Reconfigurable Hardware Technology: an Emerging Paradigm for Combined Software-Hardware Fault-tolerance Implementation

Cs. Szász

Electrical Machines and Drives Department, Technical University of Cluj (e-mail: Csaba.Szasz@emd.utcluj.ro).

Abstract: As is well known complex safety digital systems currently being designed and developed are often difficult multidisciplinary undertakings. In order to achieve their operation even under extreme conditions is important to strengthen it with fault-tolerant behaviors. One of the safest solutions is to provide these digital systems both with software and hardware redundancy. This paper is dedicated to emphasize the benefits and advantages of reconfigurable hardware technology application in combined software-hardware redundancy strategies implementation. This technology it is considered as one of the most challenging design paradigms in modern digital systems development. Arguments to support it in fault-tolerant systems development are widely discussed, as well to exploit its fine-grained parallel and distributed computing behaviors with huge re-routing abilities are suggested. An intuitive example of how to design and implement combined software-hardware redundancy and achieve high level of fault-tolerance is presented in detail. Beside the research related to the fault-tolerant digital systems development recommendations and conclusions are formulated regarding the future of the reconfigurable hardware technology paradigm in high reliability systems implementation.

Keywords: reconfigurable architectures, field programmable gate arrays, redundancy, software reliability, fault tolerance.

1. INTRODUCTION

According to a general rule definition, a fault-tolerant system is one that continues to operate safety and to properly perform its prescribed tasks even in the presence of faults occurring in hardware or software components. Fault-tolerant systems research covers a wide spectrum of applications, ranging from high reliability manufacturing systems, energy distribution networks, nuclear power plants, transportation systems, military/space systems, healthcare industry, factories, telecommunication, defense systems, air traffic control, and many more. Such systems usually are named critical processes where the occurrence of faults may cause inestimable damages and losses in human life or capital. Nowadays due to increase of dependability and demand, the complexity of digital systems has grown up exponentially. To avoid unwanted errors or failure states, the reliability of hardware and software should be improved with high reliability behaviors and abilities. In vast majority of applications fault-tolerance is achieved by using redundancy implementation strategies (hardware, software, time, or information redundancy). Therefore, redundancy is a very common approach to improve reliability and a widely spread technology of implementing fault-tolerant systems. However, with the use of last generation microelectronic technologies and available high performance software techniques many applications do not need redundancy. On the other hand if the cost of a failure is high enough, the use of redundancy is inevitable and even increasing the design costs and complexity of the systems becomes an attractive option (Suleka, 2011; Chielle, 2016). During last few decades the fault-tolerant systems design topic emerges as one of the most important and challenging research field for electrical engineers. A myriad of high quality research and scientific literature has been published worldwide in this area. Of course, there is not enough room to include a complete survey of the entire topic or to perform an exhaustive presentation. There only a short overview is presented regarding the most relevant achievements that support the objectives and goals of this paper. First of all, the necessity of developing a generalized method to tolerate both hardware and software fault has been outlined only in the recent past. In (Giadomenico et al., 1995) it has been proposed a unique approach for hardware and software fault tolerance. A similar point of view is also shared in (Lyu, 1995; Yang 2017). The paper (Levitin, 2006) presents the reliability and performance analysis of both hardware and software systems, in (Wattanapingskom et al., 2002) it is discussed the fault-tolerant embedded system design and optimization. Fault-tolerant system structures have been analyzed in the research works (Belli et al., 1990; Wu et al., 1997). Cost expenses calculation in modeling fault-tolerant software and hardware are described in (Bondavialli et al., 1993; McAllister and Scott, 1991). An improved fault-tolerant system using checkpoints legacy code is presented in (Leach, 2008), respectively a fault-tolerant system is using grid computing technique in (Khan et al., 2010; Poledna, 1994). A novel model for software reliability growth and optimal design of *N* version software is introduced in (Teng and Rhan, 2002). In (Yamachi, et al., 2006) has been developed a genetic algorithm for solving *N*-version program design problem. The recovery block concept in software fault-tolerance has been introduced by (Randell and Xu, 1995) and optimization models for component based recovery blocks were proposed in (Berman and Kumar, 1999). Optimal allocation of redundant components for series and parallel connection has been analyzed in (Belzence et al., 2011). Other important contributions to modeling fault-tolerant systems and to develop original hardware and software redundancy techniques are also described in (Kumar et al., 1986; Valdes and Zequeira, 2006). A general view for software fault-tolerance implementation is introduced in (Lyn, 1995), respectively in (Dugan and Lyu, 1995) are discussed methods for fault-tolerance implementation with replication. In (Yang and Meng, 2011) it is described a self-repairable system using warm standby redundancy strategy. Various researchers have developed software reliability models and software debugging methods tested both theoretically and experimentally (Lewis, 2011; Yamachi, et al., 2006, Sinca and Szász, 2017; Sari and Akkaya, 2015). Each of the above mentioned high quality researches addresses the key problems of both hardware and software redundancy implementation. Each one added something new and original to the topic of fault-tolerant systems development for high reliability applications. However, this paper it is dedicated to outline the emergence of a novel fault-tolerance implementation paradigm based on the hardware reconfigurable technology. The originality of this presentation lies on the recognition that this advent represents not only a specific view point but a novel approach regarding the combined software-hardware fault-tolerance implementation in modern digital systems.

2. RECONFIGURABLE HARDWARE TECHNOLOGY DESIGN PARADIGM

As is well known, traditional approaches of digital systems design and development operates with logical descriptions of the used components. In essence, these are implicit descriptions contained in schematic diagrams, Boolean equations, block diagrams, or wiring lists that form the components of the design. By using these traditional design methods a relatively clear distinction can be made between hardware and software. In more recent years, with the advent of programmable logic and associated technologies the classical techniques gradually have been replaced with hardware description languages (HDL). This approach incorporates a high level of abstraction where the designs are created in a lexical format that describes the essence of the design in structural, functional, or behavioral form (Scarpino, 1997). When functional performance has been verified, the logical description of the design is overlaid onto a hardware implementation. At current level of microelectronic technologies the programmable logic arrays are the most adequate chips for such implementations. These are prefabricated logical arrays within which electronic interconnections may be either enabled or disabled according to various user needs. Among these devices excels the FPGA (Field Programmable Logic Array) processors, as the top technical achievements in this field. Current trends in digital systems design and development emphasize the implementations using HDL and FPGA chips embedding reconfigurable hardware technology. This approach affords to designers more degree of freedom for efficient system presentation, as well as for versatile implementation details. Additionally, HDL is well suited for rapid design and prototyping by blurring the traditional border between software and hardware (Scarpino, 1997; Sharma, 2012). This new design paradigm also posses the advantage of a huge efficiency and provides the flexibility of general hardware approach.

