UDC 004.383 https://doi.org/10.20535/2786-8729.4.2024.304965

METHOD FOR MAPPING CYCLO-DYNAMIC DATAFLOW INTO PIPELINED DATAPATH

Anatoliy Sergiyenko
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
ORCID: http://orcid.org/0000-0001-5965-1789

Ivan Mozghovyi
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
ORCID: http://orcid.org/0000-0001-5469-486X

An overview of high-level synthesis (HLS) systems for designing pipelined datapaths is presented in the paper. The goal is to explore methods of mapping algorithms to the pipelined datapaths implementing the cyclic data flow graphs with dynamic schedules. The proposed method involves describing cyclo-dynamic data flow graphs in VHDL and optimizing them. Through examples like sequence detector and LZW decompressor, the method is demonstrated to be effective and can be implemented in HLS tools for field programable gate arrays.

Key words: data flow graph, FPGA, VHDL, datapath, pipeline, dynamic schedule.

1. Introduction

The high-level synthesis (HLS) systems are increasingly distributed by companies producing CAD tools for integral circuit and FPGA design. They are intended, in general, to speed up the computer-aided design of hardware devices. The user using such a system can describe the parallel algorithm and map it automatically into hardware described at the register transfer level (RTL). Next, the RTL description is transformed into the gate-level description using the proper compiler-synthesizer. The use of HLS systems makes it possible to speed up the design process tenfold [1].

Among HLS compilers the industrial compilers like Intel Data Parallel C++ [2], AMD-Xilinx Vitis HLS [3], Cadence C-to-Silicon compler, named Stratus HLS [4], Calypto Catapult HLS [1], are well-known. The initial algorithms for them are written in C/C++ language. But this language is not intended for the parallel algorithm representation. To do this, the HLS user has to add special pragmas and functions in the program to express the parallelism explicitely. As a result, the designed project effectiveness strongly depends on the designer`s skills.

MathWorks HDL Coder provides mapping both Matlab codes and Simulink graphical algorithm representation into the pipelined datapath [5]. The Catapult HLS compiler is able to map the graphical representation as well. In both situations, the algorithm expressess its parallelism explicitely by the synchronous dataflow (SDF) graph. This helps to get more effective hardware solutions.

However, HLS still does not provide a decent minimization of hardware costs of synthesized pipelined datapaths compared to the manual design. So, the most of integral circuit or FPGA projects are performed using the traditional register transfer level (RTL) description. Therefore, it is necessary to search for new methods of mapping algorithms into hardware computing devices. Existing methods of mapping the data flow graphs of various types have found application in the program compilation, but they are rarely used in hardware design [6, 7].

In this paper, we examine the methods of pipelined datapath design that are already known. Through these methods of analysis, it is possible to select a method for designing pipelined datapaths based on cyclic data flow graphs with dynamic scheduling that focuses on algorithm execution. Using a VHDL language, a cyclo-dynamic data flow graph (CDDF) is mapped into the datapath accompanied by the finite state machines (FSMs).

2. Pipelined datapath design using dataflow graphs. A review

Typically, datapath design involves describing the algorithm with a dataflow graph (DFG) and control flow graph, and mapping them into hardware. In such a mapping, three steps are sequentially executed: resource selection, operation scheduling, and operation assignment. After that, FSM is designed along with the interconnection scheme drawing. In spite of this, the quality of the resulting device is strongly influenced by the efficiency of the various steps of synthesis, each of which is focused on a specific objective. So, the resource selection intends to minimize hardware and operation scheduling does the time of the algorithm execution.

A very popular FSM with datapath (FSMD) design method is based on this approach. It synthesizes both the datapath and FSM that controls it [8]. A large set of algorithms can be implemented using this method. However, it cannot be directly used for synthesis of pipelined datapath that has high throughput.

For the synthesis of pipelined computers, the method of mapping the synchronous data flow graphs (SDF) has become widespread [9]. Such a graph consists of operator nodes and directed edges connecting them. It is considered that the operator in the node is executed (fired) immediately as soon as data (tokens) appear at its inputs, and then it outputs the results in its output edges. An edge serves in SDF as a dataflow and has a buffer to store the data. Usually, this is a FIFO buffer. It is represented graphically by thick dashes across the edge, that correspond to register delays.

If all data in the dataflows are one-to-one correlated, the dataflow graph is a synchronous one. For example, the data have indices, which linearly depend on the iteration number. Therefore, the execution of the algorithm on SDF has a constant period during which each node consumes and generates the same number of tokens [10]. Because of this, SDF is classified as a statically scheduled dataflow graph (SSDF) [11].

In a single-rate or uniform SDF, inputs and outputs of nodes consume and produce the same number of tokens during the calculation period. This helps to map such an SDF into a pipelined datapath that executes an algorithm with a period of one cycle. This mapping is based on the following. The nodes correspond to the logical circuits that calculate operators. The edges and their register delays are mapped into the communication lines and pipeline registers respectively [12].

