BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 68, No. 6, 2020
DOI: 10.24425/bpasts.2020.135386

PLC implementation in the form of a System-on-a-Chip

P. MAZUR[1], R. CZERWINSKI[2]*, and M. CHMIEL[2]

[1] DisplayLink, ul. Ligocka 103, 40-568 Katowice, Poland
[2] Silesian University of Technology, Department of Digital Systems, ul. Akademicka 16, 44-100 Gliwice, Poland

Abstract. The aim of the paper is to present the implementation of a PLC designed in the form of a System-on-a-Chip. The presented PLC is compatible with the IEC61131–3 standard. More precisely, the Instruction List language is the native language of the designed CPU, so there is no need for multiple language transformations. In the proposed solution each instruction of the CPU program written in Instruction List is directly translated to machine code. The designed CPU is capable of performing logic operations up to 32-bit Boolean data types. However, the developed CPU is very flexible due to its architecture: data memory can be addressed as bit/byte/word/dword. Moreover, diverse blocks such as timers, counters, and hardware acceleration blocks, can be connected to the CPU by means of an APB AMBA bus. The designed PLC has been implemented in an FPGA device and can be used in cyber-physical systems and Industry 4.0.

Key words: PLC, FPGA, AMBA, APB, IEC 61131-3.

1. Introduction

A Programmable Logic Controller (PLC) is an industrial digital computer, adapted for the control of manufacturing processes, such as robotic devices or assembly lines. The main advantage of PLCs over the classical methods of implementing control systems is the possibility to change the operating algorithm without the need to reconstruct the entire control system [1].

Growing requirements caused by the implementation of cyber-physical systems and Industry 4.0 expose shortcomings in classical PLCs for different tasks such as classical functionality (control systems), data acquisition and processing, and communication. That is the reason for building PLC-based systems where the CPU is one of many modules besides communication modules, etc. However, Modern PLCs can be built in a System-on-a-Chip (SoC) form—for example to increase safety [2]. Moreover, programmable SoCs can be implemented using Field-Programmable Gate Array (FPGA) technology. There is no need to use external modules for different functionality in such a system. This saves time (for operation) and money, because of deep integration of functionalities.

The PLC performance does not always satisfy the recent requirements in large and highly responsive systems. A control program can be directly implemented in the FPGA with hard-wired logic for higher response and reduced implementation cost/space. In this case PLC instructions are converted into hardware description language (VHDL/Verilog) code, and then synthesized and implemented as logic circuit with various peripheral functions [3–5]. The problem is that such a solution requires synthesis and the structure is not universal. Completely another solution offers modern Systems-on-Chip, where the development is moving toward multiprocessor based design. It is suggested that multiprocessor SoCs will become the predominant class of embedded systems in future [6]. Multiprocessor designs are also used in the field of industrial control [7]. It turns out that the hardware approach can be combined with processor/multiprocessor-based approach and build classical SoC dedicated to industrial control [8–10].

The basic parameters of industrial controllers are: processing time of one thousand instructions (Scan Time), access time to internal and external resources, and transition time, which are closely related [1, 11]. When developing the optimal design of central units designed for operation in control systems using PLC controllers, all these parameters should be taken into account. Proposed solutions should execute the control program as soon as possible, which can be achieved by reducing the time of execution of individual instructions, minimizing the time of access to object signals, and optimizing the response time to changes in the state of process variables. This is of particular importance if there is a need for rapid response to emergency situations, as well as when part of the control system requires a very short response time to input changes [12].

The way programmable controllers work is defined by the IEC61131 standard. The third part of the standard describes the syntax and semantics of programming languages for PLCs [13, 14]. Some manufacturers offer controllers that can be programmed using languages compliant with IEC61131–3 [15–18]. It seems that very often the hardware structure of the PLC controller does not comply with the software standard. Manufacturers use „translators"—programs written in the standard language are converted into their own language, and then compiled [14]. This approach often means that the use of a PLC is not optimal. In fact, the programmable controller resources are not compatible with the standard. The best solutions can be achieved when the IEC61131-3-based Instruction List (IL) language is the native CPU language [17, 19].

*e-mail: rczerwinski@polsl.pl

Manuscript submitted 2020-03-31, revised 2020-07-17, initially accepted for publication 2020-10-08, published in December 2020

Each microprocessor has its own assembly language, so PLCs' central processing units are built on the basis of such microprocessors. Therefore, when general purpose microprocessors are used, we deal with the situation of a multi-level translation of a program written in one of the IEC standard programming languages, in order to finally get the machine code (Fig. 1). In general, the IEC instructions are substituted with a set of microprocessor-based instructions (functions, procedures or macros). In the presented compilation flow the IL program is directly translated into machine code, as it is highlighted in Fig. 1. When dedicated microprocessor is used, the IEC-based IL language can be directly translated into machine code that saves execution time [9, 10]. Moreover, such a microprocessor can effectively use its resources to perform instructions. The example of data memory construction is presented in the paper. Data processing for bit/byte/word/double word data can be processed in atomic way without masking and other data long time preprocessing. Simple example is presented in Fig. 2, where three simple operations LD/AND/ST (Fig. 2a) must be translated into three macros (Fig. 2b) in order to execute them by means of CPU.

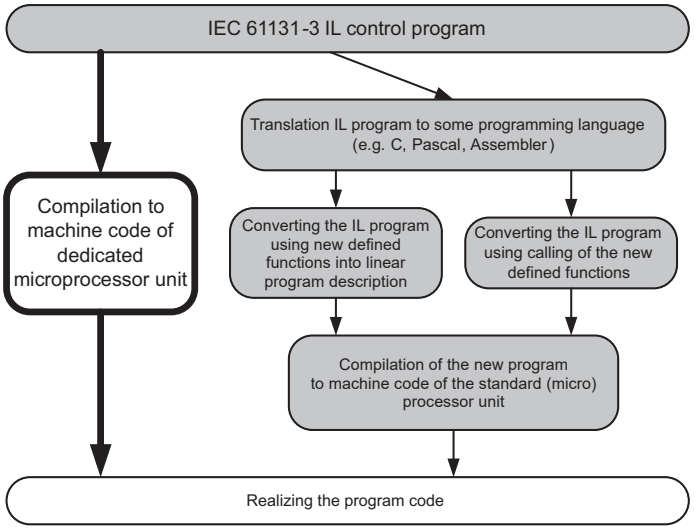

Fig. 1. Program translations flow

Increased requirements in relation to cyber-physical systems, and further Industry 4.0, require new concepts of PLCs. Our aim is to present the implementation of a PLC designed in the form of a System-on-a-Chip. The main contribution is the design of the PLC structure with atomic operations for different data widths and a structure that enables implementing dedicated blocks such as communication interfaces (here: counters and timers). It is important that the designed PLC is compatible with the IEC61131-3 standard [13].

