

Article

Reconfigurable State Machine Module for FPGA-Based Programmable Controllers

Zbigniew Hajduk 

Department of Computer and Control Engineering, Rzeszów University of Technology, 35-959 Rzeszów, Poland; zhajduk@kia.prz.edu.pl

Abstract

Existing field-programmable gate array (FPGA) implementations of programmable logic controllers (PLCs) still suffer from limited speed compared to high-performance industrial controllers or restricted programming flexibility, if a control algorithm is directly mapped into hardware without a CPU. In this article, a new approach is proposed that may mitigate these drawbacks. The approach is to integrate with an FPGA-based PLC system, a configurable hardware module, called the RSM, that is capable of implementing some combinational or sequential control algorithms on a hardware basis. The key idea behind the RSM module is that the actual functionality of the module is not statically set during FPGA project implementation by FPGA synthesis tools, but it can be established at any time during FPGA-based PLC system operation. Since the FPGA synthesis tools are not needed, this approach combines high programming flexibility with the high performance ensured by hardware implementation. This article presents a detailed architecture of the RSM module, which is capable of implementing up to 12 Boolean functions of up to 12 variables, or a state machine with up to 12 state variables, along with some IEC 61131-3-compliant function blocks. The RSM can find application in cases where high-speed implementation of logic control algorithms is needed, whereas the other less time-demanding tasks can be executed by the CPU.

Keywords: programmable logic controllers; field-programmable gate arrays; IEC 61131-3 standard

1. Introduction

Programmable logic controllers (PLCs) are widely used in many industrial automation applications. Their robustness, easy programming using the IEC 61131-3 standard languages [1], and resilience to harsh industrial environmental factors make them particularly suitable for these applications. Nowadays, the execution speed of PLC programs seems to be an essential issue for some types of applications. One of the PLC speed improvement possibilities, particularly exploited in research areas, is the use of field-programmable gate array (FPGA) technology.

FPGAs provide the feasibility of direct implementation of control programs in parallel hardware logic, enable integration of hardware-accelerated modules such as function blocks, and allow the design of customized multicore CPUs optimized for specific control operations [2–4]. Along with their reconfigurability feature, they seem to be a promising platform for the performance enhancement of PLCs.



Academic Editor: Stefano Ricci

Received: 17 September 2026

Revised: 12 October 2026

Accepted: 14 October 2026

Published: 18 October 2026

Copyright: © 2026 by the author.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons](#)

[Attribution \(CC BY\)](#) license.

1.1. Related Works

There are a number of research papers involving FPGAs in the context mentioned above. For example, in [4], a comparison of four different classes of FPGA implementations of the ladder diagram (LD) language is considered, namely the interpreted software, compiled software, interpreted hardware, and compiled hardware. This work also explores parallel versus serial execution of LDs in hardware and identifies timers as important resources used in parallel implementation. A dedicated software tool, named Hestigo, which was developed in [5], converts LDs to synthesizable Verilog hardware description language (HDL) code for the FPGA implementation. Another tool that converts LD control programs into Verilog HDL, suitable for FPGA implementation of control systems, is presented in [6]. An optimized conversion of the ladder diagram into the VHDL description is considered in [7]. Ref. [8] shows, in turn, an architecture of the dedicated ladder-rung processor (LRP) that natively executes LD programs. Both FPGA and ASIC implementations of this processor are considered in that work. A method of hardware control program implementation, which allows direct mapping to look-up tables (LUTs) of an FPGA, is described in [9]. The method involves LD and sequential function chart (SFC) programs according to the IEC61131-3 standard.

There are also several works involving a specific central processing unit (CPU), optimized for PLC operations. In [2], the design and implementation of a PLC microprocessor, adhering to the IEC-61131-3 standard, are presented. This processor was implemented in an Intel/Altera Cyclone V FPGA and needs three clock cycles for Boolean operation execution (60 ns for the Cyclone V FPGA). Another processor that executes the instruction list (IL) language of the IEC 61131-3 standard is briefly described in [10]. This processor involves a five-stage pipeline and was simulated for the Xilinx Spartan-6 FPGA. An interesting microprocessor design, capable of execution of the IL language, is presented in [11]. Four clock cycles are needed to execute a single instruction by this processor. The authors consider its implementation in AMD/Xilinx (Artix-7 family) and Intel/Altera (Cyclone IVE) FPGAs.

A complete PLC controller implemented on an FPGA chip is presented in [12]. Special attention was paid in this case to the design of a dedicated CPU. Another design of a programmable logic controller targeted for FPGA implementation is documented in [13]. The instruction set of this CPU involves a subset of the IL instructions defined in the IEC 61131-3 standard. In [14], another PLC design, targeted for FPGA implementation, and its full development environment are considered. The PLC may contain more than one CPU, whose machine language is compliant with the EN 61131-3 standard. In [15], an architecture of a programmable logic controller that uses the RISC-V processor [16] is considered. The RISC-V executes IL programs converted to the C language and then compiled to native RISC-V machine instructions, while some function blocks (according to the IEC 61131-3 standard) are implemented directly in hardware. An FPGA-based controller that controls an industrial cold-walled atomic layer deposition system is presented in [17]. In this case, the LD control program is converted by a dedicated tool into Verilog HDL code, which is used for the FPGA implementation. A hardware implementation of specific function blocks for FPGA-based programmable logic controllers, namely timers and counters of the IEC 61131-3 standard, is considered in [18,19], respectively.

Some of the previous works by the author of this paper also include FPGA implementations of PLC-related components. In [20], the design of a complete multiprocessor programmable controller was presented, including analog and digital input/output modules. This controller was programmed using the CPDev control environment [21], which compiles PLC programs, according to the IEC 61131-3 standard, to intermediate codes typically executed by software virtual machines running on dedicated microprocessor

platforms. Therefore, the instruction set of the single CPU of the controller presented in [20] directly implemented all instructions of the intermediate code, generated by the CPDev. However, these instructions were complex, required multiple clock cycles to perform an instruction, and were not optimized for hardware implementation. This was the factor that compromised the overall performance of the CPU. An upgraded version of the CPU, the instruction set of which directly implemented the IEC 61131-3 IL language instructions, was presented in [22]. This CPU, based on the RISC paradigm, was able to execute instructions within a single clock cycle. Therefore, the test programs, running on a single CPU, performed significantly faster than in the case of [20], but still slower than high-performance industrial controllers.

1.2. Analysis and Contribution

Considering the short survey presented in the previous paragraphs, works related to the use of FPGAs to support PLC tasks can be divided into the following categories: namely, the conversion of LD (or SFC) programs to the codes represented in VHDL or Verilog hardware description languages [4–9], the design of dedicated CPUs, optimized for PLC operations [2,10,11,22], the FPGA implementations of complete PLC controllers, including customized CPUs [12–17,20], and the design of dedicated hardware function blocks allowing very fast execution of particular tasks [18,19]. However, taking into consideration the second and third categories, despite the fact that the overall performance of the FPGA-based PLCs (with a single CPU) might be better than many industrial microprocessor-based PLCs, it still is not as high as that of the high-performance PLCs or programmable automation controllers (PACs). This is due to the significantly lower clock frequency that can be obtained in FPGAs compared to ASIC CPUs (usually around 100 MHz versus a few GHz) [11,22]. Therefore, the solutions that can compete with high-performance PLCs are located in the first and fourth categories. However, these solutions suffer from significantly lower flexibility than the remaining categories. Apart from PLC program conversion tools, they require the use of FPGA vendor-dependent synthesis and implementation tools, which is an inconvenient, time-consuming process and may cause licensing (legal) issues if applied by regular PLC users.

