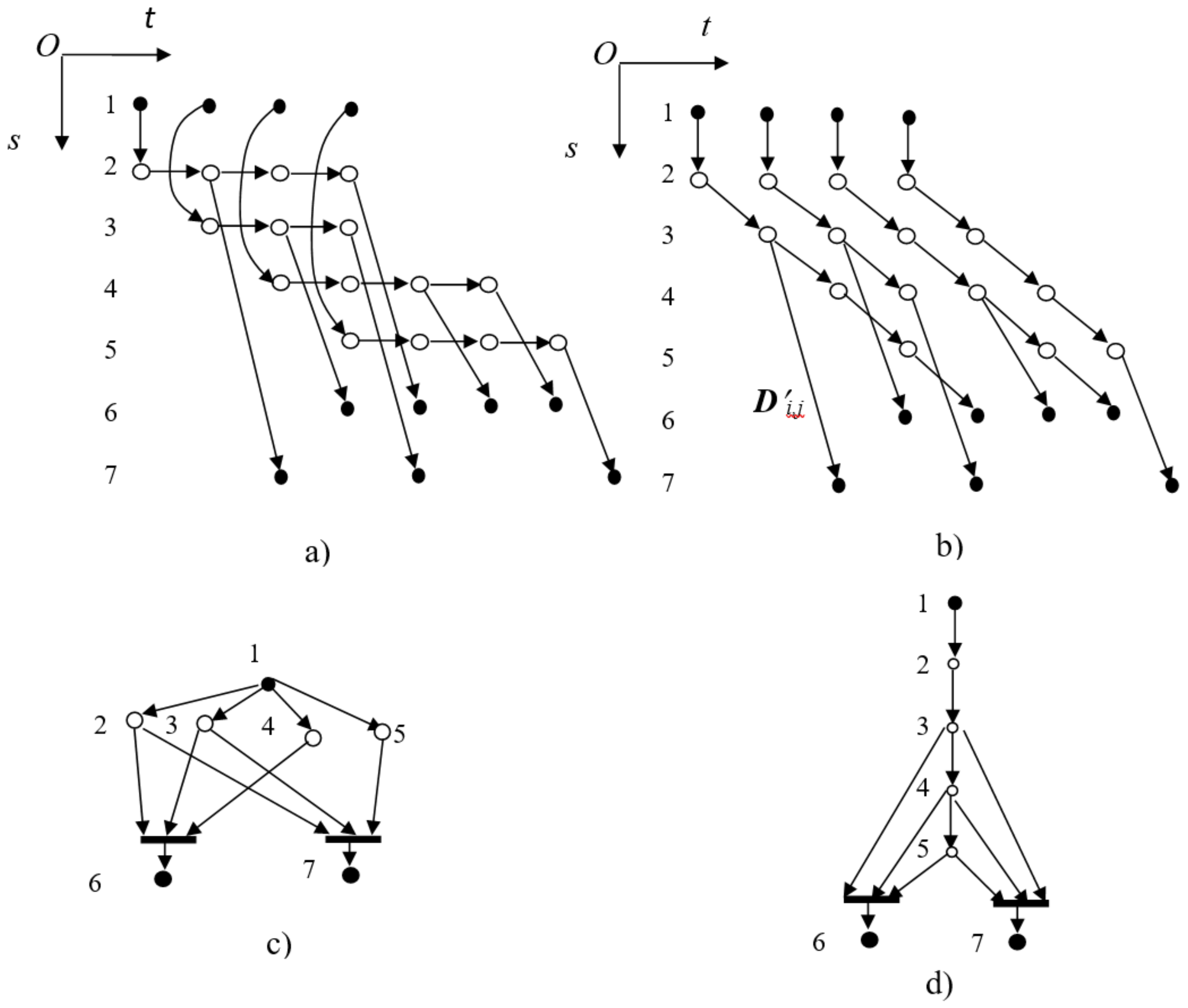

Fig. 2. AC which edges placed according to the strategy of RAM (a) or pipeline buffer (b) synthesis, and respective RAM (c) and FIFO (d) configurations

Analysis of these structure configurations shows that they correspond to two-port RAM (one port to read-write, second one only to read) and pipelined data buffer, respectively. Applying one or another strategy of connection number minimizing, the designer can orient the process of synthesis of the data buffer to implementation in the form of RAM or a register pipeline. The strategy should be chosen taking into account the following features.