The PLC general architecture is presented in Section 2. The CPU construction is described in details in Section 3. The most important part „Data Memory module" is presented in Subsection 3.3. AMBA APB interface is shown in Section 4, while the example application is described in Section 5. Next,

a)	b)
LD A (bit)	Load A into ACCU_A; A -> ACCU_A Mask bit(nr_A) in ACCU_A; 0000A000 AND ACCU_A -> ACCU_A Shift right ACCU_A to the right; 0000000A -> ACCU_A Store ACCU_A into the ACCU_B; ACCU_A -> ACCU_B
AND B (bit)	Load B into ACCU_A; B -> ACCU_A Mask bit(nr_B) in ACCU_A; 00B00000 AND ACCU_A -> ACCU_A Shift right ACCU_A to the right; 0000000B -> ACCU_A ACCU_A AND ACCU_B -> ACCU_A; ACCU_A AND ACCU_B -> ACCU_A (C)
ST C (bit)	Shift left ACCU_A to the nr_C; 000C0000 -> ACCU_A Store ACCU_A into the ACCU_B; ACCU_A -> ACCU_B Load C into ACCU_A; C -> ACCU_A ACCU_A OR ACCU_B -> ACCU_A; ACCU_A OR ACCU_B -> ACCU_A ACCU_A -> C; ACCU_A -> C

Fig. 2. Example of simple program and its translation to general purpose microprocessor

in Section 6 there are experimental results. Paper closes with conclusions.

2. PLC general architecture

In general, the Central Processing Unit (CPU) is responsible for executing programs. However, the CPU must also be prepared to establish communication between particular blocks. If engaged devices remain in close proximity (on the same chip) and there is a tight requirement in terms of operating speed, parallel buses are used. For cabled communication, a serial bus is the only reliable option (ProfiBus, MODBUS, ProfiNet, USB, SPI). The structure of a designed PLC is depicted in Fig. 3. It utilizes both serial communication in the form of an SPI interface utilized only for CPU programming, and a parallel one in the form of an AMBA APB interface used for connecting peripheral blocks to the CPU.

The major advantage of the presented structure is its modularity. Given that the APB interface gives the ability for multi-slave operation [20], a handful of peripheral devices can be utilized, extending the capabilities of the PLC according to specific needs.

A simplified block diagram of the designed CPU is depicted in Fig. 4. It consists of two communication interfaces:

- SPI-Slave interface for CPU programming,
- the APB-Master interface for connecting specialized peripheral blocks like external memory, calculating units, counters, timers, other function blocks.

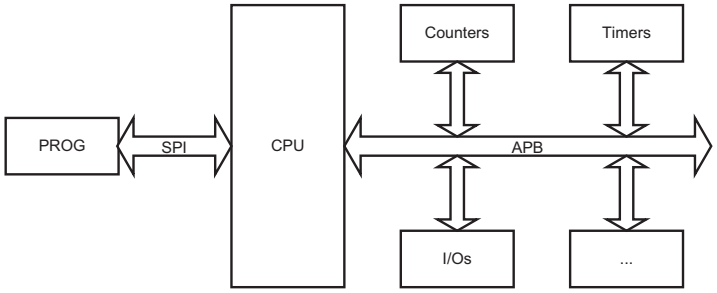

Fig. 3. Structure of the proposed programmable logic controller

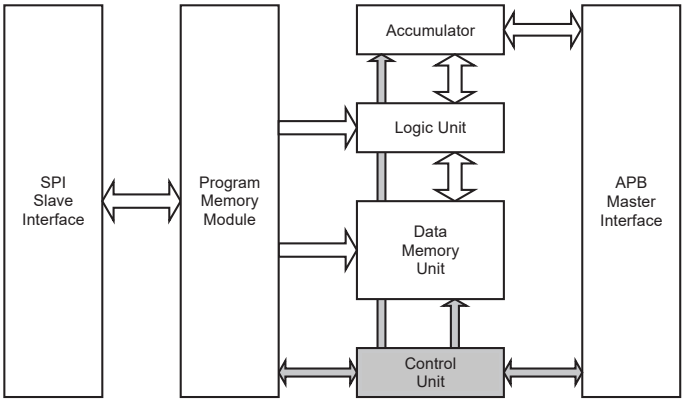

Fig. 4. Structure of the Central Processing Unit

The most important part of the designed CPU is the sequential, Finite State Machine (FSM)-based Control Unit that controls both program execution and data flow between the CPU and peripheral devices connected to the APB bus. The designed CPU follows the principles of the Harvard, single argument, microprocessor architecture with Program and Data Memories separated and single Accumulator register.

3. CPU construction

The fundamental operation of any CPU, regardless of the physical form they take, is to fetch and decode both the instruction code and the operand from Program Memory and prepare the Program Counter for the next instruction. In the designed CPU, this process is controlled by a sequential module called the Control Unit. Its principle of operation is to decode the instruction code fetched from the Program Memory and generate appropriate control signals for other modules. The Control Unit consists of two Mealy Finite State Machines: the first for instruction cycle control and the second for APB-bus control. The operation of the Control Unit varies depending on the type of the instruction that is preceded. Moreover, specific instructions utilize different modules of the CPU. The Control Unit operations will be discussed in this section, together with other details of the CPU construction.

The Program Memory module consists of four blocks: Programming Controller, Program Memory, Program Counter and Instruction Register. The instruction fetching process is performed by the Program Counter and Program Memory and is controlled by the Control Unit. The Program Counter addresses the Program Memory thus controlling the flow of program execution. During normal operation, program execution is linear and the Program Memory output register is updated with the contents from consecutive addresses. There might however be some exceptions from that flow in the form of instructions that can modify the Program Counter content directly (jump instructions).

3.1. Logic Unit module. The Logic Unit is an integration module that provides the data path between Program Memory, Data Memory and Accumulator register (named CR—Current Result). First of all it consists of two multiplexers: one generates input data for the Accumulator register, and the other generates input data for the Data Memory. Its principle of operation is very straightforward. Based on the instruction code (instr_code[7:0]) the appropriate input signal is selected, and taken as the output to the Accumulator register (lu_out_acc[31:0], Fig. 5) or the Data Memory (lu_out_dm[31:0], Fig. 6). The unit is of 32-bit width. PLCs are mostly 32 units because it is enough for most problems in industrial control.

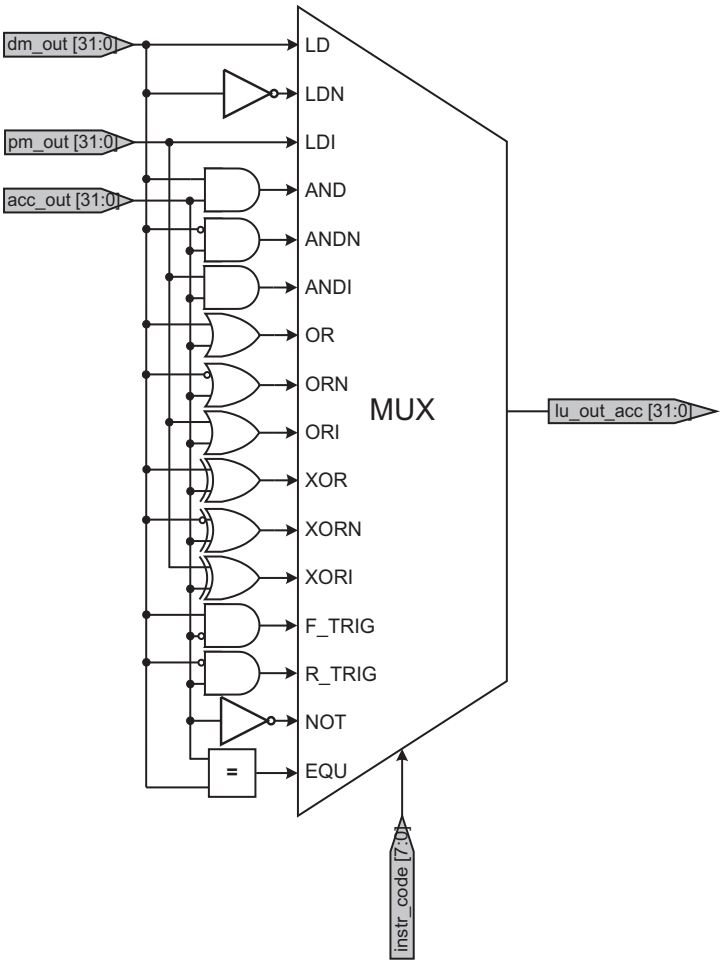

Fig. 5. Basic operations for accumulator input

Basic operations defined in the IEC 61131–3 standard are implemented by means of logic functions connected to the Accumulator by means of a multiplexer (Fig. 5). It generates input data for the Accumulator register and can be treated as load-type. The load-type instructions include: LD/LDN/LDI, AND/ANDN/ANDI, OR/ORN/ORI, XOR/XORN/XORI, NOT, EQU and F_TRIG/R_TRIG.