As the results of tests conducted by [20,22] indicate, the highest performance drawback of the FPGA-implemented CPUs, with reference to high-speed PLCs, is particularly observed for a relatively simple sequence of logic operations. Therefore, in this paper, a new approach is proposed, which combines high flexibility, the same as for the middle category solutions, with high performance ensured by hardware implementations. This approach is to add to the FPGA-based PLC system a hardware module (or multiple modules) capable of calculating a number of combinational functions with a certain number of variables, or running a sequential control algorithm described by a state machine with a certain number of states and with some IEC 61131-3-compliant function blocks. The hardware module is called the reconfigurable state machine (RSM) module since its function can be changed during normal PLC operations, without reusing FPGA synthesis tools.

The general idea of the RSM module slightly resembles the use of programmable input/output (PIO) modules in some contemporary microcontrollers, for example [23]. Similarly to PIOs, there can be multiple RSMs in the systems running in parallel with the CPU. However, contrarily to PIOs, where their functionality is defined by a sequence of machine instructions executed by a specific PIO processor, the RSM performs operations directly in hardware as purely combinational functions or as a sequential circuit. The exact functionality of the RSM is, in turn, configured in a similar manner as look-up tables (LUTs) inside an FPGA are programmed. However, the RSM can be configured at any time during the operation of the FPGA-based PLC system, by the CPU or an external PLC

programmer, without the use of FPGA synthesis tools. Running in parallel with the CPU means, in turn, that the RSM can be assigned to the realization of stand-alone, time-critical logic control operations, while the CPU, at the same time, can be engaged in other, less demanding tasks. The RSM can also be used as a hardware coprocessor for the CPU, allowing very fast (in a single clock cycle) calculation of combinational functions with a certain number of variables.

The remainder of the paper is organized as follows. Section 2 presents in detail an architecture of the RSM module. Section 3 provides, in turn, simulation results and implementation outcomes, whereas Section 4 concludes the article.

2. The RSM Module

The RSM is an additional hardware block in the FPGA-based PLC system, which performs either a sequential control algorithm, described by a state machine, or a combinational function with a certain number of input variables. The RSM can operate independently of the CPU, exercising, for example, time-critical logic control, or it can act as a Boolean coprocessor for the CPU, calculating some Boolean equations defined by truth tables. The RSM communicates with the CPU through a global variable communication scheme, according to the IEC 61131-3 standard.

The general block diagram of the RSM module is presented in Figure 1. The diagram resembles in part the architecture of a Moore-type sequential control circuit with the next-state combinational function, state memory, depicted in Figure 1 as the D-type flip-flop, and the combinational output function. In addition, the RSM also includes a number of specific function blocks (compliant with the IEC 61131-3 standard), connected within the feedback loop, and the single-event upset (SEU) detector and recovery circuit, which allows the safe implementation of state machines with the one-hot encoding scheme.

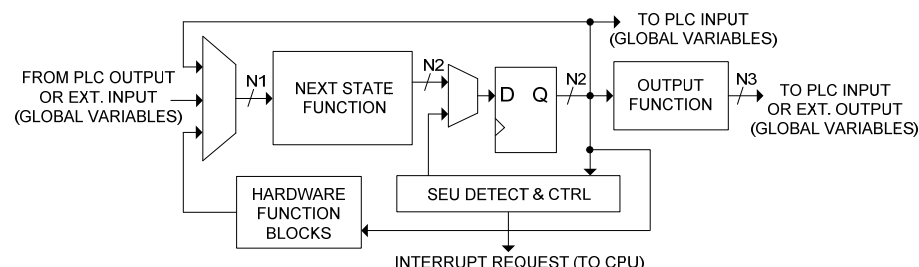


Figure 1. Simplified block diagram of the RSM module.

The key idea behind the RSM module is that the truth tables for the next state and output functions, along with connections of signals within the feedback loop, defining the exact architecture of the sequential control circuit, can be configured at any time during the PLC system operation. The configuration can be performed by the CPU using global variables (the external input/output address space for the CPU from [22]) or by an external PLC programmer. No FPGA vendor-dependent tools are required for the configuration. However, dedicated software may be needed to specify a state machine graph describing a custom control algorithm and convert it into the configuration words for the RSM. This software, however, is not the subject of this article.

In a general case, the block of the next state function in Figure 1 can calculate in parallel N_2 Boolean equations with N_1 variables, whereas for the block of the output function, N_3 Boolean functions with N_2 variables can be calculated. The N_1 , N_2 , and N_3 parameters in Figure 1 also correspond directly to the number of bits of the signal buses connecting key components of the RSM module. However, in the specific case presented in the further part of the article, all these figures are equal to 12 ($N_1 = N_2 = N_3 = 12$). In addition, to save

FPGA resources, the output function has a simplified architecture, which allows only some functions of 12 variables, or any function of only 6 variables, to be implemented. In the case of the next state function, each equation can calculate any function of 12 Boolean variables.

2.1. The ZLUT12 Block

This block is an essential component of the RSM module. It performs any combinational function of 12 variables, which can be configured at any time during the system operation. This functionality is achieved by the use of a select type of LUT inside an FPGA chip that can act as basic distributed RAM memory blocks. In AMD/Xilinx 7-series FPGAs, LUTs of this type constitute the library components called RAM64X1S [24]. A single RAM64X1S block with synchronous write and asynchronous read operation is capable of storing 64 data bits. Therefore, similarly to a regular 6-input LUT, it can be used to implement any combinational function of six variables. If a higher number of variables is required, more of these blocks, appropriately connected to each other, are needed.

Figure 2 presents the architecture of the ZLUT8 block, which allows for an implementation of any combinational function of eight variables. This block involves four RAM64X1S blocks with the A0...A5 inputs connected together. These inputs act as the address inputs when the write operation is performed, or as the data input for the combinational function in the case of the normal operation. Data input (D) and write enable input (WE) are used during the write operation, which specifies the combinational function performed by the ZLUT8 block. The regular six-input LUT is statically configured, during a standard FPGA configuration process after the power is delivered to the chip, as the 4:1 multiplexer. I0...I3 are the data inputs, whereas I4 and I5 are the address inputs of this multiplexer that also act as the two additional inputs (I6, I7) for the eight-variable combinational function performed by the ZLUT8 block.

Analogously, for the realization of a 10-input combinational function, four ZLUT8 blocks and a regular six-input LUT are needed. For a 12-input function, in turn, four blocks performing a 10-input function and a six-input LUT are needed. Therefore, for the ZLUT12 block, performing the latter mentioned function, a total of 64 RAM64X1S blocks and 16 six-input LUTs are needed. For simplicity, the exact architecture of the ZLUT12 block is not presented here. However, this architecture can easily be inferred based on the above description.

2.2. Architecture of the RSM Module

The simplified architecture of the essential part of the RSM module is presented in Figure 3. It involves 12 ZLUT12 blocks, which allow for an implementation of 12 independent Boolean equations. Each equation, in turn, can include up to 12 variables. These variables may stem from the registered outputs (FD-0...FD-11 flip-flops) of the ZLUT12 blocks, global variables of the PLC system (INT_INP[0]...INT_INP[11] inputs in Figure 3), external digital inputs of the PLC (EXT_INP[0]...EXT_INP[11] inputs in Figure 3), or from the outputs of hardware function blocks (M2_I3[0]...M2_I3[11] inputs in Figure 3). For each ZLUT12 block, the exact origin of these variables is configured through multiplexers MUX-i and MUX2-0-i...MUX2-11-i, respectively, where $i \in \{0, 1, \dots, 11\}$ identifies the particular ZLUT12 block. MUX1-0-i...MUX1-11-i multiplexers, together with the WE signal and common data inputs D [63:0], are used in the configuration (programming) process when the particular function, performed by the RSM, is being defined. Using the MUX4-i multiplexer, the initial state or recovery state (when an SEU occurs) can be defined if the RSM is configured to perform the function of a sequential control circuit.