When synthesizing the buffer based on RAM, the variable x_i is allocated in the respective register, i.e. the chain of delay nodes is located on a straight, which is parallel to the axis Ot. Also, one register is assigned to several variables whose periods of existence do not overlap, i.e. several chains of delay nodes are located on a straight, which is parallel to the axis Ot, and these chains do not overlap. At this process, the edges $\boldsymbol{D}_{i,j}$, which are adjacent to the outputs of the edges $\boldsymbol{K}_{i,j}$ of the AC before balancing the relation

$$\max_{i,j}(t_{\mathrm{D}i,j}) \leq L \tag{1}$$

is satisfied, where $t_{\mathrm{D}i,j}$ is the time component of the vector-edge $\boldsymbol{D}_{i,j}$. If it is not observed, it is necessary to cut the balanced AC C'_{Av} into several subconfigurations, each of which will corresponds to its own

RAM or ensure overwriting of the variable x_i for which it is not observed the inequality (1), in the second register of the RAM after L clock cycles. It is obvious that the volume of the resulting RAM for AC C'_{Av} with λ input nodes (bold points in Fig. 2) is equal to

$$N_P = \lambda. \tag{2}$$

When the pipeline buffer is designed, then the variable x_i is sent to the adjacent pipeline register in each clock cycle and, passing through a chain of $t_{Di,j}$ registers is outputted from it to the input of PU which receives this variable. This is equivalent to the fact that the chains of adjacent nodes $\boldsymbol{K}_{i,j}$ of the delay operators at uniformly increasing coordinates $s_{i,j}$, and $t_{i,j}$ are placed along parallel lines, located at an angle to the axis *Ot* (Fig. 2, b). Therefore, the value of $t_{Di,j}$ in (1) can be any, however, to minimize the number of the register pipeline stages, the number of different values of the vectors $\boldsymbol{D}'_{i,j}$ must be minimal. The number of registers in the pipeline is equal to

$$N_P = \max_{i,j}(t_{Di,j}). \tag{3}$$

Thus, AC which performs the data transfer between input and output ports after its balancing and optimization according to one of two strategies gives a minimized amount of memory in the resulting data buffer. We get a buffer structure with memory organized in the form of RAM or a register pipeline. At the same time, the number of registers in RAM is smaller than in the pipeline of registers, if the number of input nodes that are mapped to one port node (the number of different variables entering one PU) in AC is less than the maximum delay of the variable that is calculated in this PU, i.e. at

$$\lambda = \max_{i,j}(t_{Di,j}). \tag{4}$$

When the resulting pipelined buffer is performed in the SRL16 primitive, then the method must take into account the fact that it has a single output (see Fig. 1). This adds the additional limitation to AC placement in the space that only a single edge must connect any delay node with the node which is mapped into the output port PU. AC in Fig. 2, b does not satisfy this condition. Therefore, it is split into two subconfigurations in Fig 3, a, which satisfies it and is mapped into the structure with two units implemented in SRL16 primitives (Fig 3, b).

The SRL16 primitive has an additional clock enable input, the control of which makes it possible to slow down the data moving through the pipeline registers. When using this input, the number of registers can be minimized if the value of $R(\boldsymbol{D}_j)$ is greater than the number of available registers in the pipeline. Fig. 4 shows an example of the transformation of AC, shown in Fig. 3, a, for the purpose of additional delay of the operands. Such delays correspond to the vectors $\boldsymbol{D}_j$, which are placed parallel to the axis *Ot*. Note, that the number of nodes that have the same coordinate s must not be higher than the computation period L.

The Fig. 4 analysis shows that the technique of the clock enable control allows us to minimize both the pipeline register number and output multiplexers substantially. This is important when the pipeline registers are performed on the base of usual registers because it saves hardware and minimizes the clock period.

If the nodes-sources of considered AC have different spatial coordinates s (in the examples above s = 1), then an input multiplexor is obtained at the input of the SRL16 primitive. To minimize such multiplexers, the method can be used which is described in [39].

Thus, the method of designing the pipelined datapaths with buffers based on SRL16 primitives looks like the following. The initial data are AC, algorithm execution period L, and other optimization parameters. The method is performed in the same way as described in [37, 38], with the exceptions described below.

In the first stage of synthesis, the AC subgraphs corresponding to the transfer of operands between computer resources with time delays and/or shuffling of operands, which are expected to be mapped into separate data buffers, should be selected.