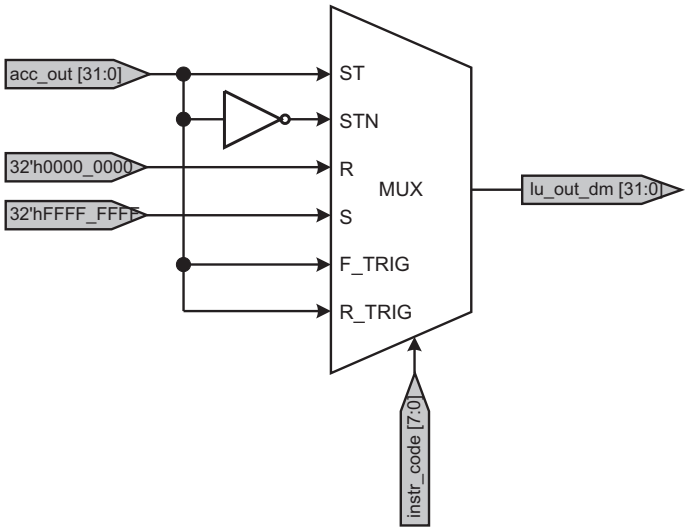

Fig. 6. Basic operations for Data Memory input

The Data Memory MUX (Fig. 6) provides the signal to be written at the appropriate memory address. It realizes store-type operations, that are: ST/STN, R, S, and F_TRIG/R_TRIG. The F_TRIG/R_TRIG operations must ensure write operations into the Data Memory because the edge is calculated by means of memory marker (previous state), so must store the operation into memory. Instructions S and R provide setting and resetting operations. The condition from the Accumulator is taken into account.

In fact, F_TRIG/R_TRIG functions are of load-store type, so they are represented in Fig. 5 as well as in Fig. 6.

The control unit forms write signals into appropriate modules. This signal is of course dependent on the realized instruction.

3.2. Instruction execution. The principle of operation for every presented instruction is identical, with the exception of instructions with immediate operands ('I' modifier) which behave differently due to the change in source of the data latched into the Accumulator register or Data Memory.

As presented in Fig. 7, a LD (load-type) instruction takes two clock cycles to execute (called phases). During the first phase, the Program Counter is incremented (*pc_en* → *pc_out*) and the Data Memory output register is updated with the operand—appropriate memory cell addressed by the Program Memory (*dm_en* → *dm_out*). During the second phase, the data generated by the Logic Unit module is registered in the Accumulator register (*acc_en* → *acc_out*). At the same time,

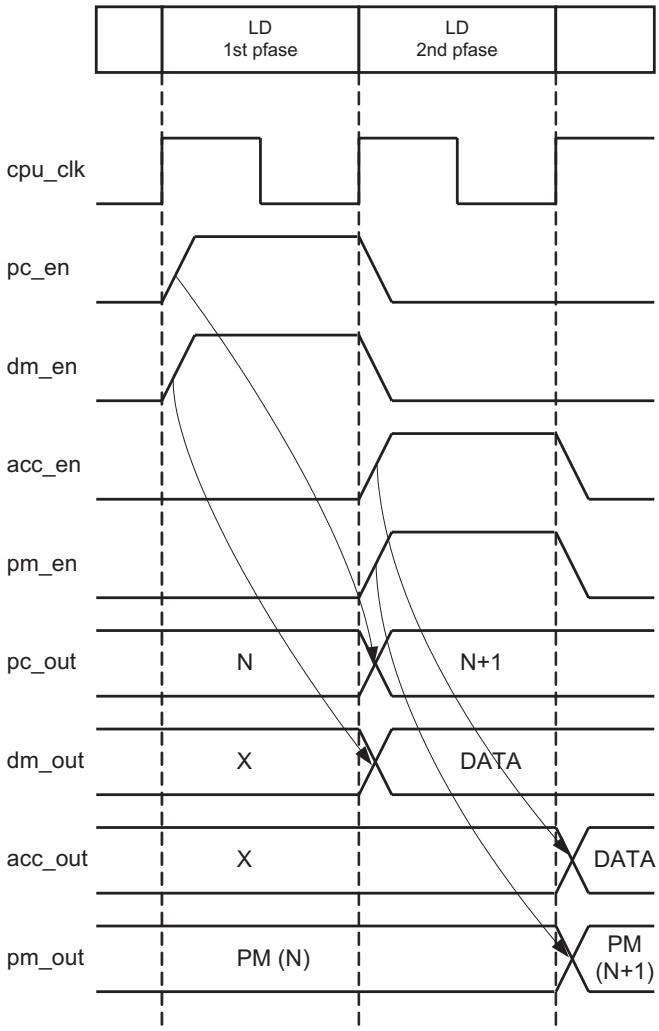

Fig. 7. LD (load type) instruction timing diagram

the Program Memory output register is updated with the instruction code and operand from the consecutive Program Memory address (*pm_en* → *pm_out*). The Control unit is designed to access synchronous (Sync) SRAM memories. That is the reason for one clock cycle delay in memory access.

Load-type instructions with immediate operands heavily affect the Program Memory architecture. In general there are two conceptions of operand-placing in the Program Memory that are common in modern microprocessors: both instruction code and corresponding operand may have the same, or consecutive addresses. These solutions differ in the number of memory access cycles needed to be performed during a single instruction cycle. In the designed CPU, both solutions are utilized. In the cases of instructions where the operand is in the form of a Data Memory address, both instruction code and corresponding operand have the same address (Fig. 8). In the cases of immediate addressing, both the instruction code and corresponding operand are separated (Fig. 8). This approach has been taken to reduce the design's area utilization, while maintaining high speed of execution of the most common instructions and has led to the organization of the Program Memory.

Program Memory Address	Program Memory Content	
	Bits [31:24]	Bits [23:0]
0	<instruction code>	<operand – DM address>
1	<instruction code>	<empty>
2	<operand – program constant>	
3	<instruction code>	<operand – DM address>

Fig. 8. Program Memory organization

The CPU's operation during the execution of an LDI (load-immediate-type) instruction is presented in the form of a timing diagram in Fig. 9. In addition to load-type instructions, the LDI instruction utilizes an Instruction Register. The Instruction Register is a part of the Program Memory module and its purpose is to hold the instruction code for the last three phases of an LDI instruction's execution (as the Program Memory output register is updated with the program constant instead).