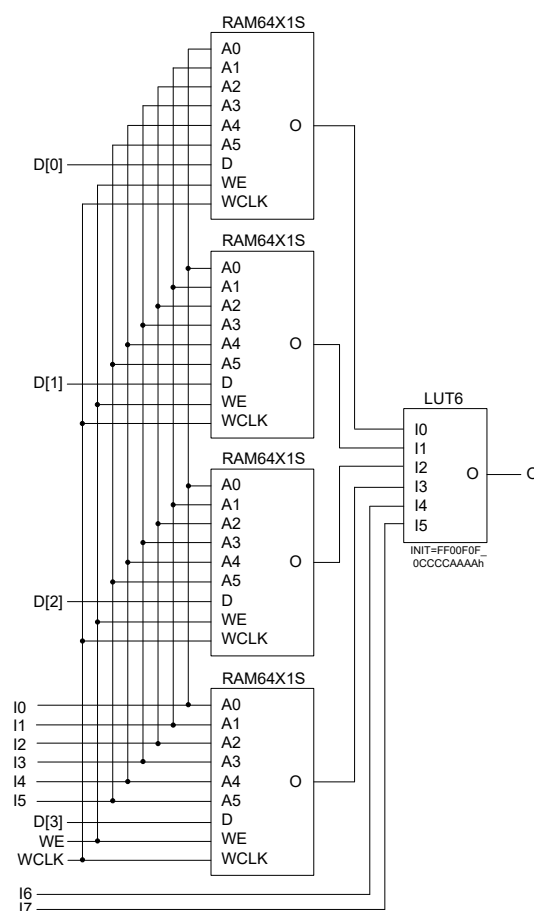


Figure 2. Architecture of the ZLUT8 block.

The black square symbol related to the address inputs of multiplexers in Figure 3 denotes the configuration bits, delivered by the control block (CTRL). These bits, similarly to the other input bits of the CTRL block, such as I11..I0, CONF, D[63:0], INT_INP[11:0], etc., are concatenated into 32-bit words. Each one of these words has an address assigned by the CTRL block located in the global variable address space. Using this space, the truth tables for ZLUT12 blocks, as well as address inputs of multiplexers with black rectangles in Figure 3, can be configured during the PLC system operation. The same address space can also be used to assign current values to input variables for the ZLUT12 blocks and to read their registered outputs during system operation.

The CTRL block of the RSM module is directly interfaced with the CPU from [22]. This block also controls the behavior, particularly during the configuration phase, of the block of the output function and the hardware function blocks. These blocks are not directly depicted in Figure 3. They will be described briefly in the following subsections.

2.3. Architecture of the Block of the Output Function

The block of the output function, which is also distinguished in Figure 1, calculates output values of a sequential control circuit if the RSM module is configured to implement such a type of control circuit. According to the Moore model, these values depend only on the current state of the state machine (Q_RSM [11:0] outputs in Figure 3). Since the RSM module depicted in Figure 3 may utilize up to 12 state variables, the block of the output function should perform a number of Boolean equations of 12 variables. Therefore, the ZLUT12 blocks, described in Section 2.1, can be used to support this task. However, a simplified architecture of the block of the output function is actually applied, which

allows only some functions of 12 variables, or any function of only six variables, to be implemented. This architecture is presented in Figure 4.

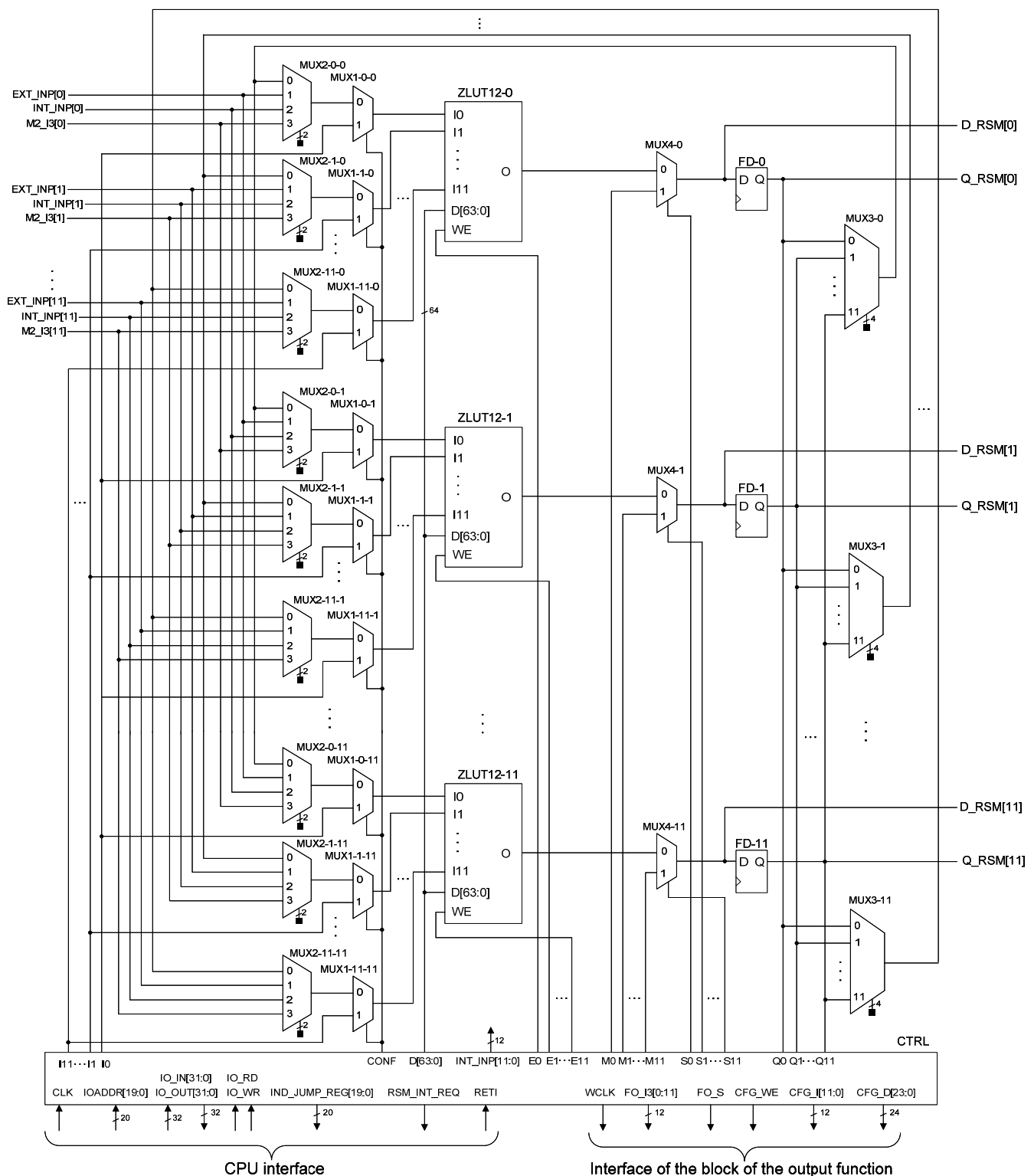


Figure 3. Architecture of the key part of the RSM module.

The block of the output function comprises 12 sub-blocks, called FOUT_0...FOUT_11. Each of these sub-blocks contains two RAM64X1S components and a regular six-input LUT (the top-right area in Figure 4, covered by the rectangle with a noncontinuous line). The top RAM64X1S component (H) can be configured to perform any six-input combinational

function of the Q_RSM[5:0] state variables, whereas the lower RAM64X1S component (L) can implement any six-input combinational function of the remaining Q_RSM[11:6] state variables. The regular six-input LUT is, in turn, statically configured to perform a logic alternative function (OR) of two outputs of RAM64X1S components if the FO_S input signal (the I2 inputs of the regular LUT) is in the low logic state, or this LUT simply re-affirms the logic state existing on the CFG_I[i] configuration input (the “i” denotes the number of a subsequent FOUT block) in the case where the FO_S is in the logic high state. This functionality allows outputs of the RSM to be defined when the sequential control circuit is stopped by the CPU.