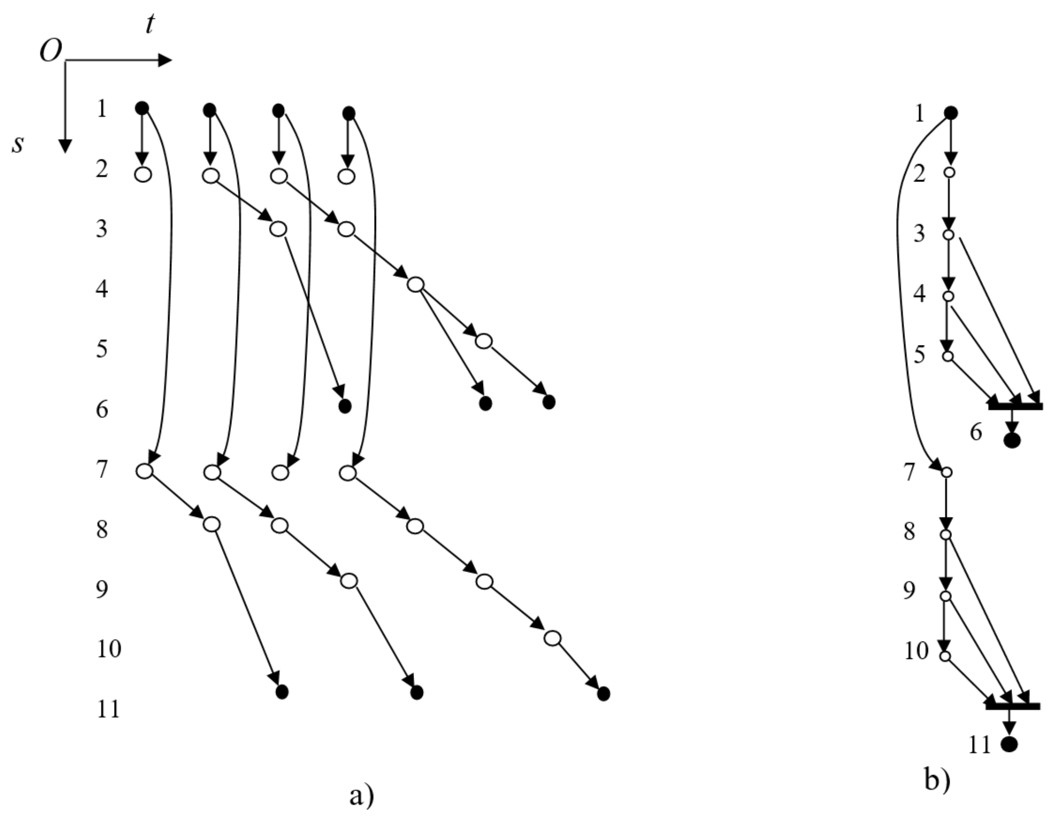

Fig. 3. AC which is split to AC in Fig. 2, b (a) and its mapping into SRL16 structures (b)

In the first stage of synthesis, the AC subgraphs corresponding to the transfer of operands between computer resources with time delays and/or shuffling of operands, which are expected to be mapped into separate data buffers, should be selected.