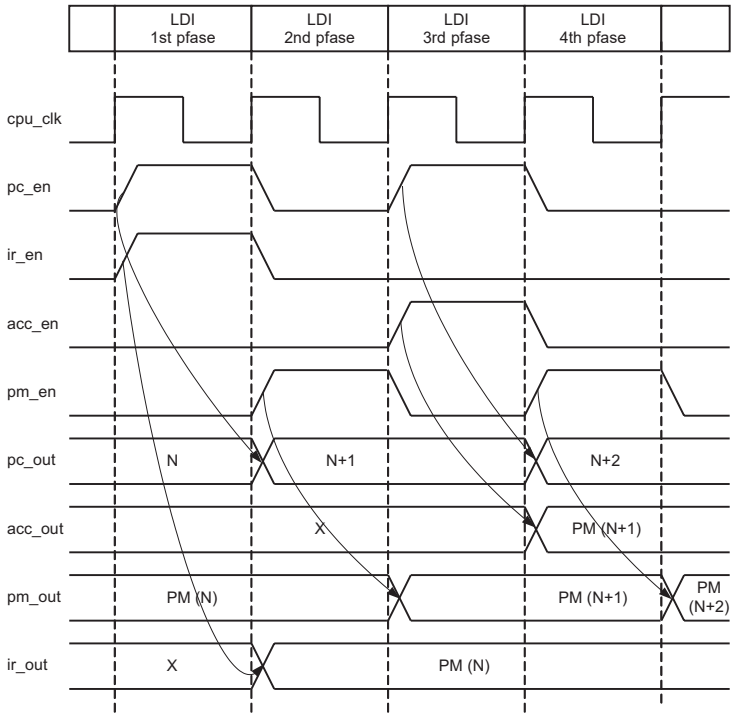

Fig. 9. LDI instruction transaction

As presented in Fig. 9, an LDI (load-immediate-type) instruction has four phases of execution. During the first the Program Counter is incremented (*pc_en* → *pc_out*) and the instruction code is registered in the Instruction Register (*ir_en* → *ir_out*). Then, during the next phase, the Program Memory output register is updated with the program constant (from the consecutive Program Memory address; *pm_en* → *pm_out*). During the third phase, data generated by the Logic Unit is registered in the Accumulator register (*acc_en* → *acc_out*). At the same time, the Program Counter is incremented once again (*pc_en* → *pc_out*). During the final, fourth phase, the Program Memory output register is updated with the instruction code and operand from the consecutive Program Memory address (*pm_en* → *pm_out*).

There are several instructions that perform data store into the Data Memory (store-type instructions). They take two clock cycles to execute and their process is analogous to load-type instructions.

As long as load and logic operations are obvious in the Accumulator-source-multiplexer (Fig. 5), then the F_TRIG/R_TRIG functions are a significant change compared to classical PLCs. In classical PLCs edge detectors are realized as function blocks, that is in the form of a program. This consumes time for calculating the function block result, because a trigger function requires at least two instructions and a shadow register for Accumulator (CR'), as presented in List. 1 and List. 2. The structure presented in Fig. 5 enables calculation of F_TRIG/R_TRIG within one instruction cycle. Edge detectors are considered in detail in [9].

Listing 1. Trigger function with additional Current Result (CR') memory bit: version 1

```
LD          IN      ;IN->CR, CR'
R_TRIG      MEM     ;(CR AND MEM) -> CR
            ;NOT CR' -> MEM
ST          Q       ;CR -> Q
```

Listing 2. Trigger function with additional Current Result (CR') memory bit: version 2

```
LD          IN      ;IN->CR
R_TRIG      MEM     ;CR->CR'
            ;(CR AND MEM) -> CR
            ;NOT CR' -> MEM
ST          Q       ;CR -> Q
```

F_TRIG/R_TRIG instructions consist of both data load into the Accumulator register and data store into the Data Memory. As presented in Fig. 10, a R_TRIG instruction takes three clock cycles to execute. During the first phase, the Program Counter is incremented (*pc_en* → *pc_out*) and the Data Memory output register is updated with the operand (appropriate memory cell addressed by the Program Memory; *dm_en* → *dm_out*). During the second phase, separate data generated by the Logic Unit module is registered in the Accumulator register and the Data Memory (*acc_en* → *acc_out*, *dm_en* → *dm_out* and *dm_wr*). During the third phase, the Program Memory output register is updated with the instruction code and operand from the consecutive Program Memory address (*pm_en* → *pm_out*).

There are three jump instructions implemented in the CPU: JMP, JMPC and JMPCN. Both JMPC and JMPCN are conditional—based on actual state of the Accumulator register—jump instructions. A jump instruction takes two clock cycles to execute: during the first phase the Program Counter is loaded with the operand, while during the second phase the Program Memory output register is updated with the instruction code and operand from the previously loaded Program Memory address.