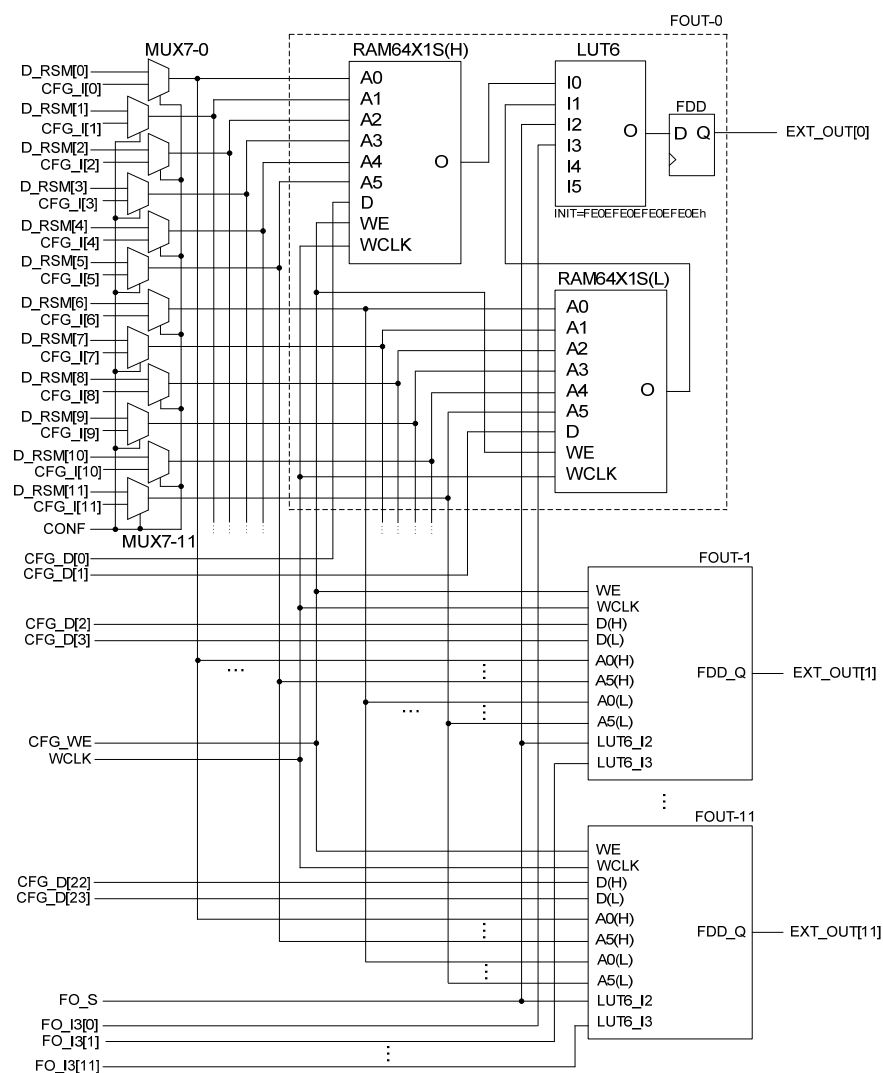


Figure 4. RTL architecture of the block of the output function.

To ensure that the outputs of the block of the output function occur in the same clock cycle as the state variables (outputs of FD_0...FD_11 flip-flops in Figure 2), the inputs of RAM64X1S components are actually connected to the data inputs of the FD_0...FD_11 flip-flops (D_RSM [11:0] outputs in Figure 3). Using, in turn, MUX7_0...MUX7_11 multiplexers, RAM64X1S components can be configured by the CTRL block of the RSM module to perform a desired combinational function.

2.4. Hardware Function Blocks

When the RSM module is configured to perform a sequential control circuit task, a transition between states in this sequential circuit may also encompass signals that are

outputs of additional hardware blocks. These blocks may be the standard function blocks defined by the IEC 61131-3, or they can be other customized blocks, implemented with the standard FPGA design methodology. As an example of the integration of hardware function blocks with the RSM module, the use of the IEC 61131-3 TON timers will be shown. However, the detailed architecture of the TON instance will not be considered here. It can be found, for example, in [18].

The architecture of hardware function blocks, including four instances of the TON timers (HFB-0...HFB-3), is depicted in Figure 5. Each instance of the TON, in turn, can be used in four different states if the RSM implements a finite-state machine. This feature has been achieved due to the application of four comparator blocks (CMP-0-i...CMP-3-i, where “i” denotes the instance number) in each TON instance. The comparator tests if the current state of the machine (the Q_RSM[11:0] lines) equals the predefined value, associated with the second (B) input of the comparator and denoted as a black rectangle (the configurable bits) in Figure 5. If this condition is met, the IN input of the TON instance is activated until the timer counts the predefined number of clock cycles. This number is delivered by the MUX6-i multiplexer, and it is directly associated with the specific state, which causes the activation of the TON instance. Such behavior is accomplished using a priority encoder (ENC-i) block with a two-bit output driving the address input of the MUX6-i. When the predefined number of clock cycles is counted by the TON instance, and the IN input is still active, the Q output is activated. Using configurable multiplexers MUX5-0...MUX5-11, the state of Q outputs of all TON instances is delivered to the M2_I3[11:0] inputs of the RSM from Figure 3. Thus, the Q outputs can then be used as a variable in the transition condition leading to activation of another state of the sequential control circuit.

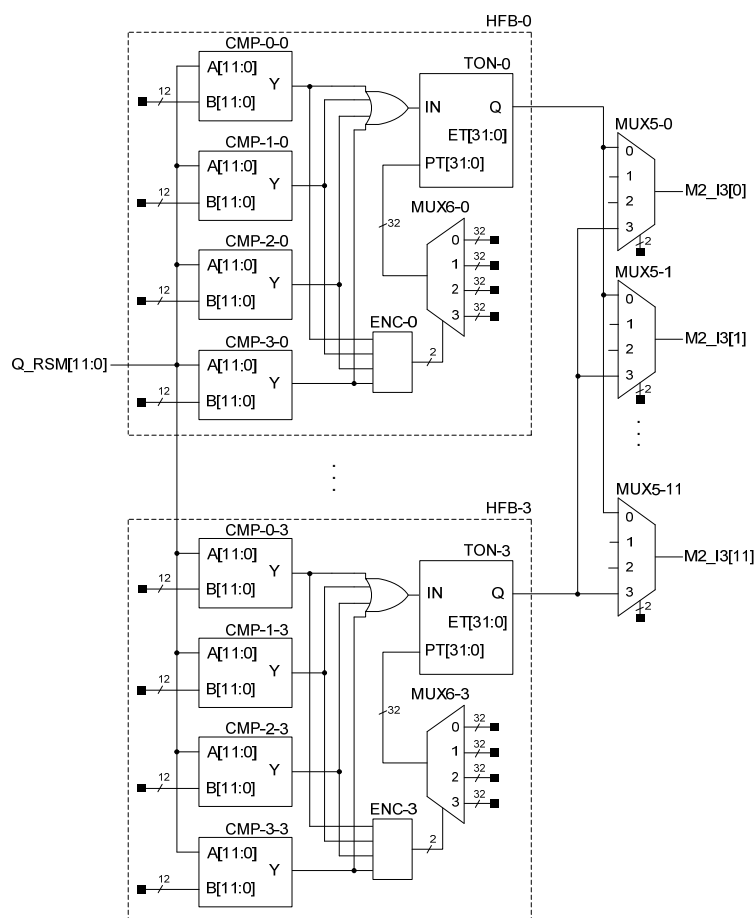


Figure 5. Architecture of TON timer hardware function blocks for the RSM module.