As it has been mentioned before, the top representing chips of this technology are the FPGAs. They possess the ability of parallelization and parallel computing, being ideally suited for distributed tasks solving or network computing applications. Toward, exploits the advantages of the fine-grained instruction level parallelism as well coarse-grained functional parallelism. Their immense computational efficiency is matched by rich on-chip interconnectivity and high bandwidth concurrent memory access, achieving huge re-routing abilities that abstract the implementation details (Husi et al., 2014; Rink and Castrillon, 2017; Du et al., 2015). By allowing multi-grid computation, FPGAs are ideal platform for fine-grained parallel computing, representing the perfect solution with which to implement highly concurrent control offering the huge advantages of flexibility, low-power consumption, speed, adaptability, and case of scale. The great majority of researchers and scientists involved in microelectronic technologies development generally agree that HDL combined with reconfigurable hardware paradigm represents the future of circuit and device design.

3. FAULT-TOLERANT HARDWARE DESIGN PARADIGM

The fault-tolerant hardware design paradigm means the use of additional hardware components or physical modules in order to achieve fault-tolerance that are unnecessary for a fault-free operation of the considered system. This solution always means extra costs, size, and weight, but as microelectronic components have become smaller and less expensive the hardware redundancy design paradigm becomes more practical (Shin, 2016).

In the related scientific literature there are distinguished three basic form of hardware redundancy: passive, active, and hybrid (as combination of the first two). Shortly defined, passive redundancy achieves fault-tolerance by masking the fault that occurs without requiring any action from the operator's side. Simply mask the faults and do not attempt to provide for its detection. Active redundancy achieves fault-tolerance by detecting the faults which occur, locates them, and performs recovery actions in order to restore the initial fault-free state of the system. Active redundancy is also named as dynamic redundancy method. Of course, the hybrid redundancy combines the advantages of both the passive and active approaches (Sari and Akkaya, 2015; Rampratap 2016).

However, the most common form of passive hardware redundancy is triple modular redundancy (TMR). There three perfectly identical modules perform the same functions and tasks inside the digital system with a majority decision element determining the output of the system. If one of the modules enters into failure state the voter will mask the fault by recognizing the result of the two remaining fault-free units as correct (Johnson, 1989; Coulouris, 2011). In this article the design method relying on the TMR strategy by using the reconfigurable hardware technology paradigm will be presented and discussed. For this reason it is considered the block diagram shown in Fig. 1. There are three identical FPGA-based development boards (*Module_1, Module_2, and Module_3*) embedding hardware reconfigurable technology.. Each module delivers the processed results via its own output bus labeled O_1, O_2, and O_3 in the figure. These are input signals of the voter element (or decision unit) which performs a majority voting strategy over the inputs. Therefore, the occurred fault remains masked inside system and the proper result is released to the output of the voter element via the bus labeled *Y*. If it is considered that all the hardware modules operates correctly (without faulty) by emitting the signals O_i $(i=1\div3)$ to the inputs of the voter, the operation of the TMR system it is described by the logical equation:

$$Y = O_1 \cdot O_2 + O_2 \cdot O_3 + O_1 \cdot O_3 \tag{1}$$

It is known that the reliability $R(t)$ of a system is function of time and it is expressed by the probability that the system will operate correctly throughout the time interval [0, t], assuming that was performing correctly without any faults or errors at time $t=0$. In other words, reliability is the probability that the system will not fail by a given time t, under a given set of imposed operation conditions. In contrast, the probability of failure by a given time t is referred to as the unreliability of the system. However, considering the individual system components or modules reliability it is possible to mathematically deduce the system global reliability coefficient.

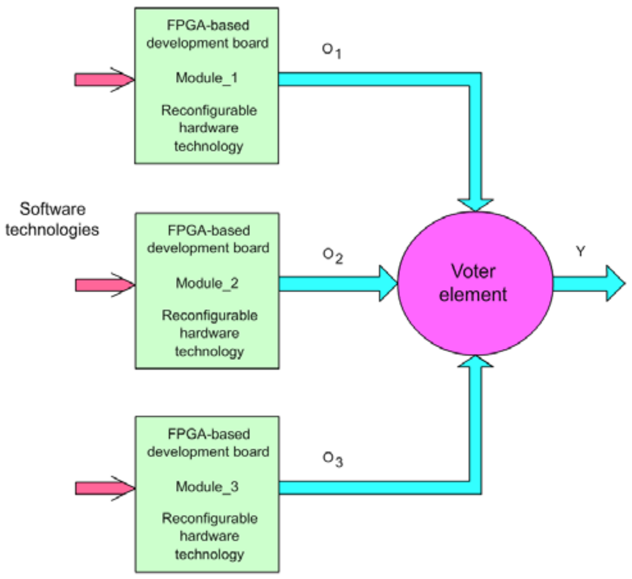

FPGA-based development board
Module_1
Reconfigurable hardware technology
O1
Software technologies
FPGA-based development board
Module_2
Reconfigurable hardware technology
O2
Voter element
Y
FPGA-based development board
Module_3
Reconfigurable hardware technology
O3

Fig. 1. Block diagram of the reconfigurable technology-based TMR implementation strategy.

By using the well known mathematical relations expressing the global reliability of a system with n modules connected in a serial configuration (Johnson, 1989),

$$R(t) = 1 - \prod_{j=1}^{n} (1 - R_j(t)) \tag{2}$$

respectively in parallel,

$$R(t) = \prod_{j=1}^{n} R_j(t) \tag{3}$$

where $R_j(t)$ means the reliability of each component module of the system $(j=1\div3)$, it is possible to calculate the global reliability of the TMR digital system shown in Fig. 1. If the reliability of a single FPGA-based hardware module is labeled with R_M, results the following equation:

$$R_{TMR} = R_M^3 + 3 \cdot R_M^2 \cdot (1 - R_M) \tag{4}$$

where are included all the possible operation modes of the considered system (when all modules operates without any fault, respectively the three situations when one module is in a faulty state and the remaining two operates correctly).