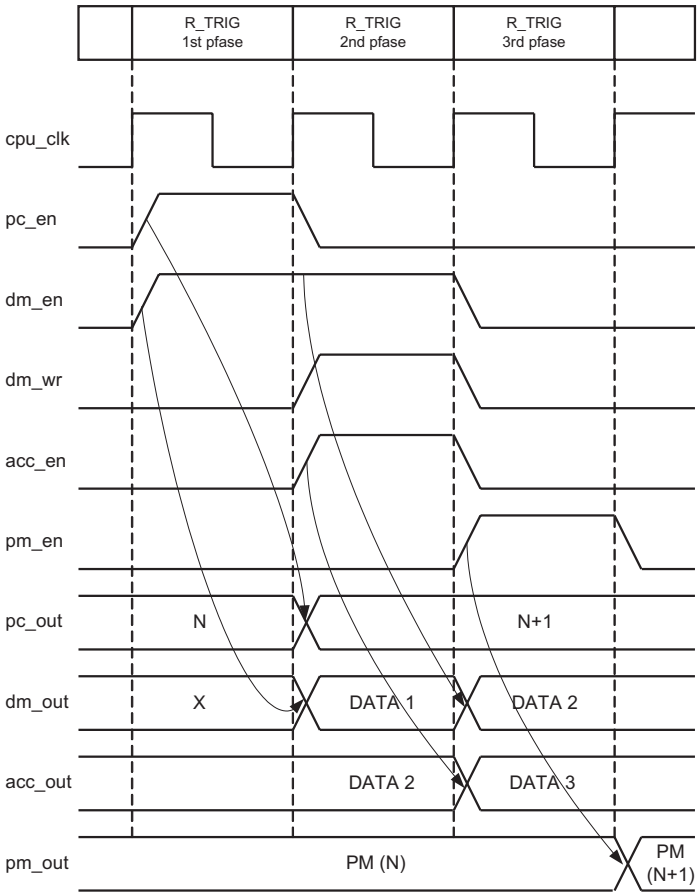

Fig. 10. R_TRIG instruction, timing diagram

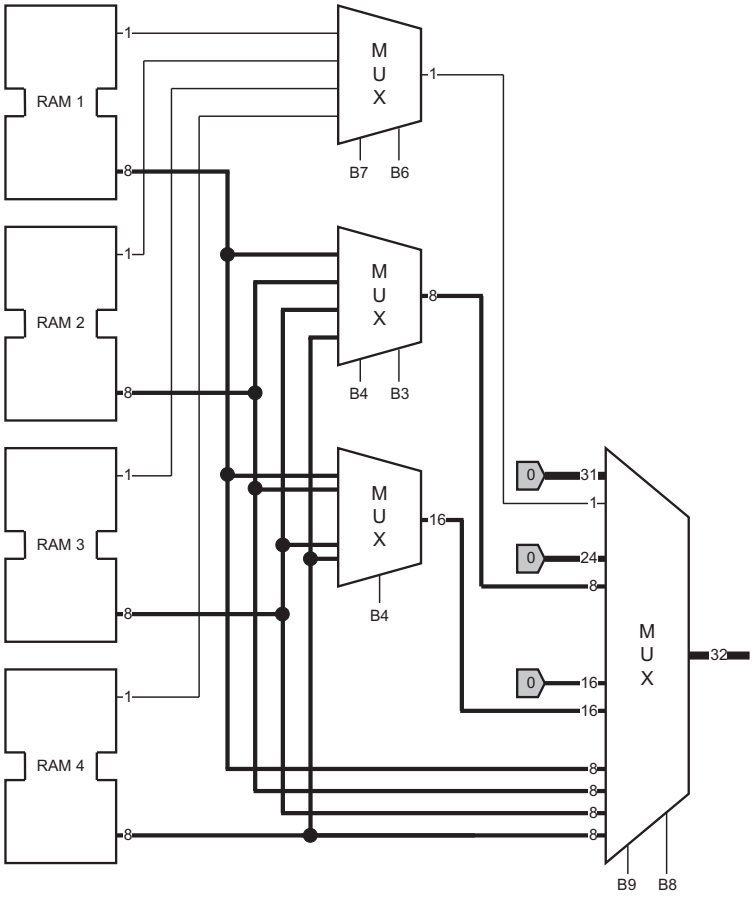

Fig. 11. Data Memory block diagram: output part

3.3. Data Memory module. The Data Memory module is a memory block responsible for holding program variables. Although the designed CPU is of 32-bit architecture, the Data Memory has been designed to enable 1-bit, 8-bit (byte) and 16-bit (word) access as well, without the need for software masking. So the particular bit/byte/word can be read/written in one clock cycle. The output part of the Data Memory module is depicted in Fig. 11.

As presented in Fig. 11, the Data Memory module consists of multiplexers and the Data Memory itself. This structure is similar to modern memories, where data can be written by means of bytes. However, in the designed memory data can be written and read by means of bytes as well as words. Moreover, the memory block consists of four identical dual-port memories—a single instance of the dual-port memory used in the design is depicted in Fig. 12.

As presented in Fig. 12, dual-port memory utilized in the design has two ports, one is 1-bit and the other is 8-bit wide. Due to utilization of four instances of presented dual-port memory in the Data Memory block, an 8-bit, 16-bit or 32-bit access can be performed by simultaneously enabling one, two or four instances of dual-port memories respectively.

While the control over read and write accesses is carried out by the Control Unit module (by generating dm_en and dm_wr signals), the selection of appropriate dual-port memories is performed by multiplexers (see Fig. 13). The coding of Data Memory access types is presented in Table 1.

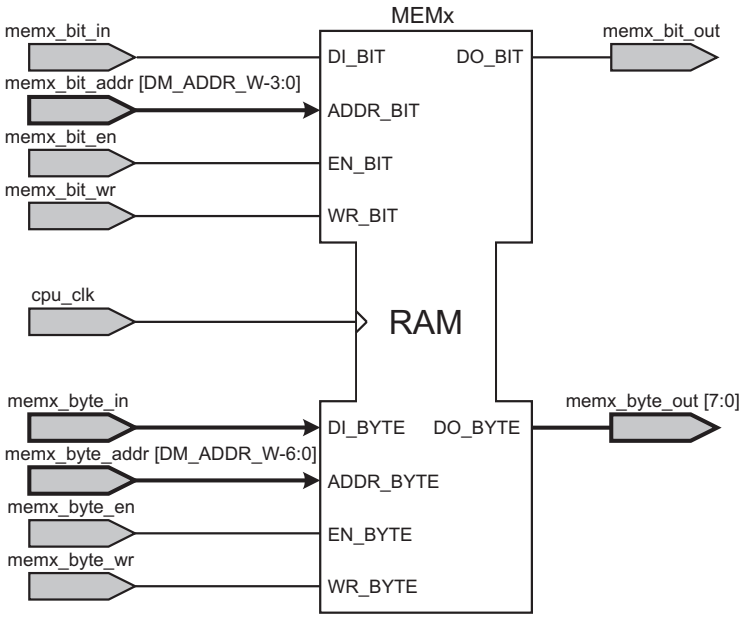

Fig. 12. Dual-port memory

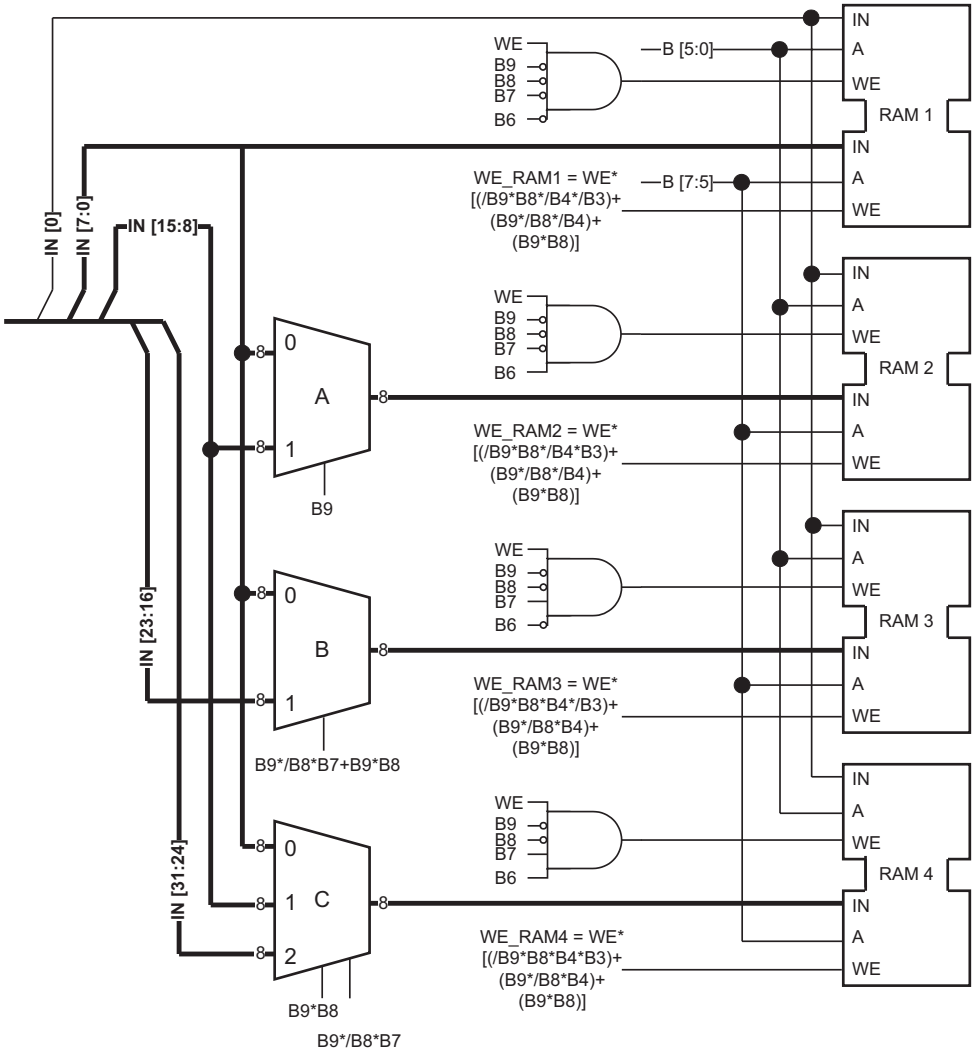

Fig. 13. Data Memory block diagram: input part

Table 1
Data Memory access types

Data Memory access type	Binary Code
BIT	00
BYTE	01
WORD	10
DWORD	11

An example of the Data Memory address frame for a configuration with the Data Memory address being 8-bit wide is presented in Fig. 14. As presented, the Data Memory is very straightforward. For example, to access the specific bit of a given byte in the Data Memory, bits B7–B3 of a BIT address shall mirror the given BYTE address and by appropriate (binary coded) setting of bits B2–B0 each BIT of a given byte can be accessed separately. This rule applies to every access type. Two MSBs denote the access type (bit/byte/word/dword). The example is shown.