2.5. SEU Detection and Control Circuit

The RSM module also supports the so-called safe implementation of finite-state machines (sequential circuits). When a single-event upset phenomenon occurs, it can affect the state of flip-flops, which can lead to a transition to the invalid state and the malfunction of the control circuit. However, if the one-hot encoding scheme is used to implement a state machine, the aftermath of an SEU event can be relatively simply detected and corrected by the dedicated combinational circuit. This circuit is embedded in the RSM module. It controls the state of all flip-flops involved in the implementation of a state machine (FD-0...FD-11 in Figure 3), and reacts if more than one flip-flop is in the high logic state, or if none is active. Thus, the SEU detection circuit simply counts the number of flip-flops that are activated. If this number is different than one, the control circuit sets the predefined recovery state using the MUX4-0...MUX4-11 in Figure 3. Similar functionality of a safe implementation of finite-state machines is offered by FPGA synthesis and implementation tools, for example [25].

When the SEU detection circuit is activated or the sequential circuit attains certain predefined states configured in the same way as the other configurable bits and words, an interrupt is requested to the CPU. The control block sets the RSM_INT_REQ line (Figure 3) and also sets the IND_JUMP_REG bus to the predefined 20-bit value, which contains the address of the interrupt service routine located in the program memory address space of the CPU.

To facilitate interrupts, the CPU from [22] has been upgraded and supplemented with a basic interrupt system. An interrupt can be requested by an external INT_REQ signal. When the interrupt system is enabled (the dedicated bit within a word located in the global address space is activated), an interrupt causes an immediate jump to the interrupt vector, located at the 00040h address in the memory address space. The contents of key CPU registers, such as the accumulator and the T register [22], are automatically saved, releasing the user from manually performing that task. When the interrupt is serviced, other interrupts are disabled. The dedicated instruction RETI has also been added to the CPU instruction set, facilitating the return operation from an interrupt. This instruction pops the return address from the stack memory, restores the values of key registers, re-enables the interrupt system, and deasserts the external RETI signal. The latter signal is activated, in turn, when the interrupt is acknowledged by the CPU.

Another upgrade of the CPU encompasses the indirect jump instruction. This instruction can now also cause a jump to the address, which is present on the external 20-bit input bus called IND_JUMP_REG. This bus can be driven by the RSM module, which has a bus with the same name (Figure 3).

The RSM can also be configured to assert a predefined value on the IND_JUMP_REG bus if a sequential control circuit reaches certain states or an SEU event occurs. This feature facilitates interrupt service. If the first machine instruction, located at the address of the interrupt vector, is the indirect jump instruction mentioned above, the CPU can execute code directly assigned to a certain state of the state machine implemented by the RSM.

3. Simulations and Implementation Results

The following subsections show essential functional features of the PLC system that include the upgraded CPU and the RSM module. All simulation waveforms presented in this section have been obtained using the Isim simulator integrated with the Xilinx ISE 14.7 environment. The waveforms reflect only the results of the behavioral simulation of the Verilog HDL code describing the RSM module along with the upgraded CPU from [22].

3.1. The RSM Module as a Boolean Coprocessor

Figure 6 shows the simulation results for the PLC system that calculates some Boolean expressions using the CPU and, simultaneously, the RSM module. Signal names without the underscore sign “_” directly represent the external signals and buses of the CPU or the RSM module. The other signals are related to the internal buses of these units. For example, “_PC[19:0]” and “_ACC[31:0]” are the 20-bit program counter and the 32-bit accumulator of the CPU, respectively. All numerical values within the waveforms (inside the green lines) are given in hexadecimal format. The exception is the EXT_OUTPUT[11:0] signal bus, whose state is given as a binary number. The other values (i.e., the time scale values and bus widths) are in decimal format.

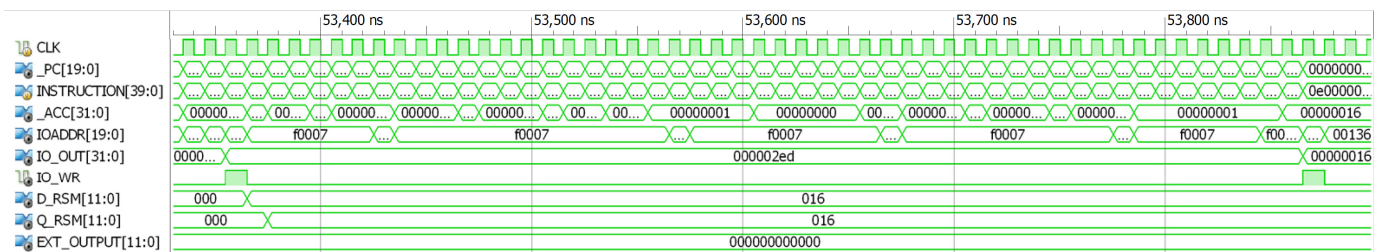


Figure 6. Behavioral simulation of PLC system with the RSM module as a Boolean coprocessor.

A set of the following five Boolean equations is considered for the waveforms in Figure 5:

$$y1 = (x1 * x6') + (x5 * x6'), \quad (1)$$

$$y2 = (x1 * x6) + (x2 * (x7 + x8)') + (x3 * x9), \quad (2)$$

$$y3 = (x2 * x8) + (x3 * (x7 + x9)'), \quad (3)$$

$$y4 = (x3 * x7) + (x2 * x7) + (x5 * x10'), \quad (4)$$

$$y5 = (x5 * x10) + (x5 * x6). \quad (5)$$

Calculation of Equations (1)–(5) constituted the important part of the logic control (LC) test programs applied by [20,22] for CPU performance evaluations.

Listing 1 presents, in turn, a fragment of the assembler code executed by the CPU for the waveforms in Figure 6. The first “STIO” instruction (from the 00102h address) writes the actual parameter values of the Boolean variables $x1$, $x2$ to the specific address located in the global variable address space where the internal input variables for the RSM reside (the INT_INP[11:0] bus in Figure 3). An execution of this instruction by the CPU is indicated by the first appearance of the impulse on the IO_WR line in Figure 6. Then, the CPU starts executing the sequence of instructions implementing the equations. The CPU needs as many as 51 clock cycles to complete the calculation of these equations, whereas the response of the RSM module (the state of the Q_RSM[11:0] line) is available right at the next clock cycle after the execution of the “STIO” instruction mentioned above. This means that, in this case, the RSM works 51 times faster than the CPU.

The result of the calculations performed by the CPU is also written to a global variable (the second “STIO” instruction in Listing 1), resulting in the second appearance of the impulse on the IO_WR line (Figure 6). However, calculations executed by the CPU require a considerably longer amount of time, compared to the RSM module response. This amount of time depends on the complexity of the equations (the number of terms and variables within each equation) and the number of equations that need to be calculated. The bigger these numbers are, the higher the speed advantage of the RSM module, which always needs only one clock cycle to calculate the result.

Listing 1. Fragment of the assembly code executed by the CPU for waveforms in Figure 6.

```

0x00100: LDAL 0x000002ED
0x00101: ST 0xF0007,0x00,0x3
0x00102: STIO 0x00400
0x00103: LDA 0xF0007,0x06
0x00104: ANDN 0xF0007,0x00
0x00105: ST 0xF0007,0x0A
0x00106: LDA 0xF0007,0x02
0x00107: ANDN 0xF0007,0x00
0x00108: OR 0xF0007,0x0A
0x00109: ST 0xF0006,0x00
; ...
0x00134: LDA 0xF0006,0x00
0x00135: STIO 0x00800

```

3.2. The RSM as a Sequential Control Circuit

The primary goal of the RSM module design was to implement some sequential control circuits in hardware, thus offering a much shorter response time than using a CPU, and enabling flexibility similar to that of software implementations. An example of a sequential control algorithm is shown in Figure 7 as a diagram of a finite-state machine. The algorithm involves two inputs k_1 , k_2 , four outputs Z_1 , Z_2 , M , Z_3 , and three instances of the TON timer. It is worth noting that the software implementation of this algorithm using the one-hot encoding scheme was exactly the LC test program applied by [20,22] for CPU performance evaluations. The important part of this test program was the calculation of Equations (1)–(5). These equations, in fact, implement the one-hot encoding scheme for the diagram in Figure 6, where $x_1 = A$, $x_2 = B$, ... $x_5 = E$, $x_6 = k_1$, $x_7 = k_2$, $x_8 = T1.Q$, $x_9 = T2.Q$, $x_{10} = T3.Q$, and $y_1 \dots y_5$ are the next states for the A...E state variables, respectively.