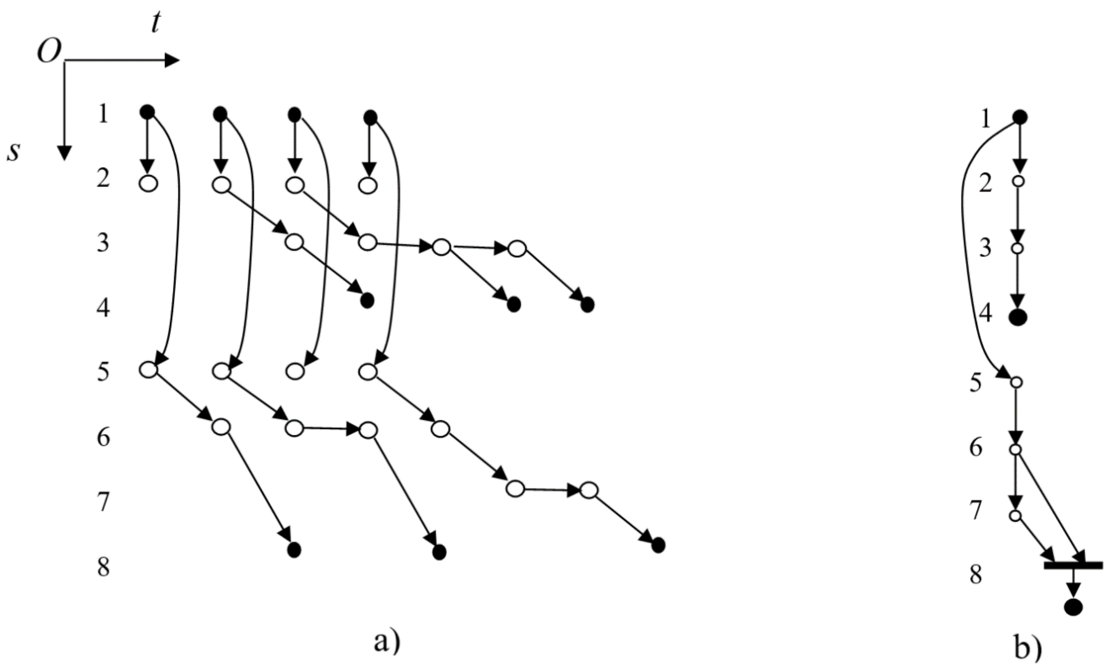

Fig. 4. Modified AC Fig. 3, a (a), and its mapping into SRL16 structures (b)

In the second stage, it is necessary to balance the dependence edges using the intermediate delay nodes. The number of intermediate delay nodes is minimized, if possible. The delay nodes are placed on parallel lines that are at an angle to the time axis or parallel to this axis in such a way that adjacent delay nodes differ in time coordinates by one beat. The requirements for the correct placement of nodes are fulfilled, including the requirement to implement a buffer with one input and one output. If it is impossible to get a single input in the buffer, the heuristic of minimizing the number of inputs of the additional multiplexer at the buffer input is used according to [39], and if it is impossible to receive a buffer with one output, the chain of delay nodes is split so that they are mapped in additional buffers (see Fig. 4).

The dependency edges together with the corresponding delay nodes which are incident to the nodes consuming the buffered data should be mapped in the data buffer. When a control algorithm is designed, if only edges are displayed in the buffer that is at an angle to the time axis, then operands are written to the buffer in each clock cycle. If there are edges that are parallel to this axis, then writing to the buffer is prohibited in the corresponding clock cycles (see Fig. 4).

At the third stage, the pipelined datapath is described in VHDL according to the method presented in [38] and is compiled into an FPGA configuration that contains the buffers based on SRL16 primitives, which correspond to the selected AC subgraphs.

Experimental results

Consider the design of the input buffer for the pipelined datapath performing the 8-point discrete cosine transform (DCT). The DFG of this algorithm is often based on the Chen algorithm [41]. This algorithm is distinguished in that its period of the pipelined computations is equal to $L = 8$ clock cycles, eight input data of a single DCT transform need to be delayed and permutated in the input buffer before their calculations. DFG of the first stage of this algorithm which needs the data buffer is shown in Fig. 5.

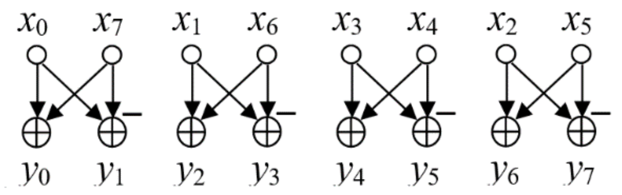

Fig. 5. DFG of the first stage of the DCT algorithm

Optimized AC which is mapped into pipelined buffer and adder, and respective structure configuration are illustrated in Fig. 6. Here, the resource names are placed in the *Os* axis and the clock cycle number modulo $L = 8$ is mapped in the axis *Ot*. The addition-subtraction operator node has the plus sign. This AC is described in VHDL as follows.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.Numeric_STD.all;
entity DCT_BUF is
       port(
             CLK : in STD_LOGIC;
             RST : in STD_LOGIC;
             START : in STD_LOGIC;
             X : in SIGNED(8 downto 0);
             Y : out SIGNED(8 downto 0)
       );
end DCT_BUF;
architecture synt of DCT_BUF is
       type TARRAY16 is array (0 to 15) of SIGNED(8 downto 0);
       type TN is array(0 to 7) of natural range 0 to 15;
       constant al: TN:=(7,8,8,9,8,9,11,12);
       constant ar: TN:=(0,1,3,4,7,8,8,9);
       signal r1,r2:TARRAY16; -- register array of SRL16
       signal cycle:natural range 0 to 7;
       signal sm,l,r: SIGNED(8 downto 0);
begin
       CT8:process(CLK) begin -- period counter
             if CLK'event and CLK='1' then
                   if START='1' then
                         cycle<=0;
                   else
                         cycle<= (cycle+1) mod 8;
                   end if;
```