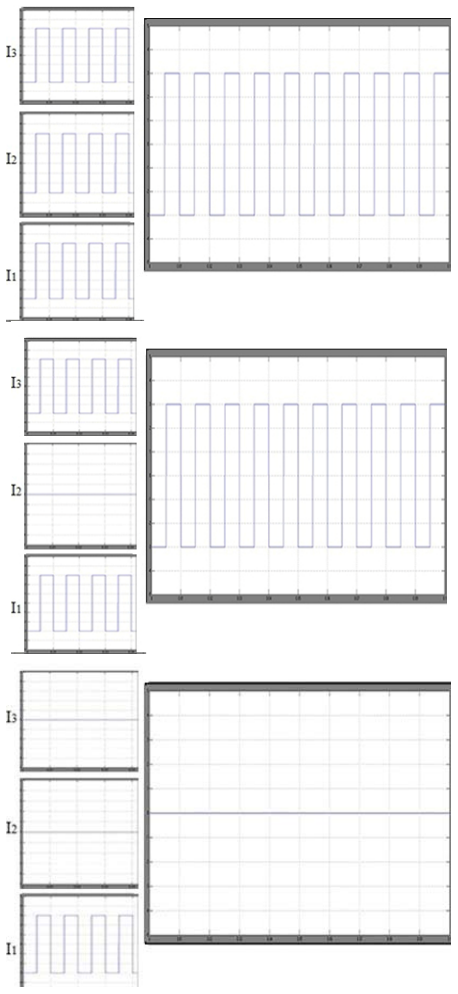


Fig. 2. The TMR digital system operation mode (Sinca and Szász, 2017).

A very convenient method to illustrate the TMR system operation principle is to use waveform diagrams. For this reason in Fig. 2 it is plotted a set of simulation results performed in Matlab/Simulink software environment (Sinca and Szász, 2017). In the top of the figure is shown the situation when all the three modules operates without any fault. In this case the voter element receives the I_1, I_2, and I_3 input signals and passes to the output the same digital signal (plotted on the right side of the diagram). Under is the case when *Module_2* enters into a failure state and the voter element recognizes the result of the two remaining fault-free units as correct (Sinca and Szász, 2017). A very similar situation will be obtained corresponding to failure of *Module_3*. Obviously, when two simultaneous error occurs (both *Module_2* and *Module_3* are faulted) the topology is unsuitable to detect or distinguish them. Therefore, the TMR can't handle such failure states and should be replaced with other more reliable hardware redundant architectures. At the same time, is necessary also to mention here that the probability of instantaneous faults of two different modules is low in practice.

4. FAULT-TOLERANT SOFTWARE DESIGN PARADIGM

In general terms, software fault tolerance is the ability of software to detect and recover from an unexpected faulty or failure state. Software faults always are the result of human designer errors in interpreting a specification or correctly implementing an algorithm. Among the most commonly used methods to design and implement fault-tolerant software should be mentioned here the recovery blocks technique, the self-checking method, and the *N*-version implementation strategy. The recovery blocks method is a simple solution to achieve fault-tolerance which operates with an arbitrator confirming the results of various implementation of the same algorithm (Lyn, 1995; Randell and Xu, 1990,). In this case the entire system is constructed on fault-tolerant blocks and the arbitrator determines the correctness of various blocks operation mode. The recovery block method put emphasis on the specification part of the problem by pressuring for creation of different multiple functional alternatives for the same application. One other suitable method to implement fault-tolerance is the self-checking software. This is not a rigorously described method in the literature which embeds extra checks solutions as well as check-pointing and rollback recovery methods added in high reliability systems. The weak point of this method consists on its lack of rigor with potential surprising events and effects.

Perhaps the most popular method to achieve software fault-tolerance is the *N*-version software implementation strategy. This solution is in fact the software application of the well known *N*-way hardware redundancy. It means the implementation of the same task with *N* modules, where each module is made with *N* different solutions. They spread the utilization of different software languages, application of different software technologies, or programming the same task by using different algorithms. In this way it is encouraged the diversification as much possible, including different toolsets, design strategies, or different software environments. Independently developed software versions also provide tolerance to software design faults. In this paper a versatile solution to implement *N*-version software redundancy is presented and discussed. Notwithstanding with many other solutions presented in international references, there it is emphasized a fault-tolerant software design paradigm relying on theoretical basics and support of the hardware reconfigurable technology. The main idea of this proposal is shown below in Fig. 3.

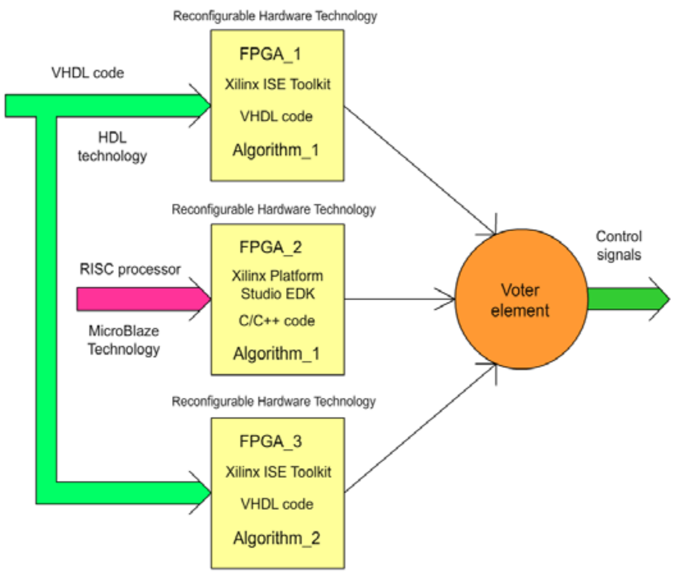


Fig. 3. The N-version software redundancy implementation strategy.