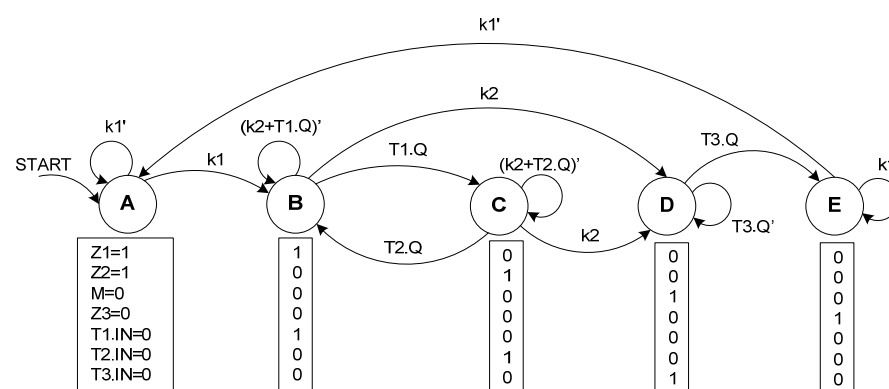
**Figure 7.** Diagram of finite-state machine describing a sequential control algorithm.

Figure 8 shows the simulation waveforms for the RSM module configured as a sequential circuit that implements the diagram shown in Figure 7. All values within the waveforms are hexadecimal. The RSM starts the control circuit (sets the initial state of the state machine) when the CPU sets a certain global variable, which is indicated in the waveforms as the subsequent pulses on the IO_WR and _INIT_RSM lines. The k_1 and k_2 inputs in Figure 7 are assigned to the two least significant bits of the EXT_INPUT[11:0] bus. The outputs Z_1 , Z_2 , M , and Z_3 are, in turn, present on the consecutive bits of the EXT_OUTPUT[11:0] bus, starting from the least significant bit. The time constants for the TON T1, T2, and T3 instances were set to 10, 10, and 20 clock cycles, respectively.

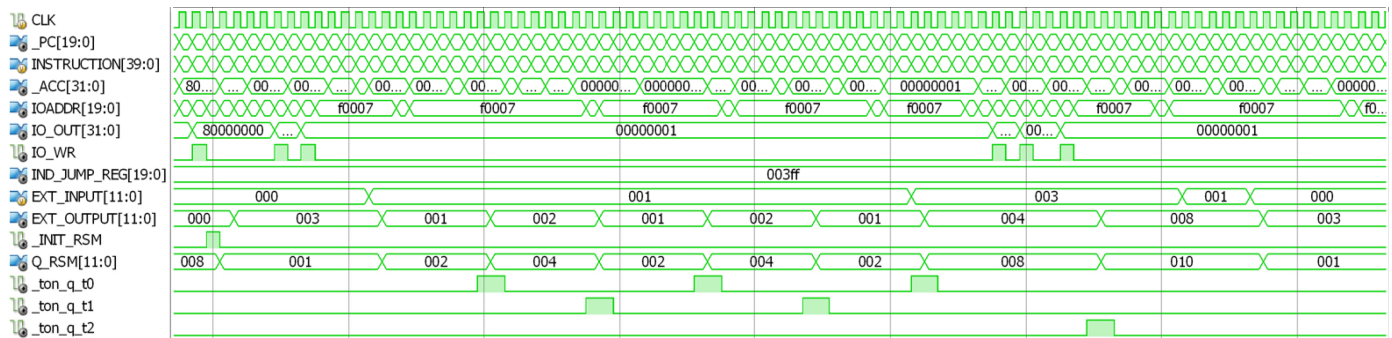


Figure 8. Behavioral simulation of the PLC system with the RSM module implementing the FSM from Figure 7.

It is worth noting that the RSM and the CPU work independently of each other, which can be seen in Figure 8. The CPU executes every instruction within a single clock cycle, which is also worth stressing. However, the RSM can also be configured to interact with the CPU when certain events occur, such as when the state of the Q_RSM[11:0] bus reaches one of the predefined values. An example of this interaction is shown in Figure 9.

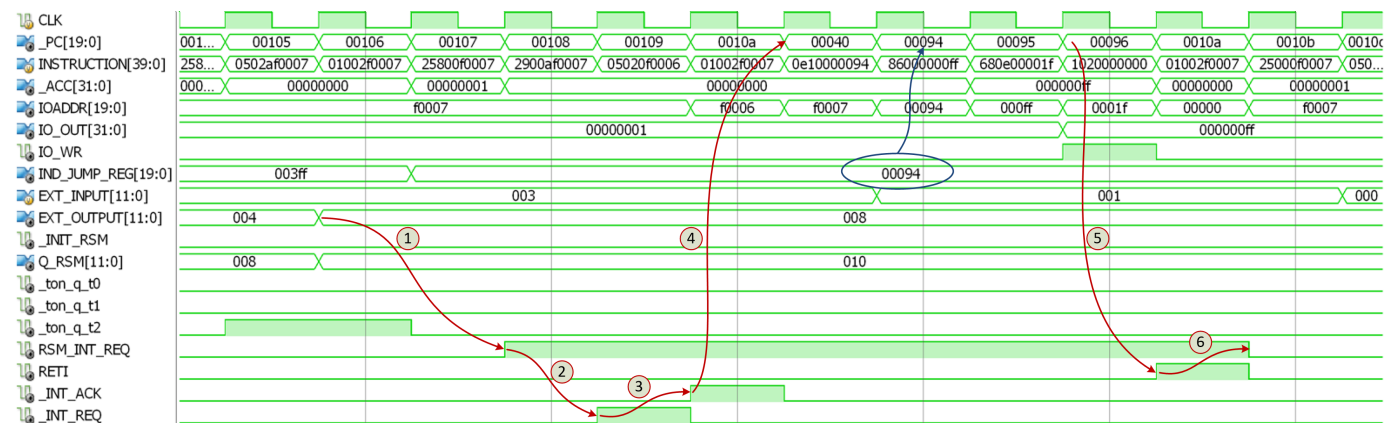


Figure 9. Simulation of the interrupt feature of the PLC system with the RSM module.

The 008h value that appears on the Q_RSM[11:0] line causes the RSM module to request an interrupt (the arrow with the number 1 in Figure 9, which leads to the activation of the RSM_INT_REQ signal). In the next clock cycle, the request is detected by the CPU (the arrow with number 2) and acknowledged (arrow 3), which causes the program counter of the CPU to be loaded with the interrupt vector value (arrow 4).

The first instruction located at the address of the interrupt vector (00040h) is the indirect jump instruction (the 0E10000094h code on the INSTRUCTION[39:0] bus), which causes the jump to the address that is issued by the RSM module on the IND_JUMP_REG[19:0] bus (00094h in this case). This address corresponds directly to the value on the Q_RSM[11:0] bus, which causes the interrupt. Address assignments to the corresponding Q_RSM[11:0] values can be made during the configuration of the RSM module. For the current version of the RSM module, up to 13 pairs of these can be defined. The return from interrupt instruction (arrow 5) sets the RETI line of the CPU, which causes deactivation of the RSM_INT_REQ line in the next clock cycle.