Access Type		Address							
B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
0	0	BIT address							
0	1	BYTE address					-	-	-
1	0	WORD address				-	-	-	-
1	1	DWORD address			-	-	-	-	-

DWORD Address			Example			
B_7	B_6	B_5				
0	0	0		0x012		
0	0	1				
0	1	0			0x240	
0	1	1				
1	0	0				
1	0	1		0x1B0		
1	1	0	0x3C0			
1	1	1				

Fig. 14. Data Memory address frame

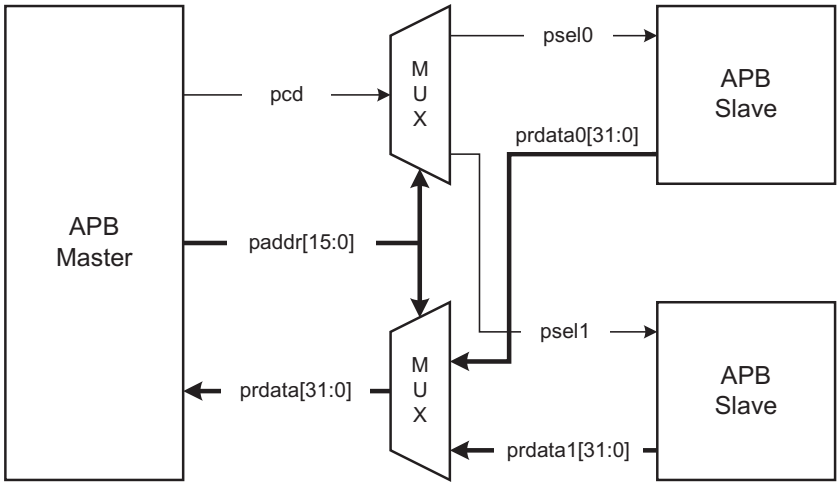

Fig. 15. AMBA 3 APB Multi-Slave operation mode

4. AMBA APB Master interface

The designed CPU has been equipped with an AMBA 3 APB Master interface as primary bus to communicate with peripheral devices. This is a new idea in constructing PLCs and gives the significant advantage that modules, including function blocks, can be connected to the CPU in a very flexible way. For example function blocks like timers, counters or bistable functions may be used by means of the APB bus. Moreover, it is not necessary to design standard function blocks. The most interesting are custom (application specific) blocks. A constructor can design for example a hardware module that supports some signal processing, artificial neural networks [21], dedicated time interval measurement module [22], a module for data interface or other dedicated module [4].

The APB Master Interface implemented in the CPU fully supports the AMBA 3 APB standard and does not utilize the optional PSLVERR pin [20]. The AMBA 3 APB standard allows for Multi-Master operation. If there are two or more APB-Slave devices connected to the bus, there is a need of an additional decoding module that would generate separate PSELx signals for each APB-Slave device. The exemplary system that consists of one APB-Master and two APB-Slave devices is presented in Fig. 15.

The APB transfer is initiated if one of two APB-related instructions have been decoded: APB_WR or APB_RD. These instructions are extensions to those compatible with the IEC 61131-3 standard. Thanks to simple instructions APB_WR or APB_RD a programmer can exchange data between modules connected to the APB bus and CPU. Such a program can be included as part of the operating system of the designed PLC. This makes this PLC very flexible.

Control over the APB Write or APB Read transaction is then taken by the APB FSM located in the Control Unit. After a successful transaction, the APB FSM relinquishes the control, and the CPU returns to normal operation. This process is presented by the examples of both APB_WR and APB_RD instructions depicted in Fig. 16 and 17 respectively. In both