Before to discuss this implementation concept, let is assume first that the hardware redundant digital system given in Fig. 1 should execute a well-aimed user defined control task. This task will be programmed by using an adequate algorithm. In order to achieve a fault-tolerant software implementation, here two very different software technologies will be addressed. The first is the HDL technology discussed above in the paragraph 2. This is characterized by a high level of abstraction with designs created in lexical format describing structural and functional behaviors. The HDL was specially developed for reconfigurable hardware technologies implementation and in our case will be implemented in VHDL (VHSIC Hardware Description Language) code. By following the *N*-version software redundancy implementation concept, the same control task is programmed via two independently developed different algorithms (labeled *Algorithm_1* and *Algorithm_2* in the figure). One algorithm will be uploaded on the FPGA-based development board labeled here *FPGA_1*, the second on *FPGA_3*. These algorithms are developed by using the same Xilinx ISE software toolkit. With the purpose to implement another redundancy level, the same control task now will be programmed by using a second technology. For this scope it is preferred the most evaluate MicroBlaze technology embedded in the Xilinx Platform Studio EDK (Embedded Development Kit) software toolkit (Digilent Co., 2016). This provides a convenient physical environment for embedded processing applications by allowing the utilization of the MicroBlaze processor-based technology and EDK/SDK (Software Development Kit) environment. In essence, the MicroBlaze is a 32 bit

Wishbone compatible full-featured and FPGA optimized RISC (Reduced Instruction Set Computer) soft processor for use in FPGA designs applications. This platform allows the implementation of single-processor or two-processor based systems in MicroBlaze technology by using the *C/C++* language environment. Therefore, the control task developed in this technology will be also uploaded on the development board labeled *FPGA_2*. It is important to notice that at the end a versatile *N*-version redundancy has been reached by using two different software technologies (HDL and MicroBlaze), two software environments (VHDL and C/C++ code), respectively two different types of algorithms (*Algorithm_1* and *Algorithm_2*). Jointly they form a high performance software redundancy system, as well as represent an original and modern approach of the fault-tolerant software design paradigm.

5. RESULTS AND DISCUSSION

Having clarified the main theoretical aspects regarding the used design paradigms, in-depth research efforts was directed to experiment the fault-tolerant digital system embedding combined software-hardware redundancy. In the first stage the hardware redundant system has been implemented and tested. A general view of the used laboratory setup is shown in Fig. 4. There for the FPGA-based development boards three stand-alone Spartan-3E Starter Kit units has been used. The key features of a Spartan-3E Starter Kit board are as follows: Xilinx XC3S500E Spartan-3E FPGA processor with 232 user I/O lines, Xilinx 4 Mbit Platform Flash, 64Mbyte DDR SDRAM, 16Mbyte of parallel NOR Flash, 16Mbits of SPI serial Flash, 2x16 character LCD, PS/2 mouse or keyboard port, VGA display port, 10/100 Ethernet PHY, two RS232 ports, 50MHz clock oscillator, Hirose FX2 expansion connector, three Digilent 6-pin connectors, 4 channel D/A converter, 2 channel A/D converter, rotary encoder with push-button shaft, 8 LEDs, 4 slide switches, 4 push buttons (Digilent Co., 2016). This configuration represents a versatile development platform and allows the utilization of both the Xilinx ISE software toolkit and the MicroBlaze processor-based technology within frame of the Xilinx Embedded Development Kit (EDK) software environment.

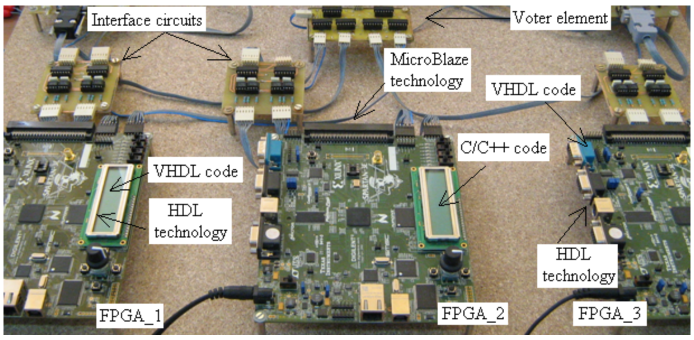


Fig. 4. Laboratory setup: the fault-tolerant digital system.

The FPGA-based development boards have been interfaced then with a digital voter element. The one-bit digital voter circuit configuration is shown at the top of Fig. 5 with its input bits labeled O_{11}, O_{21}, O_{31}, and the output bit Y_1 (see Fig. 1). This combinatory logic circuit implements a majority decision strategy (Johnson, 1989). Because during the experiments a 4-bit output bus was considered for each board, finally a *4x* 1-bit voter has been implemented as is expressed in Fig. 5. Accurate test operations prove that the designed hardware system operates properly and expresses high reliability behaviors in case of an unpredictable board-fault occurs