The representation of SDF in a multidimensional space makes it possible to formally design the pipelined datapaths with a given period of algorithm execution [13]. The SDF use is limited by the set of algorithms in which nodes execute the same operations in each period.

The cyclo-static dataflow (CSDF) is a more general model than SDF. It considers that the actors can have different numbers of executed tokens in different firings, but the amount of these tokens in a single cycle is stable. Therefore, such a model provides a static schedule [14].

The parametrized SDF (PSDF) is a more general and impressive model of the cyclic algorithms. It considers that the nodes can perform a set of different operations, which can be switched depending on the configuration of tokens in the node inputs. Moreover, the graph can be hierarchical, i.e., its node can be expanded as some SDF or FSM [15].

But CSDF, SSDF, and PSDF could not express the algorithms in which the operator execution is performed dynamically depending on the data. They are practically used only in the automatic programming but not in the hardware design [6, 15].

The DFG with the dynamical shedule, the dynamic dataflow (DDF) model could not get its schedule at compile time. The specific time period of the node firing is derived for it only during the algorithm execution for concrete input data set. In such an algorithm execution, the deadlock hazard can occur. However, the most of practical algorithms belongs to this cathegory. They are programmed and tested with the concrete data sets detecting the possible deadlocks. But they are implemented in hardware very rarely.

The core functional dataflow (CFDF) model of computation is a highly expressive DDF model with the properties of PSDF, in which the blockings of the node firings are removed using specific rules of such firings [16]. This model is concretized in the dataflow schedule graph (DSG). Such a graph represents the time-multiplexed execution of CFDF across a set of hardware resources [17]. However, DSG finds the effective implementation only in the multithreading programming [6].

There is a wide class of cyclic algorithms which could not be represented by SDF, CSDF, or SSDF. However, they are less expressive than CFDF algorithms. They distinguished in that the algorithm period depends on the data which it executes. For example, it is the compression algorithm, each output code calculation period is variable and strongly depends on the data. Therefore, such an algorithm has a dynamic schedule and the respective computation model is named cyclo-dynamic dataflow (CDDF) [18]. The hardware implementation of such an algorithm is performed usually by the FSMD method, and therefore, it is complex and often ineffective. Consider the design of a new method of mapping CDDF into the pipelined datapath.

3. The aim and objectives of the research

The aim of the research is a new method of mapping CDDF into the pipelined datapath. The objectives are mapping conditions formulation, graph node mapping details, graph optimization method selection, and the proposed method validation.

4. Method of mapping cyclo-dynamic dataflow

4.1. Prerequisites for the method creation

The method is intended for the design of pipelined datapaths on FPGAs. This means that the dataflow implementation and hardware execution of the FPGA architecture must be considered. And a CDDF model arrangement must ensure both correct hardware implementation and deadlock absence.

CDDF can be mapped into the pipelined datapath as well as homogeneous SDF can be. Such a mapping is possible when a set of conditions is satisfied. Firstly, the necessary conditions are that CDDF must be deterministic and free of deadlocks [18]. Each CDDF node is mapped into the respective logic scheme. Then, the dataflow represented by a weighted edge is mapped into the respective register chain or FIFO buffer. Finally, when the resulting structure is described by some hardware description language like VHDL the conditions of proper mapping into hardware must be satisfied.

So, the mapping conditions are the following.

1. The control token must have a limited number of values, which indicate dynamic behavior. It must be present in the same phase (iteration) where it determines which phase should be executed, and it must not be affected by the input datum value or index. [14].

2. The graph should not have cycles of dependencies without any delay in the edges [14]. Due to this condition, the corresponding data dependency graph has no dependency cycles and, therefore, provides structural solutions without blocking. As a result, there will be no loop in the combination circuit, which will result in an unintended latch [18].

3. All delays that load the edges must have the initial data or tokens; this condition is named the live cycle condition [19].

4. CDDF is a cycle-based model. Each cycle executes a different number of iterations. As in the homogeneous SDF [14, 19], each node must consume and generate tokens at the same rate each iteration.

5. When a cycle is completed, CDDF must return to its initial state. This ensures the period-safety of CDDF [19].

6. FSM which generates the control tokens must be determined one and none deadlock must be in it.

7. The logic scheme which is described in the VHDL language using the process operator, must use the IF-THEN-ELSE and CASE logical operators. The deadlock free condition is that the ELSE alternative and all alternative branches of the CASE statement are present. The similar features have the WHEN-ELSE and WITH-SELECT operators as well [20].

8. The dataflow is described in the VHDL language as a process that is triggered by the edge of a common clock signal, in which assignments are made to the signals that mark the FIFO registers [20].

CDDF is mapped into hardware as its description in VHDL along with the FSM descriptions. The pipelined datapath is described as a set of descriptions of the graph nodes. Each node along with the adjacent output edges are described by one process operator which is sensitive to the common

clock signal. However, a set of such process operators can be combined into a single process operator [21].

The CDDF elements have a symbolic representation, such as SDF elements. The examples of which are shown in Fig. 1. So, the counter is set using the incrementing node of and register delay as in Fig. 1, a. At the node inputs, the enable, initialization control data as well as the data stored in the register delay are inputted. A similar counter but with a separate output edge is shown in Fig. 1, b.

