www.czasopisma.pan.pl PAN www.journals.pan.pl
INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2014, VOL. 60, NO. 1, PP. 41–46
Manuscript received January 10, 2014; revised March, 2014. DOI: 10.2478/eletel-2014-0004

About Implementation of IEC 61131-3 IL Function Blocks in Standard Microcontrollers

Mirosław Chmiel, Jan Mocha, Edward Hrynkiewicz, and Dariusz Polok

***Abstract*—The paper presents considerations on implementation of function blocks of the IL language, as fragments of control programs that use these blocks. Subsequently, the predefined function blocks of the IL language have been applied to implementation in a Central Processing Unit for a programmable controller based on standard microcontroller from such families as MCS-51, AVR and ARM with the Cortex-M3 core. The considerations refer to the IL language revision that is fully compliant with the IEC-61131-3 standards. The completed theoretical analysis demonstrated that the adopted method of the module description is really reasonable and offers substantial advantages as compared to direct calls of function modules already developed as subroutines. Also the executed experiments have proved the feasibility to arrange central units of programmable controllers on the basis of standard microcontrollers and such central units may be competitive to compact CPUs available on the market for typical PLCs.**

***Keywords*—Central Processing Unit, Programmable Logic Controller, microprocessor control, microprogramming, programming languages, language operators**

I. Introduction

IN total, the IEC-61131-3 standard defines five programming languages. Two of them are text languages (Instruction List – IL and Structured Text – ST) whilst three of them are graphic ones (Ladder Diagram – LD, Function Block Diagram – FBD and Sequential Function Chart – SFC) [1]–[3]. The IL language can be deemed as the assembler for PLCs since in majority of programming environments any control program developed initially in other programming languages, both text and graphic ones, is finally converted to the form of an instruction sequence. In addition, the program originally set up in IL language usually runs at highest rates. These factors served as the reasons that the CPU considered in this paper uses the language classified to the family of the IL languages.

This paper is continuation of considerations presented in the study [4] and presents the notation method that makes it possible to describe complex components of the IL language, such as counters, timers and other function blocks, by means of simple operators of the IL language. However, it proved necessary to use several additional operators that are not defined by the mentioned standard. Simple instructions of the IL language have been implemented as subroutines in the C language available for standard microcontrollers, which was already presented in the previous work [4]. In subsequent works complex instructions were described in the same way and were implemented to standard microcontrollers too. Results from the implementation are outlined in the final part of this study.

M. Chmiel, J. Mocha, E. Hrynkiewicz, and D. Polok are with the Silesian University of Technology, Institute of Electronics, Gliwice, Poland (e-mails: {Miroslaw.Chmiel; Jan.Mocha; Edward.Hrynkiewicz; Dariusz.Polok}@polsl.pl).

The objective of the paper is to demonstrate that a CPU executing control program prepared according to the IEC 61131-3 standard can be also implemented with use of microcontrollers of any generation, both older design and more recent ones. Such a solution may prove cost-effective since more expensive CPUs offered by manufacturers of programmable controllers (PLCs) can be substituted with cheaper solutions based on microcontrollers or microprocessors.

The paper is structured in the way that Section 2 outlines basic ideas that guided development of the paper and Section 3 explains how the notation method applied to the instruction lists affects CPU structures of programmable controllers. Section 4 recalls some information from the IEC 61131-3 standard and presents the basic modules that must be implemented within CPU structures. Incorporation of large modules, i.e. functional blocks is revealed in Section 5. Experiments carried by authors with achieved results are described in Section 6, whilst Section 7 comprises recapitulation of the paper and some conclusions for future efforts.

II. Basic Ideas

A Central Processing Unit (CPU) of a PLC can be designed in many ways whilst the CPU architecture is the key factor that is crucial for execution time of a control routine. The simplest way for implementation of a CPU is the use of a microcontroller. It can be a standard microprocessor, a dedicated microprocessor [5] or a standard microcontroller, for instance the CPU of the S7-200 PLC employs a microcontroller of MCS-51 family [6]. On the other hand, CPUs are frequently designed as multiprocessor systems that are made up of two units operating with data of bit and word formats or they incorporate a hardware coprocessor that assists execution of specific types of operations [7]. Application of FPGA units makes it possible to design dedicated microprocessor systems that are able to efficiently execute control program on the basis of a specialized microprocessor. Programmable devices enable also switching over from sequential-cyclic approach to hardware and parallel implementation within the resources of programmable circuit [8], [9].