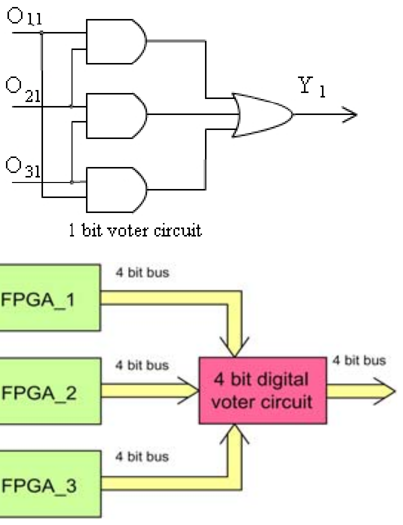
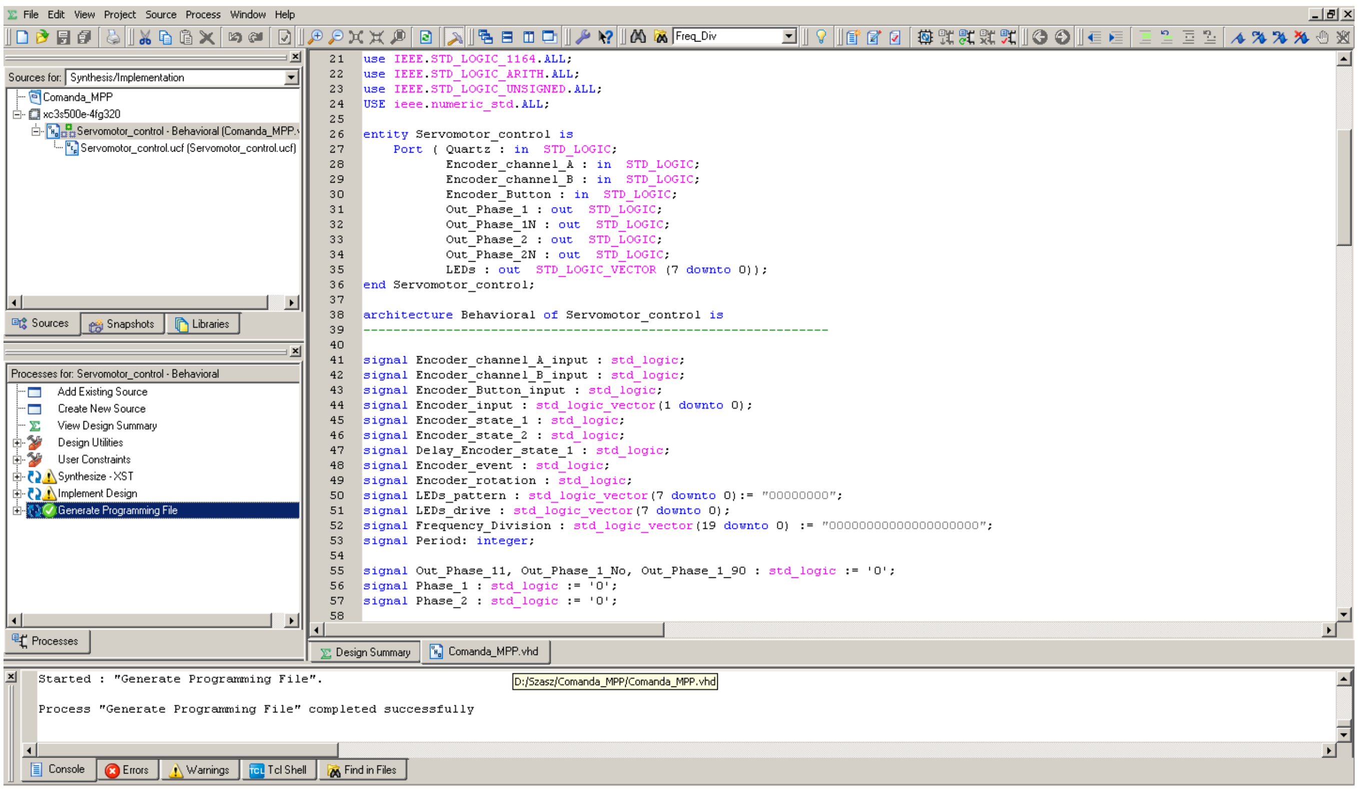


Fig. 5. The FPGA-based development boards interfacing to the 4-bit digital voter.

The hardware experimentation effort was followed by fault-tolerant software design and implementation using *N*-version redundancy strategy. In the first step it has been considered that the digital system executes a simple control task, for example: generates on its output bus two $\pi/2$ degree delayed variable frequency pulse trains (well suited for a two-phase bipolar stepper motor control). For this particular situation it has been applied the *N*-version software redundancy concept presented in Fig. 3. In accordance with this, the Xilinx ISE Toolkit software environment it is used to develop the algorithms *Algorithm_1* and *Algorithm_2* in VHDL code. The first algorithm will be uploaded on the development board labeled *FPGA_1*, the second one on *FPGA_3*. Obviously, the two algorithms implement the same control task but in two different programming ways. A general view of the implemented project by using the Xilinx ISE Toolkit is shown in Fig. 6. There in the left-upper corner of the main window are plotted the embedded project files, respectively down of these are listed the mandatory steps required for the full project implementation. The green color lighted of the *Generate Programming Files* option indicate that the project implementation was successful and a *.bit* extension file is available to be uploaded and executed on the FPGA-based development board.

In the right side of the Xilinx ISE Toolkit main window is shown a piece of the implemented VHDL code for *Algorithm_1*. There the *Port* entity lists the input/output pins of the designed hardware circuit, respectively are declared all the necessary signals used to interface this circuit to the existing physical output buses of the board.


File Edit View Project Source Process Window Help
Freq_Div
Sources for: Synthesis/Implementation
Comanda_MPP
xc3s500e-4fg320
Servomotor_control - Behavioral (Comanda_MPP.vhd)
Servomotor_control.ucf (Servomotor_control.ucf)
Sources
Snapshots
Libraries
Processes for: Servomotor_control - Behavioral
Add Existing Source
Create New Source
View Design Summary
Design Utilities
User Constraints
Synthesize - XST
Implement Design
Generate Programming File
Processes
Design Summary
Comanda_MPP.vhd
D:/Szasz/Comanda_MPP/Comanda_MPP.vhd
Console
Errors
Warnings
Tcl Shell
Find in Files

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
USE ieee.numeric_std.ALL;

entity Servomotor_control is
    Port ( Quartz : in  STD_LOGIC;
           Encoder_channel_A : in  STD_LOGIC;
           Encoder_channel_B : in  STD_LOGIC;
           Encoder_Button : in  STD_LOGIC;
           Out_Phase_1 : out  STD_LOGIC;
           Out_Phase_1N : out  STD_LOGIC;
           Out_Phase_2 : out  STD_LOGIC;
           Out_Phase_2N : out  STD_LOGIC;
           LEDs : out  STD_LOGIC_VECTOR (7 downto 0));
end Servomotor_control;

architecture Behavioral of Servomotor_control is
-------------------------------------------------------------


signal Encoder_channel_A_input : std_logic;
signal Encoder_channel_B_input : std_logic;
signal Encoder_Button_input : std_logic;
signal Encoder_input : std_logic_vector(1 downto 0);
signal Encoder_state_1 : std_logic;
signal Encoder_state_2 : std_logic;
signal Delay_Encoder_state_1 : std_logic;
signal Encoder_event : std_logic;
signal Encoder_rotation : std_logic;
signal LEDs_pattern : std_logic_vector(7 downto 0):= "00000000";
signal LEDs_drive : std_logic_vector(7 downto 0);
signal Frequency_Division : std_logic_vector(19 downto 0) := "00000000000000000000";
signal Period: integer;

signal Out_Phase_11, Out_Phase_1_No, Out_Phase_1_90 : std_logic := '0';
signal Phase_1 : std_logic := '0';
signal Phase_2 : std_logic := '0';
```

```
Started : "Generate Programming File".