The RSM module can exhibit similar behavior when an SEU event occurs (if configured to react to an SEU). An example of this kind of event is shown in simulation waveforms presented in Figure 10. The change of the value on the Q_RSM[11:0] bus from 008h to 00Ch (more than one flip-flop within a state memory, for the one-hot encoding, becomes

active) causes an immediate activation of the internal `_SEU_DT` signal (arrow 1), which indicates that an SEU just happened. In the next clock cycle, the 020h recovery state is established on the `Q_RSM[11:0]` bus, and an interrupt is requested to the CPU (arrow 2). For the considered example configuration of the RSM, the transition from the recovery state to a regular state can happen if the CPU sets a certain Boolean global variable. This is shown in simulations in Figure 10, where the penultimate instruction before return from the interrupt writes a value at the I/O address 00400h, which ultimately leads to the change on the `Q_RSM[11:0]` bus (arrow 3) and a restart of the control circuit implemented by the RSM.

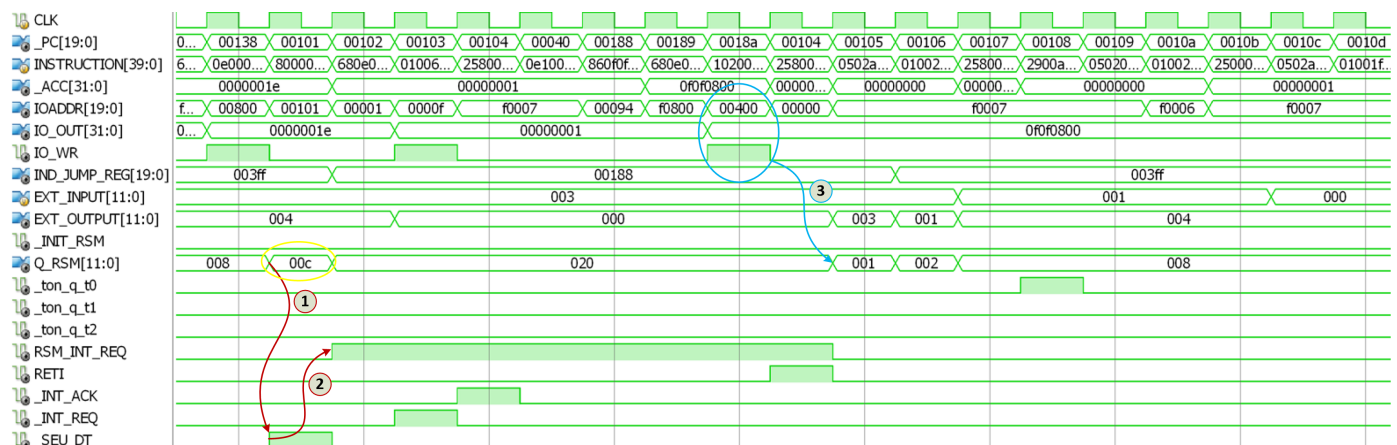


Figure 10. Simulation of an SEU event.

However, in the real-world scenario, an SEU event can also affect the CPU circuitry, or the other circuitry of the RSM module. In this case, the CPU would probably not be able to respond and acknowledge the interrupt, or the functionality of the RSM module could be disrupted. Therefore, the example considered illustrates only a potential feature of the proposed RSM module.

3.3. Implementation

The architecture of the RSM module has been specified using Verilog HDL. The post-place and route results of the implementation of the RSM for the AMD/Xilinx Artix-7 FPGA family chip (the XC7A100T-3CSG324 available on the Digilent Nexys A7 evaluation board) are presented in Table 1. Xilinx ISE 14.7 software was used as the synthesis and implementation tool. The balanced optimization strategy was applied for this implementation.

Table 1. Utilization of FPGA resources for the AMD/Xilinx XC7A100T chip.

Unit	LUTs	FFs	Max Clock Freq. [MHz]
RSM	816 (1.29%)	1175 (0.93%)	160.2
CPU + RSM + CM	4301 (6.78%)	3165 (2.49%)	73.7

The results encompass the standalone RSM module as well as the basic PLC system, including the upgraded CPU from [22], the RSM module, and the simple communication module (CM). The latter module was a very simplified version of a similar module applied in [20]. It allowed only for downloading the program code to the program memory of the CPU and for the configuration of the RSM module. The implementation results revealed that the FPGA resources requirement, in terms of the number of look-up tables (LUTs) and

flip-flops (FFs), for the RSM module is relatively low, around 1% of the available resources for the targeted chip. It is worth noting that, unlike for the CPU, the number of required FFs is substantially larger than the number of LUTs. This is due to the high number of configuration bits needed by the RSM (every configuration bit requires a single FF). The maximum clock frequency that can be applied to the RSM module is rather high, more than two times higher than that for the standalone CPU from [22] or the overall PLC system considered in Table 1. The latter system applied, in fact, a common clock signal for both the CPU and the RSM. Therefore, the performance of the RSM in this system is reduced due to the application of a lower clock frequency than the maximum value.

The utilization of FPGA resources of the RSM module constitutes only 19% of LUTs and 37% of FFs required by the whole system considered in Table 1. However, if a higher number of Boolean equations with a higher number of variables is needed or more than one RSM module is used in the system, the number of FPGA resources required by the RSM will be higher.

As far as performance evaluation is concerned, Table 2 shows the calculation times of the logic control algorithm described by the finite-state machine from Figure 7 for different controllers, including high-performance PACs. These times cover exclusively the internal algorithm execution time, excluding delays of communication, input/output updating, etc. The detailed methodology of these measurements is described in [22]. As the results indicate, the execution time of the control algorithm implemented by the RSM module, being a part of the PLC system clocked at 72 MHz, took only 13.9 ns (the period of one clock cycle). Thus, the PLC system with the RSM module was almost 18 times faster than the fastest controller considered in Table 2. It is worth noting that the advantage of this type of system is greater the more complex the control algorithm. This stems from the very nature of the RSM module as a hardware realization of the algorithm described by Boolean equations or a finite-state machine.

Table 2. Execution times for the test algorithm.

Controller	Time [μ s]
[20] Xilinx Spartan-6 @40 MHz	11.5
[22] Xilinx Artix-7 @74.1 MHz	1.55
Beckhoff CP6607 @533 MHz	1.32
Beckhoff CX4140 @1.91 GHz	0.25
RSM module Xilinx Artix-7 @72 MHz	0.0139

4. Conclusions

A new approach for the performance improvement of FPGA-based PLC controllers was proposed in this article. The approach is to integrate with a PLC system a configurable hardware module (one or multiple modules), called the RSM, that is capable of implementing combinational or sequential logic control algorithms on a hardware basis. The module can operate in parallel with the CPU. The key idea behind the RSM module is that the actual functionality of the module is not statically set during FPGA project implementation by FPGA synthesis tools, but can be established (configured) at any time during FPGA-based PLC system operation. So far, similar solutions have not been proposed in the literature.

The RSM module presented in this article is capable of implementing up to 12 Boolean functions, where each function can have up to 12 variables. It can also implement a finite-

state machine with up to 12 states for the one-hot encoding scheme. If the full encoding is used, potentially up to 4096 states can be implemented. However, in this case, the actual number of states is limited by the number of involved variables, which cannot exceed 12 variables in a next-state function for the single-state variable. Yet, the architecture presented for the RSM module is, in fact, scalable. Thus, the number of equations and the number of variables can easily be increased, which would also entail an increase in the utilization of FPGA resources. It is also worth noting that the utilization of FPGA resources for the RSM module does not depend on the implemented Boolean functions or the implemented finite-state machine, as long as they do not exceed the maximum capacity of the particular configuration of the RSM module (i.e., the maximum number of equations, input variables, and states).

Using the RSM module, some control algorithms can be executed within a single clock cycle, thus much faster than even for high-performance PLCs/PACs, whose CPUs still require many machine instructions (and clock cycles) for an implementation of the same algorithm. The RSM module offers execution speed at the same level as hardware implementations, maintaining the programming flexibility similar to software solutions.