The methods for translation of instructions of the IL language to the form of a subroutine executable by standard microcontrollers are outlined in the paper [4].

For the needs of this study some popular standard microcontrollers were investigated to make a comparison between various solutions:

- 8052 (AT89s52) of the MCS-51 family;
- ATMega16 of the AVR family;
- STM32F103RB from STM32 with the ARM Cortex-M3.

The common architecture of the investigated CPU was slightly modified during experiments to take account of the

same differences in internal structures of the applied microcontroller. Translation of instructions in the IL language was carried out in the form of program fragments and procedures of the C language [4].

III. Influence of the Notation Method for an IL onto the CPUStructure

The description, how key operators of the IL language can be implemented by means of C-language subroutines, is presented in the study, whilst this text, as it has been already mentioned, deals with issues that are commonly encountered at implementation of function blocks defined according to the IEC 61131-3 standard.

The analyze of the requirements imposed by the mentioned standard demonstrates that individual operators allow using only a single argument, which entails the need to assign a dedicated memory cell to store the current result (CR) of the operation. The mechanism for execution of individual tasks should be construed as:

CR = CR ***OPERATOR*** OPERAND

According to the formulation of the standard the CPU accumulator must be of universal nature, but the authors believe that such a solution has a substantial drawback, since it is impossible to maintain, within a single routine, a thread associated with operation on Boolean arguments in parallel to the operations on word-type arguments, which may occur when various fragments of the routine can be executed optionally, depending whether the Boolean condition is fulfilled or not, whilst the main body of the routine comprises arithmetical computations. The same takes place upon the attempt to design a combined, Bit-Byte CPU [7]. By the foregoing reason as well as due to the fact that individual commands of the IL language must be expanded as subroutines in the C language, the list of operators in the IL language shall comprise pairs of operators capable of handling those two types of arguments, e.g. LD_b for bit-type and LD_W for word-type operands.

Figure 1 presents the logic structure of a CPU and explains how internal components of a microcontroller can be applied to implementation of the suggested central unit (CPU) for a PLC controller with two accumulators:

- CR_b0 – one-bit for execution of operations on bit type variables;
- CR_W0 – 16-bit for execution of operations on integer type variables.

The CPU structure comprises also stacks of accumulators that are necessary to execute operations in brackets. Due to the reasons similar to the aforementioned ones, there are two separate stacks and two special elements: OV – an overflow flip-flop for arithmetic operations, and ACCT – a 32-bit register for timer operation [4].

IV. Function Modules Defined by the IEC 61131-3 Standard

The list of standard function blocks and their parameters is shown in Tab. I.

Standard function blocks, in contrary to typical operators, are components of sequential execution. Therefore, correct

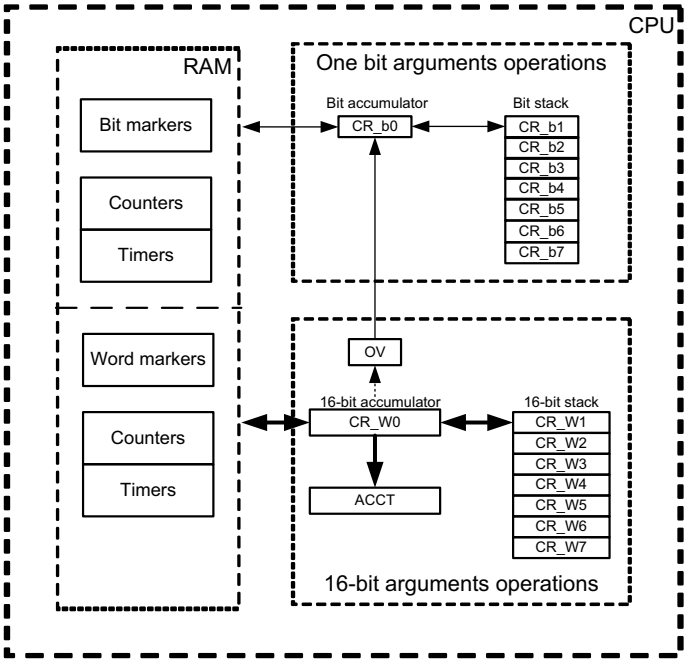

Fig. 1. Data flow in the Central Processing Unit.