```
                end if;
        end process;

      l<= r1(al(cycle));
        r<= r2(ar(cycle));
        SRL16_BUF:process(CLK) begin -- SRL16 description
                if CLK'event and CLK='1' then
                        r1<=X & r1(0 to 14); -- FIFO shift
                        r2<=X & r2(0 to 14); -- FIFO shift
                        case(cycle) is
                                when 0|2|4|6 => sm<= l + r; -- adder
                                when others => sm<= l - r; -- subtractor
                        end case;
                end if;
        end process;
        Y<=sm;
    end synt;
```

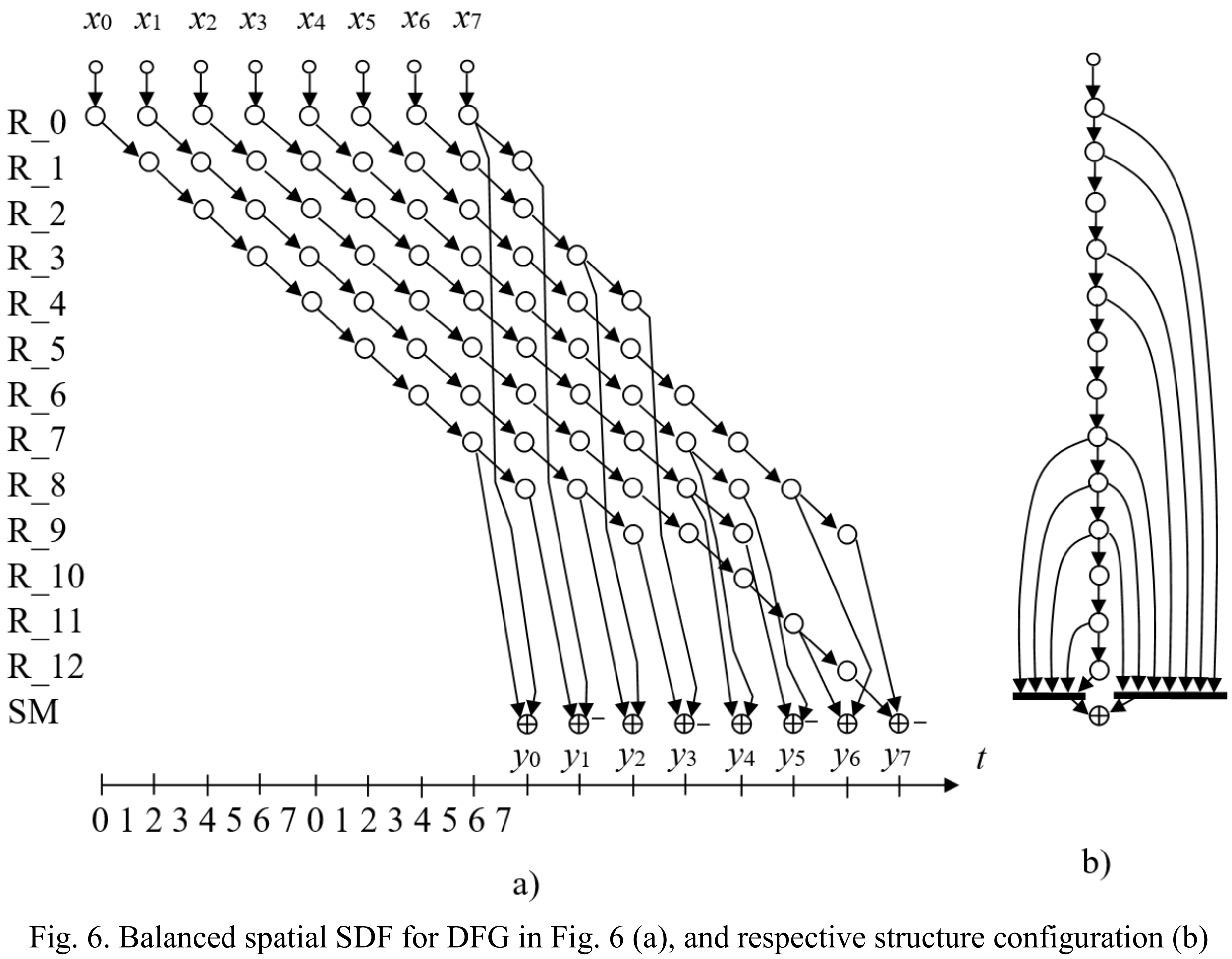
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
R_0
R_1
R_2
R_3
R_4
R_5
R_6
R_7
R_8
R_9
R_10
R_11
R_12
SM
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7
t
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
a)
b)

Fig. 6. Balanced spatial SDF for DFG in Fig. 6 (a), and respective structure configuration (b)

Here, signals *r*1, *r*2 represent two pipeline register chains, which load the input data *X* in each clock cycle. They are synthesized after splitting AC in Fig. 6, a in two subconfigurations like it is done in Fig. 4. The signals from them *l*, *r* are read at addresses which are sampled from ROMs *al*, *ar*. These signals are directed to the left and right inputs of the adder-subtractor with the register *sm* deriving the result *Y*. The calculating period counter cycle counts modulo $L = 8$ and controls both the sign of the adder *sm* and the pipeline register chains through the ROMs *al, ar*.

This project is compiled by the Xilinx ISE and Vivado CAD packages into FPGAs of different series. The results of compilations are shown in Table 1.

This table analysis shows that the ISE synthesizer recognizes the template of the SRL16 primitive and the synthesis results are the data buffers with the minimum hardware volume and good performance. The Vivado synthesizer first tries to compound both pipeline buffer branches into one and then minimizes the trigger number by substituting the chains of registers with the SRL16 primitives. The inferred structure is illustrated in Fig. 7. One can see, that additionally, the synthesizer doesn`t perform the resource sharing of the adder-subtractor. As a result, the hardware volume in the register number is much higher.

Table1.
Results of a configuration of the buffer project in FPGAs

FPGA series	Compiler	Slice Flip Flops	LUTs	LUTs used as logic	LUTs used as SRL16	Minimum clock period, ns
Virtex-4	ISE 14.7	12	37	19	18	3.14
Spartan-3A	ISE 14.7	12	37	19	18	5.47
Spartan-6	ISE 14.7	12	39	29	10	4.72
Artix-7	Vivado2016	111	44	39	5	4.06

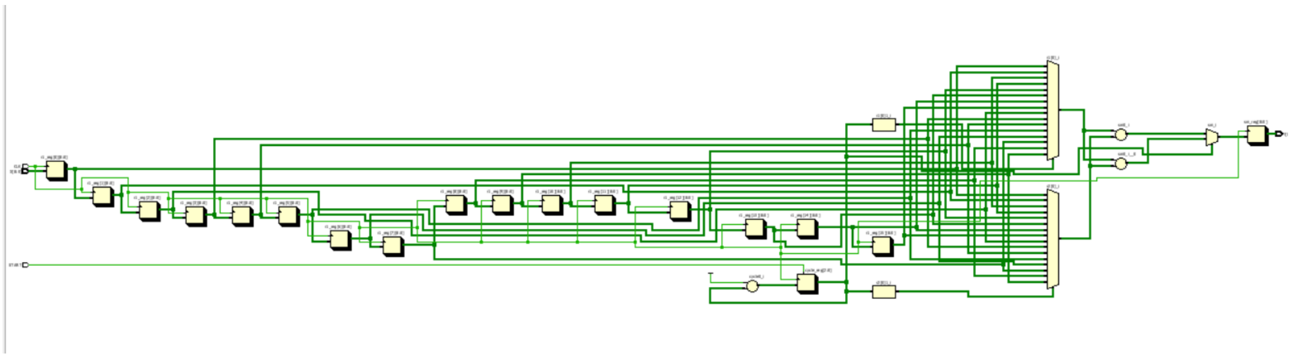
Fig.7. Data buffer structure derived by the Vivado design tool

Synthesis framework