If the random access memory (RAM) is the mapping target, then the specific subgraph of CDDF is related to it. The Block RAM (BRAM), which is used in FPGA, has storage as the register set, writing data register, write and read address registers, and respective write address decoder and read data multiplexor. The corresponding subgraph has a node for data writing (MW), storage, and reading data node (MR) (Fig. 1, c). Nodes MW and MR have the data D and address A inputs. The storage itself is depicted by a long bar. The edge that passes through the storage bar is also thickened because it characterizes *n* data buses that are attached to *n* registers of the storage.

The subgraph of FSM consists of the node that forms the next state (NS) based on the input signal X, the output state (OS) node, which forms the output data Y, and the state register (ST) which loads the edge connecting both nodes (Fig. 1, d).

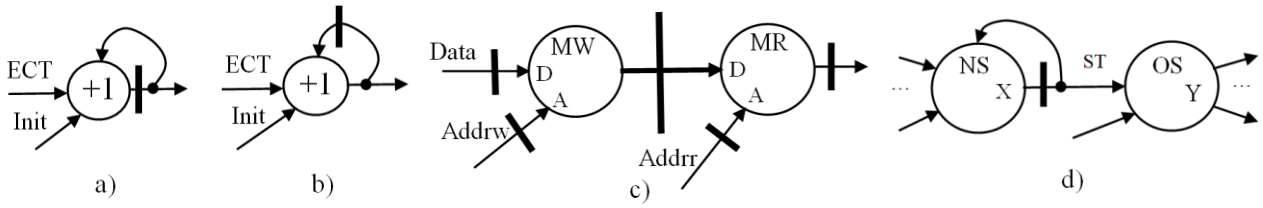

Fig. 1. Some CDDF element graphical representations: *a* – counter with output from the register, *b* – counter with output from the incrementor node, *c* – BRAM, *d* – FSM.

4.2. CDDF Optimization

When synthesizing the pipeline datapath, SDF is optimized by shortening the critical path in the corresponding datapath circuit, and then, optimized SDF is mapped to the datapath. The retiming procedure is recognized as a universal method of optimizing SDF, and it consists of such a permutation of delays in edges that does not disrupt the general execution of the algorithm. The pipelining method is most often used for SDFs, according to which the same number of delays is inserted into the edges, which are directed in the same direction relative to the graph intersection. The pipelining is similar to the retiming because it shortens the critical path as well. But at the same time, the latent delay of the algorithm increases [22]. Likewise, CDDF can be optimized using the retiming and pipelining methods.

4.3. Method of mapping CDDF into pipelined datapath

The method is intended for design of pipelined datapaths, which are configured in FPGAs and execute the algorithm using a period of one clock cycle. Initial data for design are:

– a dataflow algorithm that is represented as CDDF and that can use access to single or multi-port memory, which has the control part like FSM;

– effectiveness criterion t_C as the minimum period of the clock interval;

– library of FPGA elements of a certain series, which includes registers, adders, and BRAM with specific delays.

Design results are the device description in VHDL or Verilog, which is ready for synthesis and further configuration in FPGA.

1. Representation of the algorithm in the form of CDDF. The functions performed by the logic circuits are represented by the corresponding nodes. The data transfer between the nodes along with the corresponding delays for the required number of clock cycles are represented by edges loaded by the respective delays. The functions of storing data into BRAM and reading these data are

represented by the subgraph like the one in Fig. 1, c. The control FSM is represented by the subgraph like the one in Fig. 1, d. The derived CDDF must satisfy the necessary conditions to be deterministic and free of deadlocks mentioned above.

2. Optimization of CDDF using pipelining and retiming. The goal of optimization is to minimize the value t_C of the maximum clock signal period. It is approximated by the critical path in CDDF.

3. Mapping optimized CDDF to the hardware. At the same time, nodes with outputted edges incident to them are described by VHDL language process operators or Verilog language Always constructs. The FSM is described as a separate process. The resulting VHDL or Verilog program is a description of the functional scheme at the level of register transfers of a pipeline computer that executes a given algorithm with a period of one cycle, which is minimized in terms of duration.

The following should be taken into account during the optimization of CDDF. The critical path t_C in CDDF is the maximum path delay. It is determined as a sum of delays in the logic circuits which are relevant to the nodes that belong to the path between two edges loaded by registered delays:

$$t_C = \max \sum_i t_{Pi}, \quad (1)$$

where t_{Pi} is the delay of the i-th node of the P-th type which belongs to the considered path. It should also be noted that in modern FPGAs, the share of delay in the interconnection lines reaches 60-90%, and the delay in logic circuits accounts for 10-40% of the total delay, respectively. Therefore, the final decision to obtain an optimized project should be made after processing the VHDL file with a compiler-synthesizer, placer, and router of the FPGA CAD tool which calculate the formula (1) automatically.