TABLE I
Standard Function Blocks According to IEC 61131-3 [2]

Name	Inputs	Outputs	Description
Bistable elements			
SR	S1,R	Q1	Set dominant
RS	R,S1	Q1	Reset dominant
Edge detections			
R_TRIG	CLK	Q	Rising edge
F_TRIG	CLK	Q	Falling edge
Counters			
CTU	CU,R,PV	Q,CV	Up counter
CTD	CU,LD,PV	Q,CV	Down counter
CTUD	CU,CD,R,LD,PV	QU,QD,CV	Up/down counter
Timers			
TP	IN,PT	Q,ET	Pulse
TON	IN,PT	Q,ET	On-delay
TOF	IN,PT	Q,ET	Off-delay

execution of them in the simplest form needs a component referred to as the status memory. However the authors, upon analyzing statements of the standard as well as implementation of similar functionalities within the existing PLCs [5], [10], came to the conclusion that structuring of such functionalities into autonomous function blocks (subroutines) rather makes their implementation more difficult on defined hardware platforms. Consequently, execution of such subroutines takes more CPU time and, in particular in the IL language, makes more difficult to apply them to a control program, which will be evidenced in the further part of the manuscript.

A. Bistable Elements (Flip-Flops)

When it comes to the IL language, the required sequence of commands should be made up in the following way:

SR:
Q1 := S1 or (not R and Q1)
RS:
Q1 := not R1 and (S or Q1)

Please notice that execution of the functions for each of these components is reduced to setting up a simple routine that will look like in the following way in the IL language:

```
LD_b   S1 ;S1 -> CR_b0
PUSH_b    ;CR_b0 -> CR_b1
LDN_b  R  ;not R -> CR_b0
AND_b  Q1 ;CR_b0 OR Q1 -> CR_b0
OR()      ;CR_b1 OR CR_b0 -> CR_b0
ST_b   Q1 ;CR_b0 -> Q1
```

However, the IL language comprises also operators that enable execution of the foregoing fragment in another way:

```
LD_b R  ;R -> CR_b0
R    Q1 ;Reset Q1 if CR_b0=1
LD_b S1 ;S1 -> CR_b0
S    Q1 ;Set Q1 if CR_b0=1
```

It is the way that makes the notation even simpler and leads to savings on the execution time [11]. Please also pay attention that the second implementation of the flip-flop function is irrelevant to the preceding state and the current state will depend only on the input state.

B. Edge Detections

The mentioned standard defines two function blocks with the aim to enable detection of such situation in the control routine when a binary variable switch to the opposite state. The R_TRIG function evaluates the CLK signal and produces a “1” at the Q output when there is a rising edge (0 to 1 transition). This operation uses an internal edge detection flag – MEM – that stores the “previous” (from the previous call/execute of the R_TRIG operation) value of CLK in order to compare the current value against the flag and carry on operation with the current value. F_TRIG function detects the falling edge of CLK signal. The ST language representation of the positive edge detection operation is presented below:

R_TRIG:
Q := CLK and not MEM
MEM := CLK
F_TRIG:
Q := not CLK and not MEM
MEM := not CLK

Execution of the foregoing routines needs no additional operators and they can be noted with use of basic operators from the IL language, which is demonstrated on the example of the R_TRIG function:

```
LD_b   CLK ;CLK-> CR_b0
ANDN_b MEM ;CR_b0 AND not MEM -> CR_b0
ST_b   Q   ;CR_b0 -> Q
LD_b   CLK ;CLK -> CR_b0
ST_b   MEM ;CR_b0 -> MEM
```

However, there is no obstruction to create a new operator that shall use an element of the controller memory, for instance a marker, as its argument. In such a case the routine in the IL language will look like as follows:

```
LD_b   CLK ;CLK -> CR_b0, CR_b0'
R_TRIG MEM ;CR_b0 AND not MEM -> CR_b0
           ;CR_b0' -> MEM
ST_b   Q   ;CR_b0 -> Q
```