Apart from the limited number of variables or states offered by the RSM module, its disadvantage is also that the control algorithms implemented by the RSM can only be expressed as Boolean equations or as finite-state machine graphs. The programming languages defined by the IEC61131-3 standard cannot be applied directly. Dedicated software is also needed for the conversion of finite-state machine graphs or Boolean equations into configuration words for the RSM. However, the RSM can find applications in cases where very fast implementation of logic control algorithms is needed, along with other non-critical tasks that can be executed by the CPU.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. IEC 61131-3:2025; Programmable Controllers—Part 3: Programming Languages. International Electrotechnical Commission (IEC): Geneva, Switzerland, 2025.
2. Cancino-Escobar, M.; Delgado-Del-Carpio, M.; Solís-Cisneros, H.I.; Mota-Grajales, R.; Hernández-Gutiérrez, C.A. Low-Latency FPGA-Based PLC Microprocessor for Industrial Automation in Compliance with IEC-61131-3. *Arab. J. Sci. Eng.* **2024**, *49*, 16407–16420. [\[CrossRef\]](#)
3. Dave, D.; Raninga, P. Implementation of PLC Using FPGA. *Int. J. Sci. Eng. Res.* **2016**, *4*, 72–73. [\[CrossRef\]](#)
4. Bergmann, N.; Waldeck, P.; Shukla, S. FPGA Implementations of Ladder Diagrams. *Mod. Appl. Sci.* **2013**, *7*, 64–73. [\[CrossRef\]](#)
5. Corti, G.; Brunner, D.; Mizuno, N.; Jamieson, P. Transforming Ladder Logic to Verilog for FPGA Realization of Programmable Logic Controllers. In *Proceedings of the International Conference on Embedded Systems, Cyber-Physical Systems, and Applications (ESCS), Las Vegas, NV, USA, 17–20 July 2017*; Arabnia, H.R., Deligiannidis, L., Tinetti, F.G., Eds.; CSREA Press: Las Vegas, NV, USA, 2018; pp. 35–38.
6. Subbaraman, S.; Patil, M.; Nilkund, P.S. Novel integrated development environment for implementing PLC on FPGA by converting to synthesizable VHDL code. In *Proceedings of the International Conference on Control Automation Robotics & Vision, Singapore, 7–10 December 2010*; IEEE: Piscataway, NJ, USA, 2010; pp. 1791–1795. [\[CrossRef\]](#)
7. Kumar, A.; Kumar, P.; Modi, S.; Nath, V. Study and Implementation of Ladder Logic Conversion to VHDL for Field Programmable Gate Array (FPGA)-Based Programmable Logic Controllers (PLC). In *Proceedings of the Fourth International Conference on Microelectronics, Computing and Communication Systems, Ranchi, India, 11–12 May 2019*; Nath, V., Mandal, J.K., Eds.; Springer: Singapore, 2020; Volume 673, pp. 31–42. [\[CrossRef\]](#)
8. Aspar, Z.; Rahman, N.H.A. Implementation of a PLC Processor Architecture with LLD as Native Language on FPGA and ASIC. In *Proceedings of the IEEE International Conference on Electrical, Control and Computer Engineering (InECCE), Kuantan, Malaysia, 27–28 August 2025*; IEEE: Piscataway, NJ, USA, 2025. [\[CrossRef\]](#)

9. Milik, A.; Kubica, M.; Kania, D. Reconfigurable Logic Controller—Direct FPGA Synthesis Approach. *Appl. Sci.* **2021**, *11*, 8515. [\[CrossRef\]](#)
10. Shedge, S.; Tade, S.L. Implementation of IEC 61131-3 Standard Compatible Instruction List Processor on FPGA Platform. *Int. J. Adv. Res. Innov. Ideas Educ.* **2018**, *4*, 1220–1227.
11. Delgado-del-Carpio, M.; Hilario-Tacuri, A.; Hernández-Gutiérrez, C.A. Modeling and Implementation of a Specific Microprocessor to Enhance the Performance of PLCs Employing FPGAs. *J. Electron. Electr. Eng.* **2023**, *2*, 171–185. [\[CrossRef\]](#)
12. Czerwiński, R.; Chmiel, M.; Wygrabek, W. FPGA Implementation of Programmable Logic Controller Compliant with EN 61131-3. *IFAC Proc. Vol.* **2013**, *46*, 138–143. [\[CrossRef\]](#)
13. Chmiel, M.; Kulisz, J.; Czerwiński, R.; Krzyżyk, A.; Rosol, M.; Smolarek, P. An IEC 61131-3-Based PLC Implemented by Means of an FPGA. *Microprocess. Microsyst.* **2016**, *44*, 28–37. [\[CrossRef\]](#)
14. Chodorowski, P.; Chmiel, M. IEC 61131-3 Compliant PLC Structure Based on FPGA Multi-Core Solution. In *Proceedings of the International Conference on Signals and Electronic Systems (ICSES), Krakow, Poland, 5–7 September 2016*; IEEE: Piscataway, NJ, USA, 2016. [\[CrossRef\]](#)
15. Eassa, H.; Adly, I.; Issa, H.H. RISC-V Based Implementation of Programmable Logic Controller on FPGA for Industry 4.0. In *Proceedings of the IEEE International Conference on Microelectronics (ICM), Cairo, Egypt, 15–18 December 2019*; IEEE: Piscataway, NJ, USA, 2019; pp. 98–102. [\[CrossRef\]](#)
16. Waterman, A.; Lee, Y.; Patterson, D.A.; Asanović, K. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*; University of California: Berkeley, CA, USA, 2014.
17. Jamieson, P.; Blank, D.; Ghanem, J.; McGrew, T.; Corti, G. A Methodology for an FPGA Implementation of a Programmable Logic Controller to Control an Atomic Layer Deposition System. *Int. J. Reconfig. Comput.* **2022**, *2022*, 8827417. [\[CrossRef\]](#)
18. Chmiel, M.; Czerwiński, R.; Malcher, A. FPGA Implementation of IEC 61131-3-Based Hardware-Aided Timers for Programmable Logic Controllers. *Electronics* **2023**, *12*, 4255. [\[CrossRef\]](#)
19. Chmiel, M.; Czerwiński, R.; Malcher, A. FPGA Implementation of IEC-61131-3-Based Hardware-Aided Counters for PLC. *Appl. Sci.* **2021**, *11*, 10183. [\[CrossRef\]](#)
20. Hajduk, Z.; Trybus, B.; Sadolewski, J. Architecture of FPGA Embedded Multiprocessor Programmable Controller. *IEEE Trans. Ind. Electron.* **2015**, *62*, 2952–2961. [\[CrossRef\]](#)
21. Rzońca, D.; Sadolewski, J.; Stec, A.; Świder, Z.; Trybus, B.; Trybus, L. CPDev Engineering Environment for Control Programming. In *Trends in Advanced Intelligent Control, Optimization and Automation*; Mitkowski, W., Kacprzyk, J., Oprządkiewicz, K., Skrush, P., Eds.; Springer: Cham, Switzerland, 2017; pp. 303–314.
22. Hajduk, Z. IEC61131-3 Instruction List Language Processor for FPGAs. *Electronics* **2023**, *12*, 4052. [\[CrossRef\]](#)
23. Raspberry Pi Ltd. *RP2040 Datasheet: A Microcontroller by Raspberry Pi*; Document RP-008371-DS, Rev. 2025-02-20 [0.6.3]; Raspberry Pi Ltd.: Cambridge, UK, 2025. Available online: <https://pip.raspberrypi.com/documents/RP-008371-DS-rp2040-datasheet.pdf> (accessed on 12 September 2026).
24. AMD. *Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide*; Technical Report UG953, Version 2026.1; AMD: Santa Clara, CA, USA, 2026.
25. Xilinx. *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*; Technical Report UG687, Version 14.5; Xilinx: San Jose, CA, USA, 2013.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.