5. Experimental results

5.1. Sequence detector synthesis

A sequence detector synthesis is excellent example of CDDF mapping into application specific hardware because it is both simple and showing all the method features. Such a detector is FSM that is designed to determine whether an input string of characters corresponds to a given grammar of a particular language or not. In fact, such an automaton performs parsing of input character sequences. The software implementation of such FSM is a traditional step in the development of compilers, protocol implementation systems, Web applications, etc. Its hardware implementation makes it possible to significantly speed up detection, reduce power consumption and is typical in specialized devices.

The characters $\alpha_i \in T$ of the input words $A = \alpha_1\alpha_{2\ldots}\alpha_n$ from the input stream belonging to the alphabet of terminal symbols of the language L are sequentially fed to FSM input. Among the symbols in the stream are empty symbols $\varepsilon \in T$, which separate words from each other and are present in every alphabet implicitly.

When FSM is started or when there is a symbol ε at the input, it goes to the initial state S. As soon as the first symbol $\alpha_1 \neq \varepsilon$, appears at the input of the automaton, it goes to the next state corresponding to the symbol α_1 in the expected word. With each new symbol α_i in the input stream, which belongs to a word of language L, FSM moves to the next correct state. Otherwise, it goes into error state E, which indicates that the word being analyzed does not belong to language L.

If FSM is in the state it entered after analyzing the last symbol α_n in the word, and the next symbol is ε or a special symbol at the end of the word, then FSM goes to the final state that corresponds to the recognized word A. If there are several words in the language L, then several final states are possible.

In the operation of FSM described above, it is assumed that it receives input symbols from some input data stream. In its hardware implementation, the FIFO buffer plays the role of a stream as a means of data transfer between computing processes.

The FIFO buffer together with FSM of the regular expression in the form of CDDF is shown in Fig. 2. The FIFO buffer is usually implemented using a cyclic buffer scheme based on RAM and write pointer pw and read pointer pr. The subgraph corresponding to the FIFO buffer has a data write edge (MW), a BUF buffer, and a data read edge (MR). Edges MW, MR have data input DI and address input A. The edges of the increment I1, I2 perform the increment of the states of the pointer registers for writing pw and reading pr.

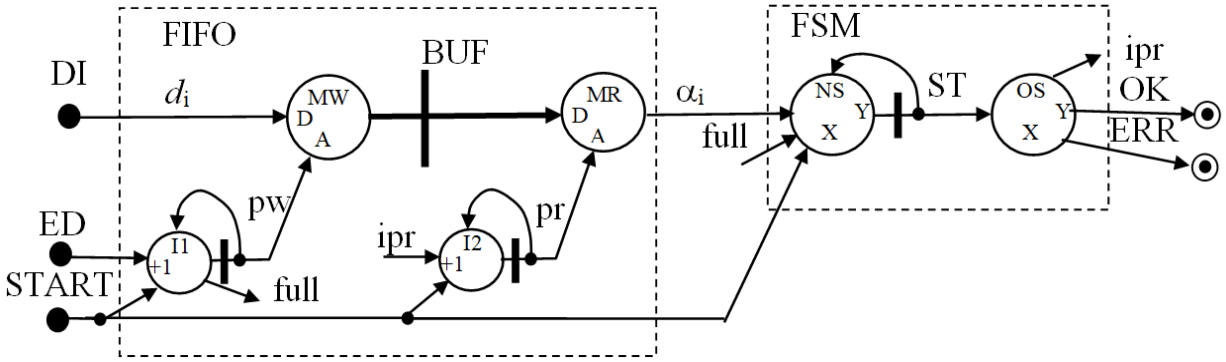

Fig. 2. CDDF of the sequence detector

After the START input signal, the pointers pw, pr and the status register ST are set to the initial state. The input stream of symbols *d* is fed to the DI input and the symbols are written to the BUF memory by the enable signal ED, which also increments the write address pw. When the FIFO is full (for example, the pw pointer reaches half of the BUF memory), the full signal is generated, which starts the operation of FSM.

A subgraph represented by FSM consists of a node that forms a next state (NS) based on its inputs X, an output control signal (OS) node that forms its output Y, and a state register (ST) that loads the edge, which connects both nodes. The FSM outputs are the ipr signal of the read pointer increment, the OK device output, signaling that a word has been found, and the ERR output, which indicates an incorrect word.

Consider a sequence detector that recognizes the lines START and STOP. This detector must match the following regular expression grammar:

$$G = START + STOP = ST(ART + OP). \quad (2)$$

The corresponding FSM diagram is shown in Fig. 3. Unlike a typical FSM diagram of the sequence detectors which are synthesized according to the grammar (2) the state node S1 is added to this diagram. FSM enters the state *S*1 when the START signal arrives, and state *S* when the FIFO input buffer is full. In addition, in the *S* state, FSM waits for the characters of the words, and FSM enters this state after receiving each word.