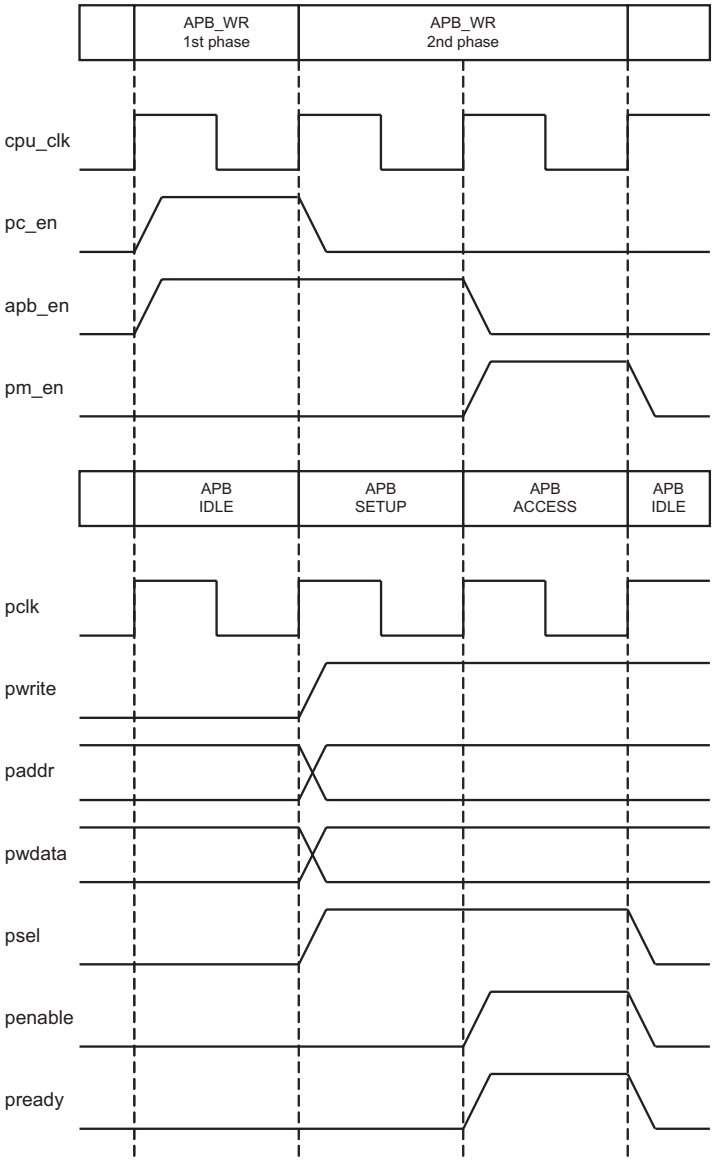

Fig. 16. APB_WR instruction timing diagram

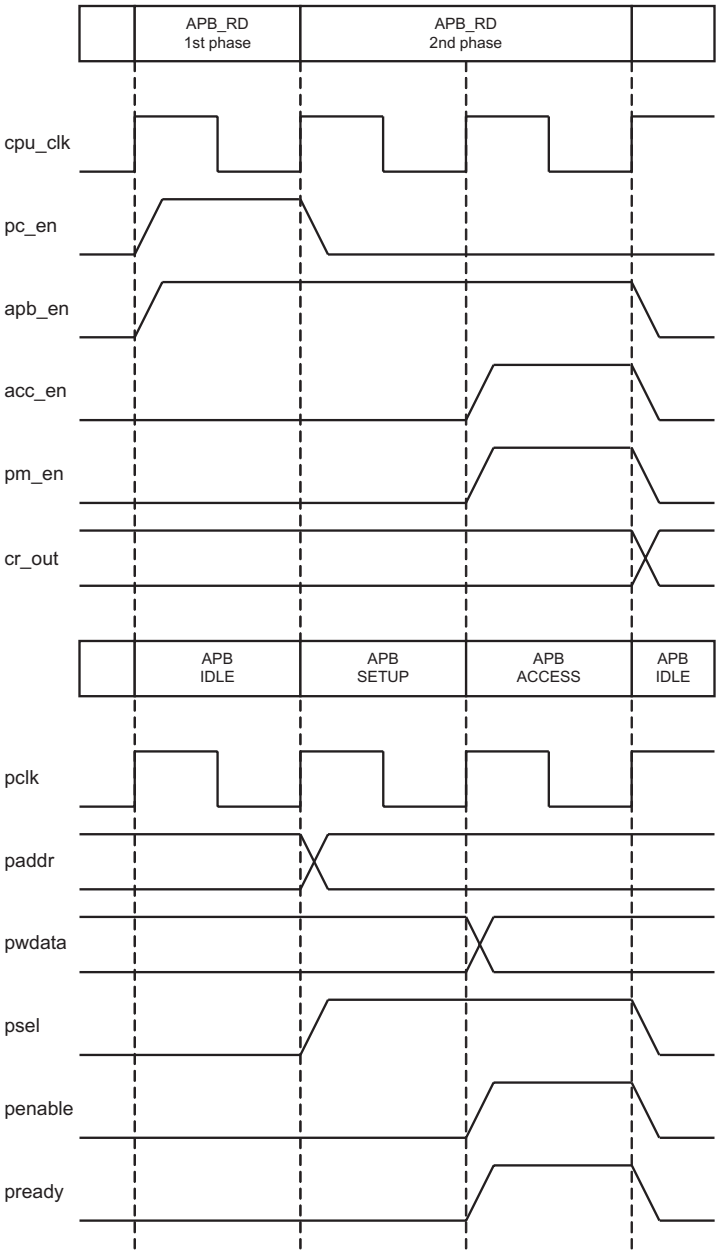

Fig. 17. APB_RD instruction timing diagram

cases APB transfer is presented as without wait states. If there are wait states forced by the APB-Slave, the only difference is the length of the APB_ACCESS phase.

5. Timer module

A timer module will be used as the example of a hardware module used as the peripheral unit connected via APB interface. In a classical PLC timers are designed as software modules. This means that a timer is a fragment of program that operates on the memory structure. In the proposed design timers can be built as hardware blocks and persist independently. There is no need to spend time operating on data. The block diagram

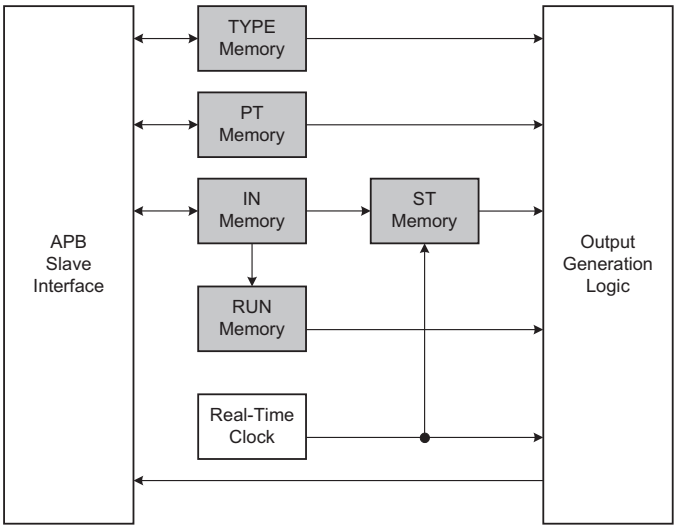

Fig. 18. Timer module block diagram

of the timer module is depicted in Fig. 18, where: TYPE is Timer Type (TP, TON, TOF), PT is Preset Time with the resolution of 1 millisecond, IN is Input and holds the Input (Start), RUN–holds the information about which timer has been started (if whatever timer has been started appropriately for its type, the corresponding field in this memory will be set High) and ST–Start Time holds the information about the state of the Real-Time Clock at which the given timer has been started; it is used for generating both ET and Q outputs (as per the IEC 61131-3 specification). Each timer consume five consecutive addresses from ABP Address map: TYPE, PT, IN, Q, and ET.

Another example of this type of module, however simpler, is a counter module.

6. Experimental results

The CPU's Verilog source code is fully synthesizable and can be implemented in any FPGA device of a sufficient size. Both the device utilization and the timing reports depend heavily on the CPU's configuration. Results for an exemplary configuration for a Xilinx Kintex7 FPGA are presented in Table 2. The address width for data memory as well as for the program memory was 6 for the first implementation and 16 for the second.

Table 2
Synthesis report

Parameter	width = 6	width = 16
Slice Registers	458	234
Slice LUTs	664	577
IOs	92	92
Block RAMs	3	66
Minimum *cpu_clk* period	4.648 ns	3.976 ns
Maximum *cpu_clk* frequency	215.128	251.484 MHz

Both device utilization reports show that the design can be successfully implemented using a small FPGA device. The design with address width 6 was based on Distributed-RAM, while the second ($width = 16$) is based on Block-RAM. The comparison of both timing reports shows that the increase in Block RAM utilization in comparison with the Register and LUT utilization has a positive impact on timing parameters of the design. Summarizing, the LD (or bit operations) can be executed within 8 ns.

The results of synthesis are comparable for example to Vex RISC-V implementation [23] (considered to be very small due to implementation by means of Spinal HDL).

The most important comparison is with vendor PLCs. However, it is a difficult comparison, because for the newest CPUs only sample data about execution times are published. In the S7–1217 CPU (Siemens S7–1200 family) Boolean instructions are executed in 80 ns [24]. The most powerful PLC family by Siemens is the S7–1500 [25]. The execution time for a bit instruction is 10 ns for the S7–1516 and 60 ns for the S7–1511 CPU. An extremely fast (and most expensive) example is the S7–1518 CPU with 1ns execution time for bit instructions and 2 ns for word operations and fixed arithmetic type operations. The Quantum CPU53414B Controller by Schneider Electric times differ from 100 ns up to 500 ns [26]. The VIPA Siemens-compatible PLCs (e.g. 314–6CF02) can execute instructions on bit variables in a minimum 10 ns [27], while General Electric PAC Systems RX3i gives a CPU that executes one Boolean instruction from 29 ns (for CRE040 type) to 253 ns (for CPU310 type) [28].

One of the most important difference between the designed unit and vendor CPUs is that the unit presented in the paper can execute bit/byte/word/double word operations in two clock cycles. This is not normal practice. Generally, one data size is privileged and data processing for such data is done directly, while for other data sizes particular data is processed for example by masking (logical operations). This takes a lot of time, e.g. in Siemens S7–312 LD for a byte takes twice as long as for a bit [29], for S7–1217 Boolean instructions are executed in 80 ns, while move word 137 ns [24].