As one can see from the method description and the design example, the considered algorithm is given in the graphical form effectively. For the design method investigations, the synthesis framework is developed named SDFCAD [42]. The framework is able to perform the graphical input of SDF of the DSP algorithms with the given period *L* and data bit width. SDF can be optimized either manually or automatically using one of the genetic programming algorithms [40]. One of two strategies of the buffer design are used by the optimization as well. In particular, the pipelined buffers for the DCT processor are synthesized automatically very well [42].

Conclusions

A new method of the data buffer design is proposed, which is intended for the complex pipelined datapaths development and configuring in FPGA. The method is based on the SDF representation in the three-dimensional space, optimization them and describing in VHDL. Depending on the optimization method the derived buffer is based either on RAM or on the register pipeline. The feature of the method consists in that the pipeline buffer is inferred into the SRL16 primitives of the AMD-Xilinx FPGA series which substantially saves the hardware. The method is built in the experimental SDFCAD framework intended for the pipelined datapath synthesis.

References

1. FPGAs for Software Programmers. D. Koch, F. Hannig, D.Ziener – Ed-s. Springer, 2016. 327 p.
2. Meyer-Baese U. Digital Signal Processing with Field Programmable Gate Arrays. 4th Ed. Springer. 2014. 930 p.
3. Kim, K., Kumar, V.K.P. Parallel memory systems for image processing. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, San Diego, California, USA (4–8 June, 1989), 1989, pp. 654–659. DOI: https://doi.org/10.1109/CVPR.1989.37915
4. Khan, S., Bailey, D., Sen Gupta, G. Simulation of triple buffer scheme (comparison with double buffering scheme). Proceedings of the 2nd International Conference on Computer and Electrical Engineering (ICCEE 2009), Dubai, UAE, 2009. Vol. 2, pp. 403–407. DOI: https://doi.org/10.1109/ICCEE.2009.226

5. Churiwala S. (). Designing with Xilinx® FPGAs: Using Vivado. Springer, Switzerland. 2017.
6. Sadrozinski H. F.-W., Wu J. Applications of Field-Programmable Gate Arrays in Scientific Research. CRC Press and Taylor & Francis. 2011. 144 p.
7. Sedcole, P., Cheung, P.Y.K., Constantinides, G.A. Luk, W. (). Run-time integration of reconfigurable video processing systems. *IEEE Transactions on VLSI Systems,* 2007. Vol. 15. No. 9, pp. 1003–1016. DOI: https://doi.org/10.1109/TVLSI.2007.902203
8. Sano K., Nakahara H. Hardware Algorithms. In: *Principles and Structures of FPGAs*, H. Amano –Ed. Springer, 2018, pp. 137-177.
9. Sklyarov V., Skliarova I., Barkalov A., Titarenko L. Synthesis and Optimization of FPGA-Based Systems. Springer, 2014, 432 p.
10. Bailey D. G. Design for Embedded Image Processing on FPGAs. Wiley-Blackwell. 2011. 482 p.
11. Sass R., Schmidt A. G. Embedded Systems Design with Platform FPGAs. Principles and Practices. Morgan Kaufmann Pub. 2010. 389 p.
12. Winterstein F., Fleming K., Yang H.-J., Bayliss S., Constantinides G. MATCHUP: Memory Abstractions for Heap Manipulating Programs. FPGA '15: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. February, 2015, pp. 136–145. DOI: https://doi.org/10.1145/2684746.2689073