One can see that the detector executes the periodical algorithm in which the inner loop (a word recogniton) has the number of cycles depending on input data and the outer loop (word flow) which is stable. So, this example represents an example of CDDF use. It should be noted that each inner loop starts with the same FSM state *S*. The respective control token full, which is generated by the node I1, has two values: true and false. According to the algorithm, the input data are entering periodically and therefore, the deadlock with this token must not occur. Therefore, the conditions that the control token must have a limited number of values, and that it must be present in the same iteration where it determines which phase should be executed, and it must not be affected by the

input datum value or index are satisfied. As a result, both CDDF in Fig.2 and FSM diagram in Fig. 3 satisfy all the mapping conditions described in Section 3.

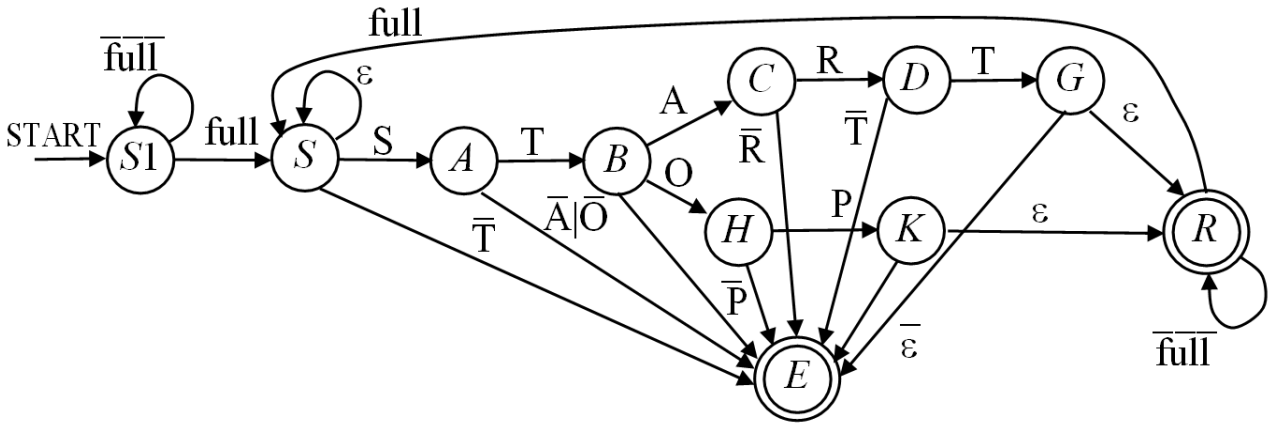

Fig.3. FSM diagram of the detector detecting the words START and STOP

The CDDF in Fig. 2 accompanied by FSM diagram in Fig.3 are described by the VHDL program shown below.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.Numeric_std.all;
entity DETECT1 is
        port(   CLK :    in STD_LOGIC;                  -- Clock
                RST :    in STD_LOGIC;                  -- Reset
                START:   in STD_LOGIC;                  -- input data starting
                ED :     in STD_LOGIC;                   -- data enable
                DI :     in STD_LOGIC_VECTOR(7 downto 0); -- input data
                STRT: out STD_LOGIC;                    -- START detected
                STOP: out STD_LOGIC;                    -- STOP detected
                OK :   out STD_LOGIC;           -- is detected
                ERR :  out STD_LOGIC);                  -- error in input data
end DETECT1;
architecture beh of DETECT1 is
    type type8 is array (0 to 15) of unsigned(7 downto 0);--buffer FIFO type
        signal BUF: type8:=(others=>x"00");   -- buffer FIFO type of the length 16
        type state is (S, S1,A,B,C,D,E,G,H,K,R);  -- FSM states
        signal ST: state;
        signal pw, pr:unsigned(3 downto 0);    -- write and read pointer
        signal full, ipr: std_logic;
        signal symb:unsigned(7 downto 0);              -- FIFO output symbol
begin
        FIFO:process(CLK,RST, BUF,pw,pr) begin       -- FIFO buffer
                if RST='1' then
                        pw<="0000";    pr <="0000";
                elsif Rising_edge(CLK) then
                        if ED='1' then
                                pw<=pw+1;              -- write pointer increment
                                BUF(to_integer(pw))<=unsigned(DI);--input datum
                        end if;
                        if ipr='1' then
                                pr<=pr+1;              --read pointer increment
                        end if;
                end if;
                symb<= BUF(to_integer(pr));            -- FIFO outputs a symbol
```

```
        if pw - pr >= 8 then                    -- condition that FIFO is full
                full<='1';
        else
                full<='0';
        end if;
end process;