Process "Generate Programming File" completed successfully
```

Fig. 6. General view of the Xilinx ISE Toolkit main window: the control task VHDL code implementation (Algorithm_1).

This first algorithm uses the well known sequential circuit scheme with two *D* flip-flops to generate the π/2 degree delayed variable frequency pulse trains (Fig. 7). This output frequency may be changed arbitrarily by using an adequate integer range variable defined by the programmer inside the VHDL source code.

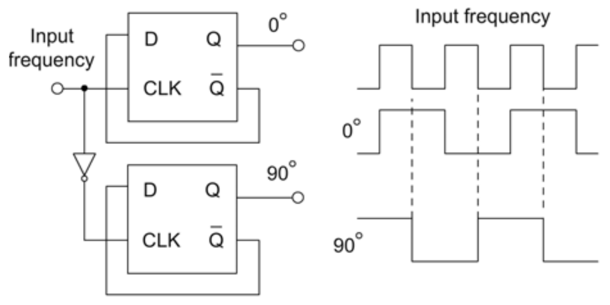


Fig. 7. The π/2 degree delayed pulse trains generation principle.

A small part of the source code written to handle the discussed control task implementation it is also given in the followings.

```
Delay_0:process(Frequency_Division(17))
begin
        if Frequency_Division(17)'event and Frequency_Division(17) =
'0' then
                Phase_1 <= not Phase_1;
        else
                Phase_1 <= Phase_1;
        end if;
end process;
        Out_Phase_1_0 <= Phase_1;
```

```
Delay_90:process(Frequency_Division(17))
begin
        if Frequency_Division(17)'event and Frequency_Division(17)='1'
then
                Phase_2 <= not Phase_2;
        else
                Phase_2 <= Phase_2;
end if;
end process;
        Out_Phase_1_90 <= Phase_2;

        Out_Phase_1 <= Out_Phase_1_0;
        Out_Phase_1N <= not Out_Phase_1_0;
        Out_Phase_2 <= Out_Phase_1_90;
        Out_Phase_2N <= not Out_Phase_1_90;
```

In this VHDL code the *Frequency_Division(17)* counter means that the initial 50MHz Spartan-3E Starter Kit clock is divided by 2^{17}, and the result represents the output frequency of the two π/2 degree delayed pulse trains.

In order to increase the software redundancy of the system, the *Algorithm_2* implements the same control task in a different manner. There the output signals frequency is changed not by using an integer range variable but by tuning the rotating encoder with push-button shaft provided by the manufacturer among the hardware facilities of the Spartan-3E Starter Kit development board. Part of the corresponding algorithm in VHDL code which implements this strategy is listed in the followings. As it has been mentioned before, the *Algorithm_2* will be uploaded on the *FPGA_3* labeled development board:

```
LEDs_display: process(Quartz)
begin
if Quartz'event and Quartz='1' then
```

```
        if Encoder_event='1' then
        if Encoder_rotation='0' then
                if LEDs_state = "00000000" then LEDs_state <=
LEDs_state;
                else LEDs_state <= LEDs_state -1;
                end if;
else
        if LEDs_state = "11111111" then LEDs_state <= LEDs_state;
        else LEDs_state <= LEDs_state +1;
        end if;
        end if;
end if;

if Encoder_Button_input='0' then LEDs_drive <= LEDs_state;
else LEDs_drive <= "00000000";
end if;
        LEDs <= LEDs_drive;
end if;
end process LEDs_display;