13. Woods R., McAllister J., Lightbody G., Yi Y. FPGA-based Implementation of Signal Processing Systems. 2nd Ed. Wiley, 2017. 448 p.
14. Granado L., Berreteaga O. Creating Rich Human-machine Interfaces with Rational Rhapsody and Qt for Industrial Multi-core Real-time Applications. *Procedia Manufacturing*. 2015. Vol.3, pp. 1903 – 1909. DOI: https://doi.org/10.1016/j.promfg.2015.07.233
15. Hwang J, Milne B, Shirazi N., Stroomer J. D. System Level Tools for DSP in FPGAs. *Proc. 11th Int. Conf. on Field Programmable Logic and Applications*, 2001, pp. 534–543. DOI: https://doi.org/10.1007/3-540-44687-7_55
16. Parhi K. K. VLSI Digital Signal Processing Systems. Design and Implementation. Wiley, 1999. 784 p.
17. Lee E. A., Neuendorffer S., Wirthlin M. J. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits System and Computers*. 2003. Vol. 12, pp. 231-260. DOI: https://doi.org/10.1142/S0218126603000751
18. Pedersen M. R., Madsen J. Optimal register allocation by augmented left-edge algorithm on arbitrary control-flow structures. *NORCHIP'2012*, 2012, pp. 1-6. DOI: https://doi.org/10.1109/NORCHP.2012.6403107.
19. Murthy P. K., Edward A. Lee E. A. Multidimensional Synchronous Dataflow. *IEEE Transactions on Signal Processing*. 2002. Vol. 50. No. 8, pp. 2064 – 2079. DOI: https://doi.org/10.1109/TSP.2002.800830
20. Cong J., Jiang W., Liu B., Zou Y. Automatic memory partitioning and scheduling for throughput and power optimization. *2009 IEEE/ACM International Conference on Computer-Aided Design*. Digest of Technical Papers, 2009, pp. 697-704. DOI: https://doi.org/10.1145/1687399.1687528.
21. Yu Y.W., Li P., Cong J. Theory and algorithm for generalized memory partitioning in high-level synthesis. *FPGA '14: Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays.* Feb. 2014, pp. 199–208. DOI: http://dx.doi.org/10.1145/2554688.2554780
22. Cong J., Wang J. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1-8, DOI: http://10.1145/3240765.3240838
23. Gallo L., Cilardo A., Thomas D., Bayliss S., Constantinides G. A. Area implications of memory partitioning for high-level synthesis on FPGAs. *24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1-4, DOI: http://10.1109/FPL.2014.6927417
24. Wang Y., Zhang P., Cheng X., Cong J. An integrated and automated memory optimization flow for FPGA behavioral synthesis. *17th Asia and South Pacific Design Automation Conference*, 2012, pp. 257-262, DOI: http://10.1109/ASPDAC.2012.6164955

25. Guo Z., Walid Najjar W., Buyukkurt B. Efficient hardware code generation for FPGAs. *ACM Transactions on Architecture and Code Optimization*. Vol. 5. No 1, 2008, pp. 1–26. DOI:https://doi.org/10.1145/1369396.1369402
26. Shi R., Wong J. S. J., So H. K. High-Throughput Line Buffer Microarchitecture for Arbitrary Sized Streaming Image Processing. *J Imaging*. 2019. Vol. 5. No 3, 34 P. DOI: https://doi.org/10.3390/jimaging5030034
27. Bailey D. G., Ambikumar A. S. Border Handling for 2D Transpose Filter Structures on an FPGA. *Journal of Imaging*. 2018. Vol. 4. No 12, 138 P. DOI: https://doi.org/10.3390/jimaging4120138