It is the method that needs a variable that stores, for the time of the routine execution, the current state of the bit accumulator – CR_b0. This requirement can be accomplished in many ways, for instance after each operation with LD_b (actually after each operation when the operation results is moved to CR_b) the variable value is stored in two memory cells – to the bit accumulator CR_b0 and to a memory cell CR_b0’ that mirrors the bit accumulator, however, these cells are not involved in execution of the instructions in question. Upon detection of an edge it is sufficient to store the content of that cell into the MEM cell. Use if the foregoing mechanism leads merely to the situation that operators for edge detection execute two functions as explained above. On the other hand, below is the problem solution that needs no modification to execution of the LD_b instruction.

```
LD_b    CLK ;CLK -> CR_b0
R_TRIG  MEM ;CR_b0 -> CR_b0'
            ;CR_b0 AND not MEM -> CR_b0
            ;CR_b0' -> MEM
ST_b    Q   ;CR_b0 -> Q
```

Please note that the two last solutions for the problem of edge detection do not need to have the last command executed at all since the fact of edge detection is most frequently used in the routine immediately downstream the location where the edge is detected. For the foregoing examples the information about edge detection is stored in the CR_b0 bit-accumulator.

C. Counters

The statements of the referred standard define three types of counters (for simplification the counters shall be described in the way that refers to imaging of the counters in one of graphic languages):

- CTU – the counter that counts up pulses supplied to the CU (Count Up) input that is sensitive to rising edge. The counter has the R (Reset) input that enables clearing of the counter content to zero as well as the register PV (Preset Value). If the current content of the counter equals to the value at the PV input the binary output Q adopts the active status (1);
- CTD – the counter that counts down pulses supplied to the CD (Count Down) input that is sensitive to rising

edge. The counter has the LD (Load) input that enables loading of values from the PV inputs to the CV cell. When current content of the counter comes to zero (0), the binary output Q adopts the active status (1);

- CTUD – the counter that is able to count in two directions and has functionalities as well as inputs and outputs of the both counters described above.

Specification of the CTUD function module in the ST language is showed below:

```
FUNCTION_BLOCK CTUD
 VAR_INPUT
  CU : BOOL R_EDGE;
  CD : BOOL R_EDGE;
  R  : BOOL;
  LD : BOOL;
  PV : INT;
 END_VAR
 VAR_OUTPUT
  QU : BOOL;
QD : BOOL;
  CV : INT;
 END_VAR
 IF R THEN
  CV := 0;
 ELSIF LD THEN
CV := PV;
 ELSE
  IF not(CU and CD) THEN
   IF CU and (CV<PV) THEN
    CV := CV + 1;
   ELSIF CD and (CV>0) THEN
    CV := CV - 1;
   ENDIF;
  ENDIF;
 ENDIF;
 QU := (CV >= PV);
 QD := (CV <= 0);
END_FUNCTION_BLOCK
```

Implementation of counter functions is much more sophisticated as compared to the two remaining groups of function blocks. As one can see above, it is necessary to increment and /or decrement content of the memory cell as well as to determine status of a binary output on the basis of the current content of the counter, i.e. the memory cell. But on the other hand, each of the mentioned functionalities is already implemented for basic operators – addition, subtraction, comparison, not to mention about loading content of a memory cell and transferring both binary and numerical information. Such a set of instruction combined with the ability to use jumps within the routine makes it possible to execute all counter functions with no need to implement additional operators. However, such a solution seems to be inefficient and it is better to define supplementary operators that enable conditional incrementing and decrementing functions.

Implementation of a counter needs mapping of a suitable data structure in the controller memory to enable storage of the current content of the counter as well as status of the PV input and the status of QU and QD outputs. Two more memory cells are necessary as well to store previous statuses

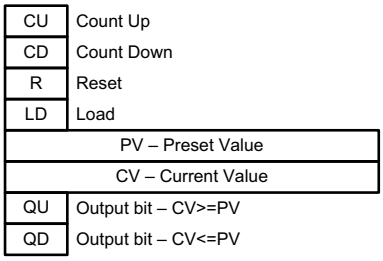

Fig. 2. Minimum representation of a counter structure.