FSM:process(CLK,RST) begin
        if RST='1' then
                ST<= S1; STRT<='0';  STOP<='0';
        elsif Rising_edge(CLK) then
                if START='1' then
                        ST<=S1;  -- state register
                end if;
                case ST is
                        when S1=>if full = '1' then
                                        ST<= S;
                                end if;
                        when S=>
                                        STRT<='0'; STOP<='0';
                                if symb = 0 then
                                        ST<=S;
                                elsif symb = character'pos('S') then
                                        ST<= A;
                                else
                                        ST<= E;
                                end if;
                        when A=>if symb = character'pos('T') then
                                        ST<= B;
                                else
                                        ST<= E;
                                end if;
                        when B=>if symb = character'pos('A') then
                                        ST<= C;
                                elsif symb = character'pos('O') then
                                        ST<= H;
                                else
                                        ST<= E;
                                end if;
                        when C=>if symb = character'pos('R') then
                                        ST<= D;
                                else
                                        ST<= E;
                                end if;
                        when D=>if symb = character'pos('T') then
                                        ST<= G;
                                else
                                        ST<= E;
                                end if;
                        when G =>if symb = 0 then
                                ST<= R;  STRT<='1';         -- START flag
                                end if;
                        when H=>if symb = character'pos('P') then
                                        ST<= K;
                                else
                                        ST<= E;
                                end if;
                        when  K =>if symb = 0 then
                                ST<= R;  STOP<='1';         -- STOP flag
                                end if;
```