28. Ikarashi Y., Ragan-Kelley J., Fukusato T., Kato J., Igarashi T.. Guided Optimization for Image Processing Pipelines. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC),* 2021, pp. 1-5, DOI: https://doi.org/10.1109/VL/HCC51201.2021.9576341
29. Özkan, M. A., Reiche, O., Hannig, F., Teich, J. (). FPGA-based accelerator design from a domain-specific language. *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL)*, Lausanne, Switzerland, 2016, pp.1–9. DOI: https://doi.org/10.1109/FPL.2016.7577357
30. Guo Z., Najjar W., Buyukkurt B. Efficient hardware code generation for FPGAs. *ACM Trans. Archit. Code Optim.* 2008. Vol. 5, No 1, pp. 6:1–6:26. DOI: https://doi.org/10.1145/1369396.1369402
31. Najjar W. A., Villarreal J., Halstead R. J.ROCCC 2.0. In: FPGAs for Software Programmers. D. Koch, F. Hannig, D.Ziener – Ed-s. Springer, 2016, pp. 191-204.
32. UltraFast Design Methodology Guide for the Vivado Design Suite. UG949 (v2013.3) October 23, 2013. 361 p. URL: www.xilinx.com
33. 7 Series FPGAs Memory Resources User Guide. UG473 (v1.14) July 3, 2019. 88 p. URL: www.xilinx.com
34. Intel® Hyperflex™ Architecture High Performance Design Handbook. Ver.: 2021.10.04. 147 p. URL: https://www.intel.com/programmable/technical-pdfs/683353.pdf
35. Chu J., Benaissa M. Low area memory-free FPGA implementation of the AES algorithm. *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 623-626, DOI: https://10.1109/FPL.2012.6339250
36. Sergiyenko A., Maslennikow O., Vinogradow Y. Tensor approach to the application specific processor design. *2009 10th International Conference - The Experience of Designing and Application of CAD Systems in Microelectronics*, 2009, pp. 146-149.
37. Sergiyenko A., Serhienko A., Simonenko A. A method for synchronous dataflow retiming, *2017 IEEE First Ukraine Conference on Electrical and Computer Engineering (UKRCON),* 2017, pp. 1015-1018, DOI: https:// 10.1109/UKRCON.2017.8100404.
38. Sergiyenko A. M., Simonenko V. P. Method of synchronous dataflow scheduling. *System research and information technologies*, 2016. № 1, pp. 51-62. DOI: https://10.20535/SRIT.2308-8893.2016.1.06
39. Sergiyenko A. M., Simonenko V. P. Otobrazenie perioditsheskich algorithmov w programmiruemye logitsheskie integralnye schemy. *Electronic Modeling*. 2007. Vol. 29. № 2, pp. 49-61. (In Russian).
40. Sergiyenko A., Serhienko A., Romankevich V. Genetic Programming of Pipelined Datapaths for FPGA, *2020 IEEE 40th International Conference on Electronics and Nanotechnology (ELNANO),* 2020, pp. 802-806, DOI: https://10.1109/ELNANO50318.2020.9088773.
41. Nikara J., Takala J., Akopian D., Saarinen J., Pipeline Architecture for DCT/IDCT. *IEEE Int. Symp. on Circuits and Systems, (ISCAS 2001),* May 6-9, Sydney, Australia, 2001. P. 902–905.
42. Sergiyenko A., Serhienko A., Romankevich V. Genetic Programming of Discrete Cosine Transform Processors. *6-th International Conference on High-Performance Computing (HPC-UA 2020).* Kyiv, 06-07 Nov. 2020, pp. 1-6.