Frequency_Division_Process: process(Quartz)
begin
        if Quartz'event and Quartz = '1' then
                Frequency_Division <= Frequency_Division + 1;
        end if;
end process Frequency_Division_Process;
Period <= Conv_integer(LEDs_drive);
Out_Phase_1 <= Frequency_Division(Period);
```

It is well known that by rotating the encoder shaft two-channel pulse trains are generated. The *LEDs_state* signal counters the generated pulses and displays their binary code value. This value it is used then to change the two π/2 degree delayed output signals frequency.

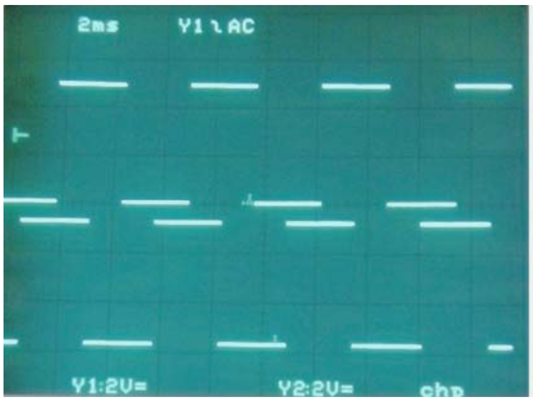
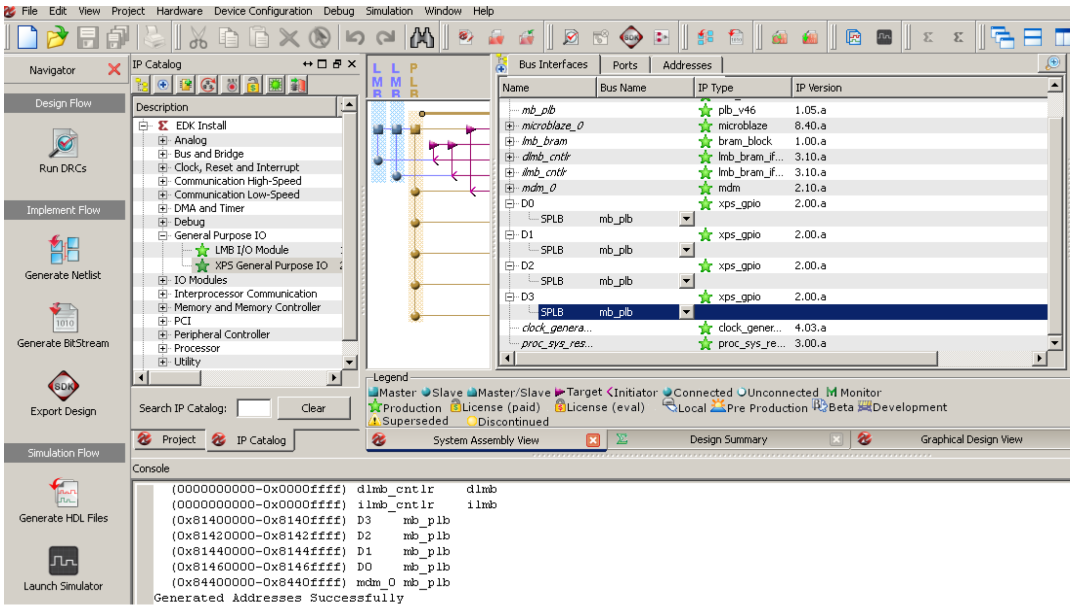


Fig. 8. The π/2 degree delayed waveforms generated by *FPGA_1* and *FPGA_3* boards (Sinca and Szász, 2017, May).

The experiments prove that both *FPGA_1* uploaded with *Algorithm_1* and *FPGA_3* uploaded with *Algorithm_2* generates the same delayed two pulse trains plotted in Fig. 8. This means that software fault-tolerance has been reached by implementing two different program versions of the same control task. The fault-tolerance level of the entire system may be more substantially increased by adding another different implementation version, now designed in frame of a different software technology. Therefore, *FPGA_2* will be uploaded with the same *Algorithm_1*, but implemented now in the Xilinx Platform Studio EDK technology using C/C++ source code. The main menu of this project developed in frame of this software platform is captured in Fig. 9.

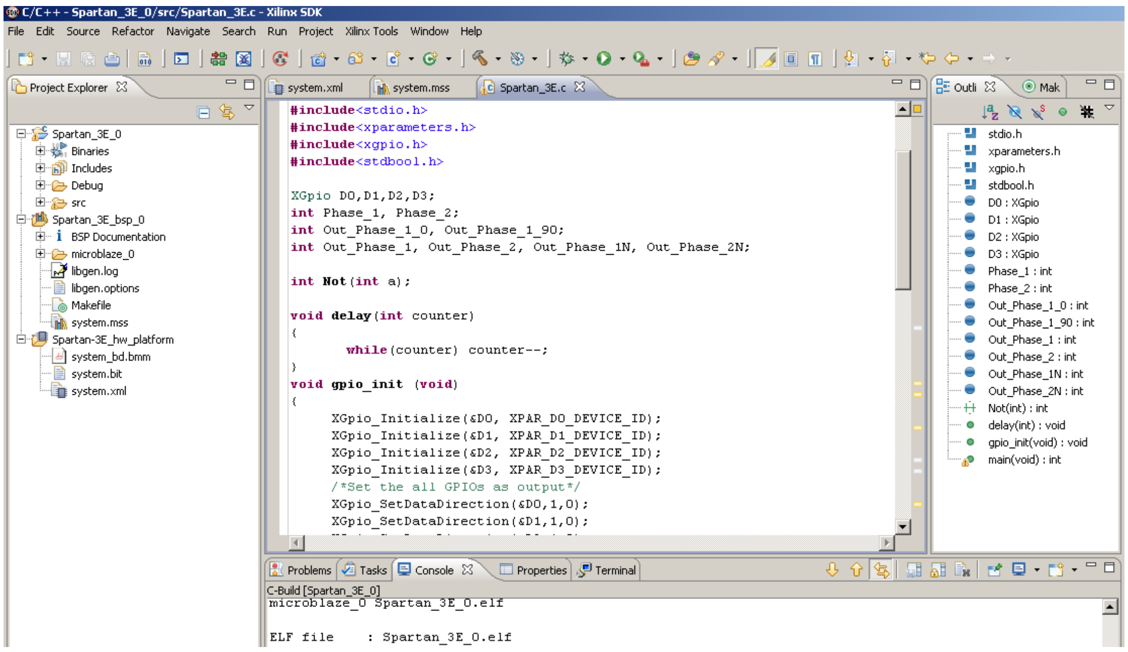


Name	Bus Name	IP Type	IP Version
mb_plb		plb_v46	1.05.a
microblaze_0		microblaze	8.40.a
lmb_bram		bram_block	1.00.a
dlmb_cntlr		lmb_bram_if...	3.10.a
ilmb_cntlr		lmb_bram_if...	3.10.a
mdm_0		mdm	2.10.a
D0		xps_gpio	2.00.a
SPLB	mb_plb		
D1		xps_gpio	2.00.a
SPLB	mb_plb		
D2		xps_gpio	2.00.a
SPLB	mb_plb		
D3		xps_gpio	2.00.a
SPLB	mb_plb		
clock_genera...		clock_gener...	4.03.a
proc_sys_res...		proc_sys_re...	3.00.a

```
  (0000000000-0x0000ffff) dlmb_cntlr    dlmb
  (0000000000-0x0000ffff) ilmb_cntlr    ilmb
  (0x81400000-0x8140ffff) D3    mb_plb
  (0x81420000-0x8142ffff) D2    mb_plb
  (0x81440000-0x8144ffff) D1    mb_plb
  (0x81460000-0x8146ffff) D0    mb_plb
  (0x84400000-0x8440ffff) mdm_0 mb_plb
Generated Addresses Successfully
```

Fig. 9. General view of the Xilinx Platform Studio EDK main window: hardware configuration settings.

There it is possible to observe the main hardware configuration settings of the *FPGA_2* development board. This embeds the MicroBlaze RISC processor with its data and instruction local memory buses (*dlmb* and *ilmb*), their controller circuits, the auxiliary clock circuits, respectively the adequate interfacing circuits to the 4-bit output port (the *D0*, *D1*, *D2*, and *D4* bits) used for control signals generation. All the memory addresses, spaces, or port configurations can be arbitrarily programmed according to a wide range of user needs. Additionally, it is also plotted on the figure the *System Assembly View*, the *Design Summary*, and the *Graphical Design View* menu of the digital system used to unburden the designer development efforts. The hardware configuration operations and settings mandatory might be followed by the C/C++ code software development operations. For this reason, after a successful hardware configuration design the EDK toolkit automatically opens the Xilinx Platform Studio SDK software module.