of the CU and CD inputs. The example structure for a single counter unit is shown in Fig. 2. The routine that corresponds to functionalities of the CTUD counter noted in the IL language may look as follows:

```
LD_b IN_CU       ;IN_CU -> CR_b0
CU Counter       ;Counter.CV++ if CR_b0
                 ;changed from 0 to 1
LD_b IN_CD       ;IN_CU -> CR_b0
CD Counter       ;Counter.CV-- if CR_b0
                 ;changed from 0 to 1
LD_b IN_RES      ;IN_RES -> CR_b0
RC Counter       ;0 -> Counter if CR_b0=1
LD_W IN_PV       ;IN_PV -> CR_W0
ST_W Counter.PV  ;CR_W0 -> Counter.PV
LD_b IN_SET      ;IN_SET -> CR_b0
SC Counter       ;Counter.PV -> Counter.CV
                 ;if CR_b0=1
LD_b Counter.QU  ;Counter.QU -> CR_b0
ST_b OUT_MAX     ;CR_b0 -> OUT_MAX
LD_b Counter.QD  ;Counter.QD -> CR_b0
ST_b OUT_MIN     ;CR_b0 -> OUT_MIN
LD_W Counter.CV  ;Counter.CV -> CR_W0
ST_W C_Value     ;CR_W0 -> C_Value
```

The SC instruction highlighted in bold is not covered by standard, although it seems indispensable for the IL language to successfully resolve the problem of counters. Obviously, one can image that the instruction may have a mnemonic similar to all commands from the STORE (ST) group, but the functionality of the command would be totally different since it should be executed only when the condition is fulfilled. Similarly, one more instruction RC should be defined for clearing (reset) of the counter structure when high (1) signal is supplied to the R input.

D. Timers

The mentioned standard defines timers by means of timing waveforms [3]. There are three types of timers described:

- TP – Pulse Timer – acts as a pulse generator which provides a pulse of constant length at the Q output upon a rising edge is detected at the IN input;
- TON – On-Delay Timer – transfers the input value of IN to the Q output with a time delay upon a rising edge is detected at IN.;
- TOF – Off-Delay Timer – delays a falling edge in the same way as TON does it for a rising one.

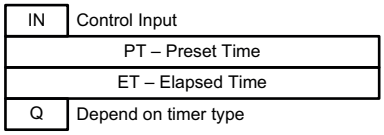

Fig. 3. Minimum representation of a timer structure.

It is impossible to specify how the timers should be implemented within the function blocks that execute the functionalities of waveform generators, thus these functions are usually defined only as timing diagrams. The standard imposes no solution for implementation of timers and manufacturers are very scarce in disclosing their ideas in available documentation. The solution for the timer functions not only needs implementation of the device operation by means of programming languages but also the method itself that is applied to count the elapsing time [4]. The example below and Fig. 3 explain how the code of the function for the TP timer can be defined in the IL:

```
LD_W Time      ;Time -> CR_W0
ST_W Timer.PT  ;CR_W0 -> Timer.PT
LD_b IN_START  ;IN_START -> CR_b0
TP Timer       ;Timer controlling
               ;depends on timer type,
               ;Timer.PT and CR_b0
LD_b Timer.Q   ;Timer.Q -> CR_b0
ST_b OUTPUT    ;CR_b0 -> OUTPUT
LD_b Timer.ET  ;Timer.ET -> CR_W0
ST_b T_Value   ;CR_W0 -> T_Value
```

E. Summary

The foregoing examples show that bistable elements and components for edge detection correspond to rather simple functions that employ only binary components of the central unit. The foregoing description and referred examples clearly show that such elements can be implemented in the way that is purely free of software structures referred to as function blocks. Other function blocks can be implemented in a similar way, but it needs to define a group of new operators that shall be assigned to specific operations on data structures already mapped on the CPU memory.

V. Function Blocks Calls

The standard provides three methods for calls of function blocks in the IL language. All these three call procedures are explained on the example of the call for the timer function module, where the timer type must be specified in the declaration part of the program. The calls themselves fail to distinguish the type of the timer denoted as Time:

1) Using a call including a list of actual input and output parameters provided in brackets:

   CAL Time (IN:=st,PT:=t#1s,Q=>out,ET=>val)

2) Loading and saving input parameters into structures of module cells, calling the module and reading output cells into output parameters:

```
LD t#1s   ;inputs saving
ST Time.PT
LD st
ST Time.IN
CAL Time  ;block calling
LD Time.Q ;outputs reading
ST out
LD Time.ET
ST val
```

3) Calling “implicite” by using the input parameters as operators:

```
LD t#1s
PT Time   ;input saving
LD st
IN Time   ;timer calling
LD Time.Q ;outputs reading
ST out
LD Time.ET
ST val
```

The call with use of the third method differs in such way that the CAL instruction is removed and its role is taken over by the operator that assumed the name of the timer input (memory cell), according to the rule for that method. Consequently, the IN operator appeared. However, the authors are in position that it is an abortive idea and it is more reasonable to simply set up three new operators for timers, namely TON, TP and TOF, instead of determining the timer type when all units of function blocks are being declared. These new functions shall be used as operators necessary to call functionality of a specific timer.