```
                        when R =>if   full = '1' then
                                      ST<= S;
                                end if;
                        when others=> null;
                    end case;
            end if;
        end process;
        ipr<='1' when ST=S or ST=A or ST=B or ST=C -- increment of reading pointer of FIFO
                    or ST=D or ST=G or ST=H or ST=K     else '0';
        OK<='1'   when ST=R else '0';        -- output flags
        ERR<='1' when ST=E else '0';
end beh;
```

In this program, the FIFO process describes the input buffer of symbols (four nodes of CDDF in Fig. 2), and the FSM process is a finite state machine (two other nodes in CDDF). The bars which load the CDDF edges are mapped in the respective models of the registers: pw, pr. FSM actually detects sequences. In the case ST operator, the FSM behavior is programmed, the graph of which is shown in Fig. 3. The ST signal represents the state register of this FSM. The signal assignment of this register is controlled by the long operator case ST. In each alternative when X of this operator, a transition from a given node-state X to another node is coded (Fig. 3) depending on the signals, which come from the FIFO buffer (Fig. 2). At the same time, an input symbol is compared with the given one: symb = character'pos('S').

Table 1 shows the hardware volume of the sequence detector in number of look-up tables (LUTs), flip-flops (FFs), and configurable logic blocks (CLBs) when it is configured in AMD-Xlinx Kintex-7 FPGA. This table shows the maximum clock frequency as well.

Table 1

Results of the sequence detector synthesis

Hardware volume			**Maximum clock frequency, MHz**
LUTs	**FFs**	**CLBs**	
49	14	20	446

The given results show us that the clock frequency is very high and the hardware volume is small, so detector synthesis and the proposed method are effective.

5.2. LZW decompressor syntheses

The LZW loss-less compression algorithm is distinguished in that its decompression turns out to be much simpler than the compression is. Because of this, the LZW algorithm is common in systems where decompression occurs more often than compression, for example, when decompressing GIF files. Therefore, this algorithm is widely used in many applications where the decompression is very frequently used, for example, in WEB applications. The paper [23] illustrates both the LZW algorithm and its execution in a microprocessor with a specialized architecture by hardware-software approach in detail. However, the full hardware implementation of this algorithm provides much higher throughput.

Consider an example of the development of a pipelined module that performs decompression in the LZW algorithm which is more sophisticated than the sequence detector algorithm. The LZW algorithm is represented by CDDF because, for each input code, it has to get from the dictionary the output symbols in sequence. The length of this sequence is variable depending on the input code.

The synthesis of the LZW decompressor using the method of CDDF mapping is described in [24] in detail. The optimized CDDF of the LZW decompressor takes 25 nodes and 35 register delays. It contains 2 FSM and 3 FIFO buffer subgraphs like ones in Fig.1 c, d.

The results of the synthesis of the decompressor for AMD-Xilinx FPGAs using the proposed method are presented in Table 2. There, the result of the synthesis of the similar device that performs

the same algorithm is given for comparison. This device is built using the traditional method of design and invention of its authors.

The proposed design is implemented in Xilinx Kintex-7 FPGA, and the analogous device is in Xilinx Virtex-7 series. It should be taken into account that the FPGAs of the Virtex-7 series have slightly better timing characteristics than Kintex-7, however, they are manufactured using the same architecture and technology.

Table 2.
Features of the LZW decompressors configured in AMD-Xilinx FPGA

Decompressor	**Hardware volume**			**Maximum clock frequency, MHz**	**Throughut, MB/s**
	LUTs	**FFs**	**18k BRAMs**		
Proposed	154	162	7	360	205
Kagawa [25]	213	224	13	296	295

Analysis of Table II shows that the decompressor described in [25] has the highest throughput due to excessive built-in RAM block (BRAM) memory volumes. The decompressor developed according to the proposed method has a 21% higher clock frequency and 28% lower hardware costs, as well as a 46% smaller memory volume compared to the device [25], which has the same dictionary volume, and bit sizes of input and output data. It should be noted that BRAM memory is a valuable resource, because one block of 18 kilobit BRAM in an AMD-Xilinx FPGA, on average, accounts for 800 LUTs. Therefore, the developed decompressor is certainly profitable in terms of hardware costs. The received throughput achieves 205 decompressed megabytes per second with an average compression ratio of two times.

So, the use of the proposed method simplifies the design process of the pipelined datapaths, compared to the most popular method of FSMD design. It optimizes the project both in clock frequency and in hardware volume, providing the effective utilization of the specific blocks of FPGA like BRAM.

6. Discussion

Analysis of Table II shows that the decompressor described in [25] has the highest throughput due to excessive built-in RAM block (BRAM) memory volumes. The decompressor developed according to the proposed method has a 21% higher clock frequency and 28% lower hardware costs, as well as a 46% smaller memory volume compared to the device [25], which has the same dictionary volume, and bit sizes of input and output data. It should be noted that BRAM memory is a valuable resource, because one block of 18 kilobit BRAM in an AMD-Xilinx FPGA, on average, accounts for 800 LUTs. Therefore, the developed decompressor is certainly profitable in terms of hardware costs. The received throughput achieves 205 decompressed megabytes per second with an average compression ratio of two times.

So, the use of the proposed method simplifies the design process of the pipelined datapaths, compared to the most popular method of FSMD design. It optimizes the project both in clock frequency and in hardware volume, providing the effective utilization of the specific blocks of FPGA like BRAM.

7. Conclusion

The graph models of data flow processing algorithms are analyzed and a class of cyclo-dynamic data flow graphs is selected. A method of designing the pipelined datapath that performs cyclic algorithms with a dynamic schedule is proposed. The results of designing a sequence detector and a lossless decompression device using this method are given. The method can be used manually as well as be implemented in the HLS systems. The next investigation steps will be devoted to probing this method for more complex algorithm mapping into FPGA and to designing an automatic framework for the pipelined datapath development.