Two more advantages must be indicated for the CPU presented in this paper. First, the trigger function is consistently realized in three clock cycles (even in 12 ns). This is faster than in the very quick CPU Siemens S7–319, where the R_TRIG/F_TRIG operations take 40 ns [29]. The second advantage concerns access and control times to timers and counters. It always takes less than 8 ns (for the quicker units), while in the S7–300 it takes 100 ns. The reason is that in the presented CPU, timers and counters are implemented as hardware blocks with APB access time. Moreover, the presented CPU can be reworked to achieve even faster R_TRIG/F_TRIG operations [9].

7. Conclusions

The aim of the paper was to present the implementation of a PLC, compatible with the IEC61131–3 standard. The presented CPU is capable of performing logic operations on 32-bit Boolean data types specified by the IEC 61131–3 standard. Of course, the presented solution is not developed with the same depth of complexity as the commercial solutions. However, the results achieved are very promising. The form of System-on-a-Chip is used to implement the PLC.

The most important advantage is that the developed CPU is very flexible due to its architecture: data memory can be addressed as bit/byte/word/dword and different blocks can be connected to the CPU by means of an APB AMBA bus.

The access time to bit/byte/word/dword is not dependent on data width. The access to memory is by means of the designed hardware and not by means of data processing (masking operations); so it is done in atomic way. This is very important in some applications because it increases the system security and system speed. Moreover, such a solution is compatible with the IEC61131–3 standard, where CR has no defined width.

The APB AMBA bus application gives a real possibility to implement a PLC in the form of a System-on-a-Chip. A timer application is shown in this paper. This is a hardware timer that works concurrently with the CPU. Communication with blocks is performed during the system communication in the PLC scan cycle. To perform more experiments the dedicated HMI panel was applied and the example programs were visualized. Instructions that enable communication with the APB AMBA bus are not included in the IEC 61131–3 standard, but enable building systems with hardware aided dedicated blocks (timers, counters, interfaces, etc.). Such a structure can be used in modern cyber-physical systems or Industry 4.0.

The designed CPU and proposed PLC architecture has been simulated and widely tested by UVM methodology and by means of hardware laboratory set. The CPU was the core of the system presented in Digilent Design Contest 2018.

Acknowledgements. This work was supported by the Polish Ministry of Science and Higher Education funding for statutory activities.

References

[1] W. Bolton, *Programmable Logic Controllers*. Newnes, 2009.
[2] W. Halang and M. Sniezek, "A safe programmable electronic system", *Bull. Pol. Ac.: Tech.* 58(3), 423–434 (2010).
[3] S. Ichikawa, M. Akinaka, H. Hata, R. Ikeda, and H. Yamamoto, "An FPGA implementation of hard-wired sequence control system based on PLC software", *IEEJ Trans. Electr. Electron. Eng.* 6(4), 367–375 (2011).
[4] E. Monmasson, L. Idkhajine, M. Cirstea, I. Bahri, A. Tisan, and M. Naouar, "FPGAs in industrial control applications", *IEEE Trans. Ind. Inform.* 7(2), 224– 243 (2011).
[5] A. Milik and E. Hrynkiewicz, "Synthesis and implementation of reconfigurable PLC on FPGA platform", *Int. J. Electron. Telecommun.* 58(1), 85–94 (2012).
[6] T. Dorta, J. Jimenez, J. Martin, U. Bidarte, and A. Astarloa, "Overview of FPGA-based multiprocessor systems", in *International Conference on Reconfigurable Computing and FPGAs*, 2009, pp. 273–278.

[7] A. Milik, "Multiple-core PLC CPU implementation and programming", *J. Circuits Syst. Comput.* 27, 1850162 (2018).
[8] Z. Hajduk, B. Trybus, and J. Sadolewski, "Architecture of FPGA embedded multiprocessor programmable controller", *IEEE Trans. Ind. Electron.* 62(5), 2952–2961 (2015).
[9] R. Czerwinski and M. Chmiel, "Hardware-based single-clock-cycle edge detector for a PLC central processing unit", *Electronics (MDPI)* 8(12), 1529 (2019).
[10] M. Chmiel, "FPGA-based implementation of bistable function blocks defined in the IEC 61131", *Microprocess. Microsyst.* 65, 37–46 (2019).
[11] J. Kasprzyk, *Industrial controllers programming*. WNT, 2006, [in Polish].
[12] Y. Birbir and H. Nogay, "Design and implementation of PLC-based monitoring control system for three-phase induction motors fed by PWM inverter", *Int. J. Syst. Appl. Eng. Dev.* 2, 128– 135 (2008).
[13] International Electrotechnical Commission, "EN 611313:2013, programmable controller–Part 3: Programming languages", European Committee for Electrotechnical Standardization, Tech. Rep., 2013.
[14] K. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems*. Springer, 2010.
[15] Z. Hajduk, J. Sadolewski, and B. Trybus, "FPGA-based execution platform for IEC 61131–3 control software", *Prz. Elektrotechniczny (Electrical Review)* 87(8), 187–191 (2011).
[16] Rockwell Automation, "Logix5000 controllers IEC 61131-3 compliance", Rockwell Automation Publication 1756-PM018C-EN-P, Tech. Rep., 2003.
[17] S. Rudrawar and M. Patil, "Design and implementation of FPGA based high performance instruction list (IL) processor", *IOSR J. Electron. Commun. Eng.* 1(4), 38–45 (2012).
[18] J.-H. Huang, Y.-C. Li, Z. Luo, X.-X. Liu, and K.-F. Nan, "The design of new-type PLC based on IEC 61131–3", in *Proceedings of the Second International Conference on Machine Learning and Cybernetics*, 2003, pp. 809–813.
[19] M. Okabe, "Development of processor directly executing IEC 61131–3 language", in *SICE Annual Conference, The University of Electro-Communications, Tokyo, Japan*, 2008, pp. 2215–2218.
[20] ARM Limited, *AMBA 3 APB Protocol*, 2004.
[21] Z. Hajduk, "Hardware implementation of hyperbolic tan gent and sigmoid activation functions", *Bull. Pol. Ac.: Tech.* 66(5), 563–577 (2018).
[22] G. Grzeda and R. Szplet, "Time interval measurement module implemented in SoC FPGA device", *Int. J. Electron. Telecommun.* 62(3), 237246 (2016).
[23] "VexRISC-V, An FPGA friendly 32 bit RISC-V CPU implementation", https://github.com/SpinalHDL/VexRiscv, (Access: 14.07.2020).
[24] Siemens AG, *Simatic S7-1200 Programmable Controler. System Manual*, 2019.
[25] Siemens AG, *S7-1500, S7-1500R/H, ET 200SP, ET 200pro Cycle and Reaction Times. Function Manual*, 2018.
[26] Schneider Electric, *Modicon Quantum automation platform, Hot standby system Unity Pro*, 2013.
[27] VIPA GmbH, *VIPA System 300S SPEED7-CPU 314-6CF02*, 2014.
[28] General Electric Company, *Intelligent Platforms, Programmable Control Products, PACSystems, RX7i&RX3i CPU Reference Manual, GFK-2222W*, 2015.
[29] Siemens AG, *Simatic S7–300 Instruction List, CPU312, CPU314, CPU315–2DP, CPU315–2PN/DP, CPU317–2PN/DP, CPU319–3PN/DP, IM151–8PN/DP CPU, IM 154–8 PN/DP CPU*, 2015.