The authors believe that the only reasonable method for calling function blocks defined in the IL language is the third solution since the two remaining methods shall always need more time to execute such a routine. It results from the fact that each execution of the CAL operator entails processing the entire structure of the function blocks, regardless whether all parameters of the block are in use or not as well as in spite of the fact that the status /value of these parameters subject to changes. or not.

VI. Results of Experiments

Table II summarizes execution times for all the operators that are used for execution of routines that correspond to the functions implemented for CPUs within three microcontrollers:

- The MCS-51 microcontroller series was synchronized with the clock frequency of 16MHz and the scan time (necessary to execute 700 instructions on binary variables and 300 instructions on word-type variables) is 10ms [12];
- The AVR microcontroller clocked with 16MHz frequency with the scan time equal to 1.88ms [13];
- The ARM microcontroller with 72MHz clock frequency and 1.05ms scan time [14].

The table comprises the execution times for these commands against the times of the S7-224 [5] and S7-312 [10] CPUs offered by Siemens manufacturer.

TABLE II
EXECUTION TIMES FOR OPERATORS EQUIVALENT TO FUNCTION BLOCKS (IN μS)

Operator	MCS51	AVR	ARM	S7-224	S7-312
LD_b	3,00	0,25	0,06	0,80	0,10
ST_b	9,00	0,25	0,08	1,30	0,14
S	9,75	0,31	0,19	2,90	0,14
R	9,75	0,31	0,19	2,90	0,14
R_TRIG	43,50	0,69	0,47	8,00	0,26
F_TRIG	27,00	0,69	0,47	8,00	0,26
LD_W	3,00	0,50	0,06	18,00	0,28
ST_W	12,00	4,69	0,10		0,28
CU	62,25	5,44	0,80	31,00	1,22
CD	62,25	5,13	0,83	27,00	1,31
RC	47,75	2,88	0,63	9,30	1,15
RT	47,65	2,88	0,63	16,00	1,51
SC	47,25	3,13	0,63	-	1,76
TP	40,50	2,56	0,56	-	1,20 – SP
TON	28,50	1,38	0,64	33,00	1,31 – SD
TOF	30,00	2,00	0,78	36,00	1,37 – SF

VII. CONCLUSIONS

To summarize the result it is necessary to emphasize that the attempt to design cores for CPUs of a programmable controller was successful and the controllers were fully operable and capable of executing all function blocks in accordance with the IEC-61131 standard. However, as it has been demonstrated for the newly developed CPUs, the solutions benefit from calls of function blocks, without direct implementation of them within the C language, but by means of suitable operators of the IL language that are executed upon the calls.

The authors are in position that function blocks may have some *raison d'être* for graphic languages or for the ST language, i.e. the languages with a clear structure. However, if such a strict structure is not desired, for instance in case of the IL language, there is no sense to use such a structure of calls. It is enough to map necessary data structures onto the CPU memory and supplement the list of commands with suitable additional operators that are capable of executing necessary operations on these structures. Not only it is the best solution from the viewpoint of a design engineer for the central unit but it also offers a lot of freedom for a programmer. Instead of limiting the programmer to simple use of function blocks only by calling them *en block*, it offers, with no restrictions, the possibility to incorporate any fragments of the module functionalities into any place of the control routine.

In addition, the study comprises comparison between execution times for components of the IL language that correspond to calls of function blocks within the structures of typical microcontrollers. The CPU designed on the basis of a microcontroller with the ARM core could execute the control routine faster than commercially available CPUs.