References

[1] *Calypto's Catapult 8 HLS: C-Based Hardware Design Matures.* BDTI Resources for Engineers. December 17, 2014. https://www.bdti.com/InsideDSP/2014/11/18/Calypto.

[2] *Intel oneAPI DPC++/C++ Compiler Handbook for Intel FPGAs*. Intel Corporation. 05.08.2024. https://cdrdv2.intel.com/v1/dl/getContent/805578?fileName=oneapi-fpga-add-on_developer-guide_2024.1-785441-805578.pdf

[3] *Vitis High-Level Synthesis User Guide (UG1399).* AMD Adaptive Computing. 18.12.2023. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls

[4] *C-to-Silicon Compiler High-Level Synthesis Automated high-level synthesis for design and verification*. Cadence. 2008. 4 p. http://pdf2.solecsy.com/564/5c0644f6-808c-4d18-9b31-0eec054873e5.pdf

[5] K. Kintali, Y. Gu, "Model-Based Design with Simulink, HDL Coder, and Xilinx System Generator for DSP," *MathWorks. White paper*. 2017. pp. 1–15. https://de.mathworks.com/content/ dam/mathworks/tag-team/Objects/x/86457_92077_v01_ Xilinx_WhitePaper.pdf

[6] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, *Software Synthesis from Dataflow Graphs,* Springer US. 2012.

[7] K. Lee, Y. Lee, A. Raina, et al. "Software synthesis from dataflow schedule graphs." *SN Appl. Sci.* No. 3, Vol. 142. 2021. https://doi.org/10.1007/s42452-020-04135-6

[8] D. D. Gajski, S. Abdi, A. Gerstlauer, G. Schirner, *Embedded System Design. Modeling, Synthesis and Verification*. Springer. 2009.

[9] P. Schaumont, *A Practical Introduction to Hardware/Software Codesign*. Springer. 2011.

[10] E. A. Lee, D. G. Messerschmitt, "Synchronous data flow." *Proceedings of the IEEE,* vol. 75, no. 9, pp. 1235–1245, Sept. 1987, DOI: https://doi.org/10.1109/PROC.1987.13876

[11] E. A. Lee, S. Neuendorffer, "Concurrent models of computation for embedded software". *IEE-INST ELEC ENG. IEE Proceedings – Computers and Digital Techniques*, vol. 152. no. 2, 2005, pp. 239–250. https://doi.org/10.1049/ip-cdt:20045065

[12] S. A. Khan, *Digital Design of Signal Processing Systems. A Practical Approach*. UK: Wiley. 2011.

[13] A. Sergiyenko, A. Serhienko, A. Simonenko, "A method for synchronous dataflow retiming". *2017 IEEE First Ukraine Conference on Electrical and Computer Engineering (UKRCON)*, Kyiv, Ukraine, april 2017, 2017. pp. 1015–1018, https://doi.org/10.1109/UKRCON.2017.8100404

[14] T. M. Parks, J. L. Pino, E. A. Lee, "A comparison of synchronous and cycle-static dataflow". *29th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, USA, vol. 1, 1995. pp. 204–210 https://doi.org/10.1109/ACSSC.1995.540541

[15] B. Bhattacharyya, S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems". *IEEE Transactions on Signal Processing,* vol. 49, no. 10, 2001, pp. 2408–2421. https://doi.org/10.1109/78.950795

[16] W. Plishker, N. Sane, M. Kiemb, S.S. Bhattacharyya, (2008). "Heterogeneous Design in Functional DIF". In: Bereković, M., Dimopoulos, N., Wong, S. (eds*) Embedded Computer Systems: Architectures, Modeling, and Simulation. SAMOS 2008.* Lecture Notes in Computer Science, vol. 5114. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-70550-5_18.

[17] H. -H. Wu, C. -C. Shen, N. Sane, W. Plishker and S. S. Bhattacharyya, "A Model-Based Schedule Representation for Heterogeneous Mapping of Dataflow Graphs," *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, Anchorage, AK, USA, 2011, pp. 70-81, doi: 10.1109/IPDPS.2011.128.

[18] P. Wauters, M. Engels, R. Lauwereins, J. A. Peperstraete, "Cyclo-dynamic dataflow". *Proc. of 4th Euromicro Workshop on Parallel and Distributed Processing*, Braga, Portugal, 1996, pp. 319–326, https://doi.org/10.1109/EMPDP.1996.500603

[19] P. Fradet, A. Girault, P. Poplavko, "SPDF: A schedulable parametric data-flow MoC*"*. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, 2012, pp. 769–774, https://doi.org/10.1109/DATE.2012.6176572.

[20] M. Keating, P. Brikaud, *Reuse Methodology Manual for System-on-a-Chip Designs*, 3d Ed. Kluwer. 2007.
[21] A. M. Sergiyenko, *VHDL dlya projectirovanija vychislitelnych ustroystv*. Kyiv: Diasoft. 2004. (In Russian).
[22] R. Woods, J. McAllister, G. Lightbody, Y. Yi, *FPGA-based Implementation of Signal Processing Systems*. Wiley, 2d Ed. 2017, 447 p.
[23] V. O. Romankevych, I. V. Mozghovyi, P. A. Serhiienko L. Zacharioudakis, "Decompressor for hardware applications". *Applied Aspects of Information Technology.* 2023. Vol.6, no.1. P. 74–83. https://doi.org/10.15276/aait.06.2023.6
[24] A.M. Sergiyenko, I.V. Mozghovyi: "Hardware decompressor design". *Electron. Model*. 2023, vol. 45. no. 5, pp.113-128. https://doi.org/10.15407/emodel.45.05.113
[25] H. Kagawa, Y. Ito, K. Nakano, "Throughput-Optimal Hardware Implementation of LZW Decompression on the FPGA". 2019 *Seventh International Symposium on Computing and Networking Workshops* (CANDARW), Nagasaki, Japan, 2019. pp. 78–83. https://doi.org/10.1109/CANDARW.2019.00022.

УДК 004.383 https://doi.org/10.20535/2786-8729.4.2024.304965

МЕТОД ВІДОБРАЖЕННЯ ГРАФА ЦИКЛОДИНАМІЧНИХ ПОТОКІВ ДАНИХ У КОНВЕЄРНИЙ ОБЧИСЛЮВАЧ

Анатолій Сергієнко
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна
ORCID: http://orcid.org/0000-0001-5965-1789

Іван Мозговий
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна
ORCID: http://orcid.org/0000-0001-5469-486X

У статті представлено огляд систем високорівневого синтезу для проектування конвеєрних обчилювачів. Мета полягає в дослідженні методів відображення алгоритмів у конвеєрні обчилювачі, що реалізують циклічні алгоритми з графами потоків даних із динамічним розкладом. Граф циклодинамічних потоків даних (ГЦДПД) вибрано як виразну модель для опису широкої області алгоритмів обробки потоків даних. ГЦДПД відрізняється тим, що період алгоритму залежить від обчислених даних і має динамічний розклад. Сформульовано набір умов відображення, які забезпечують розклад ГЦДПД без взаємоблокувань, коли його відображають у конвеєрний обчислювач. Згідно з запропонованому методу, алгоритм представляється ГЦДПД і керуючими автоматами. Причому останні є підграфами ГЦДПД. ГЦДПД оптимізується за допомогою методів ресинхронізації та конвеєризації. Після цього ГЦДПД та його керуючі автомати описуються мовою опису обладнання, наприклад VHDL, так само, як описується граф синхронних потоків даних при синтезі конвеєрного обчислювача. Запропонований метод включає оптимізацію ГЦДПД та його опис на VHDL для реалізації в програмованих логічних інтегральних схемах. На прикладі проєктування детектора послідовностей детально показано реалізацію методу. Більш складне відображення алгоритму декомпресії LZW демонструє, що запропонований метод досить ефективний і дає в результаті синтезу конвеєрний обчислювач, ефективність якого порівнянна з ефективністю найкращого відомого апаратного рішення. Метод може бути реалізований в сучасних системах високорівневого синтезу.

Ключові слова: граф потоків даних, програмована логічна інтегральна схема, VHDL, конвеєрний обчислювач, динамічний розклад.