In spite of the fact that microcontrollers of the MCS51 are deemed as an "obsolete" design they are subject to continuous improvements that have led, for instance, to the solution engineered and being offered by Digital Core Design (DCD) under the name of DQ80251. Simplicity, high efficiency and great performance – these three features make the 8051 microcontrollers still very popular. Using everywhere 32-bit, heavy RISC processor is pointless, when an 8-bit CPU can do the tasks more economically and eco-friendly, due to much lower power and ASIC area consumption. Digital Core Design portfolio includes the most powerful DQ80251 architecture, which is 66 times faster than the 80C51 and has 50% more efficient code space utilization, comparing to the classic 8051. Digital Core Design has a complete portfolio of 8051 processors, consisting of: very small and effective DT8051 family, most popular high performance DP8051 family and nowadays, the newest DQ80251 version of the most powerful 8051 in the world, which avails faster architecture and smaller ASIC area, than any other competitors' 8051 solution. Each family is embedded with the DoCD JTAG/TTAG real-time, non-intrusive debugging system. All in all, successful implementation of CPUs compatible with requirements of the standard may lead to a solution that would be really powerful comparable even with ARM microprocessor with substantial savings on hardware [15].

Further studies shall be focused on comparison between various methods for implementation of function blocks and calling them for execution. Results from such comparisons shall be disclosed in further studies with specification of the best implementation method.

REFERENCES

[1] F. Bonfati, P. D. Monari, and U. Sampieri, *IEC 61131-3 Programming Methodology; Software engineering methods for industrial automated systems*. ICS Triplex ISaGRAF, 2003.
[2] J. A. Rehg and G. J. Sartori, *Programmable Controllers*. Prentice Hall, 2007.
[3] K. H. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems*. Berlin Heidelberg: Springer-Verlag, 2010.
[4] M. Chmiel, J. Mocha, E. Hrynkiewicz, and D. Polok, "About implementation of IEC 61131-3 IL operators in standard microcontrollers," in *Proceedings of 12th IFAC/IEEE International Conference on PDeS'13*, Velke Karlovice, Czech Republic, September 25-27 2013, pp. 30–35.
[5] K. Koo, G. S. Rho, W. H. Kwon, J. Park, and N. Chang, "Architectural Design of an RISC Processor for Programmable Logic Controllers," *Journal of Systems Architecture*, vol. 44, no. 5, pp. 311–325, February 1998.
[6] M. S. Boggs, T. L. Fulton, S. Hausman, G. McNabb, A. McNutt, and S. W. Stimmel, "Programmable Logic Controller – Method, System and Apparatus," June 3 2003, US Patent No. US 6,574,743 B1.
[7] M. Chmiel, J. Mocha, E. Hrynkiewicz, and A. Milik, "Central Processing Units for PLC implementation in Virtex-4 FPGA," in *Proceedings of the 18th IFAC World Congress*, Milano, Italy, August 28-September 2 2011, pp. 7860–7865, vol. 18, part 1.
[8] A. Milik, "High Level Synthesis – Reconfigurable Hardware Implementation of Programmable Logic Controller," in *PDeS'06*, Brno, February 14-16 2006, pp. 138–143.
[9] J. Mocha and D. Kania, "Hardware Implementation of a Control Program in FPGA Structures," *Electrical Review*, vol. 88, no. 12/2012, pp. 95–100, 2012, (in Polish).
[10] H. Berger, *Automatic with STEP7 in STL and SCL – SIMATIC S7-300/400 Programmable Controllers*. Germany: Siemens AG, 2001.
[11] J. Kulisz, M. Chmiel, and A. Malcher, "Generating time intervals in Programmable Logic Controllers," in *Proceedings of 12th IFAC/IEEE International Conference on PDeS'13*, Velke Karlovice, Czech Republic, September 25-27 2013, pp. 42–47.
[12] T. Białas, "The compact controller based on '51 microcontroller, master thesis,," Master's thesis, Silesian University of Technology, Gliwice, 2012, (in Polish).
[13] R. Zych, "The compact controller based on AVR microcontroller," Master's thesis, Silesian University of Technology, Gliwice, 2012, (in Polish).
[14] M. Juraszek, "The compact controller based on ARM microcontroller," Master's thesis, Silesian University of Technology, Gliwice, 2012, (in Polish).
[15] DCD, "DQ80251 – Revolutionary Quad-Pipelined Ultra High Performance 16/32-bit Configurable Microcontroller," 2013.