```
#include<stdio.h>
#include<xparameters.h>
#include<xgpio.h>
#include<stdbool.h>

XGpio D0,D1,D2,D3;
int Phase_1, Phase_2;
int Out_Phase_1_0, Out_Phase_1_90;
int Out_Phase_1, Out_Phase_2, Out_Phase_1N, Out_Phase_2N;

int Not(int a);

void delay(int counter)
{
       while(counter) counter--;
}
void gpio_init (void)
{
     XGpio_Initialize(&D0, XPAR_D0_DEVICE_ID);
     XGpio_Initialize(&D1, XPAR_D1_DEVICE_ID);
     XGpio_Initialize(&D2, XPAR_D2_DEVICE_ID);
     XGpio_Initialize(&D3, XPAR_D3_DEVICE_ID);
     /*Set the all GPIOs as output*/
     XGpio_SetDataDirection(&D0,1,0);
     XGpio_SetDataDirection(&D1,1,0);
```

```
microblaze_0 Spartan_3E_0.elf

ELF file    : Spartan_3E_0.elf
```

Fig. 10. General view of the Xilinx of the Xilinx Platform Studio SDK main window: the C/C++ code software application development and implementation.

Obviously, this new software component launching represents the intrinsic next step of the development (Fig. 10). On main window of this module are plotted all the required software development steps in order to generate the executable file (with extension *.bit*) which will be uploaded at the end on the FPGA-based development board's memory. There are ranked all the folders and adequate files step-by-step generated under the software implementation process. In the middle is listed the main *C* code of the control task which self-evidently is the same two channel π/2 degree delayed variable frequency pulse trains generator. With the main purpose to evidence the major differences between the design and development technologies used to implement the same control task, a small piece of the edited *.C* source code has been cut out and listed below. This is the same part corresponding to the VHDL code presented in the first software version development and implementation (delaying the two pulse trains with *D* latches and arbitrarily setting the output pulses frequency).

```
Phase_1 = 0;
Phase_2 = 0;
clock_pulses = clock();
if (clock_pulses % 50000 ==  0)
          {Phase_1 == Not(Phase_1);}
else Phase_1 = Phase_1;
Out_Phase_1_0 = Phase_1;
if (clock_pulses % 50000 ==  0)
          {
          Phase_2 == Not(Phase_2);
          }
else Phase_2 = Phase_2;
Out_Phase_1_90 = Phase_2;
Out_Phase_1 = Out_Phase_1_0;
Out_Phase_1N = Not(Out_Phase_1_0);
Out_Phase_2 = Out_Phase_1_90;
Out_Phase_2N = Not(Out_Phase_1_90);
XGpio_DiscreteWrite(&D0,1,Out_Phase_1);
XGpio_DiscreteWrite(&D1,1,Out_Phase_1N);
XGpio_DiscreteWrite(&D2,1,Out_Phase_2);
XGpio_DiscreteWrite(&D3,1,Out_Phase_2N);
```

What is important to notice here is that with a careful setting this source code generates on the *FPGA_2* labeled board outputs exactly the same waveforms plotted in Fig. 8, corresponding to the cases when *FPGA_1* was uploaded with the *Algorithm_1* and *FPGA_3* with *Algorithm_2*. Therefore, this last observation also means that the followed *N*-version software redundancy implementation strategy was very successfully. In this particular situation three different software implementation versions have been reached for the same control task. However, the simple fact that two very different technologies have been used in this development additionally increases the software fault-tolerance of the system and outlines the originality of the design. Not least, the use of the hardware reconfigurable technology during the entire development process emphasizes the versatility and high performance of this approach.

It is important to mention here that such a hardware platform is more reliable than any of the three implemented separately on the three identical hardware modules. This may be proved in a relatively simple manner, because in case of the three separately operating boards (meaning serial interconnection) the equation (4) can be written as follow:

$$R_S = R_1 \cdot R_2 \cdot R_3 \,. \tag{5}$$

Assuming that on each board (or module) runs the same algorithm implemented on the same software technology, results that $R_1=R_2=R_3=R_M$ and the relationship (5) becomes:

$$R_S = R_M^3 \,. \tag{6}$$

By subtracting the equation (6) from (4) results that:

$$R_{TMR} - R_S = 3 \cdot R_M^2 \cdot (1 - R_M) > 0 \,. \tag{7}$$

Therefore, $R_{TMR}>R_S$ and the considered hardware platform represents the most reliable solution. Of course, by using different software implementation technologies may be considered $R_1\neq R_2\neq R_3$ but $R_1<R_2<R_3$ and the general reliability coefficient of the TMR system can be increased.

6. CONCLUSIONS

This article emphasizes the advantages and benefits of hardware reconfigurable technology application in combined software-hardware redundancy implementation. In particular outlines the emergence of a novel design paradigm relying on fine-grained parallel and distributed computing behaviors with rapidly changing hardware functionality and huge re-routing abilities according to various user needs.

The theoretical aspects discussed in the paper are supported then via an intuitive implementation example of how to design and develop combined software-hardware redundancy and to achieve high-level of system fault-tolerance. At the end of this development it can be concluded that the hardware reconfigurable paradigm may represent one of the best ways to introduce a new generation of fault-tolerant systems. This technology combined with the last-generation software solutions discussed in the paper covers full predestination for this.

Another conclusion is that reconfigurable technology is quite easy to “learn and adapt to” a specific fault-tolerant application. It is required only simple software operations, without the need of any change on the hardware architecture. Therefore, represents a well fitted solution for a large scale of different complexity fault-tolerant system developments. Hence, it can be also concluded that at current level microelectronics, the reconfigurable hardware technology is a versatile and highly recommended solution for such demanding applications.

REFERENCES

Belli F. and Jedrzeiowicz F., (1990). Fault-tolerant programs and their reliability, *IEEE Transactions on Reliability*, Vol. 16, No. 3, pp. 184-192.

Belzunce F., Marinez H., Ruiz J. (2011). On optimal allocation of redundant components for series and parallel of two dependent components. *Journal of Statistical Planning and Inference*, Vol. 141, No. 9, pp. 3094-3104.

Berman O. and Kumar U. (1999). Optimization models for recovery black schemes. *European Journal of Operational Research,* Vol 115, No. 2. Pp. 368-379.

Bondavalli A., Giadomenico F., and Xu J. (1993). A cost-effective and flexible scheme for software fault tolerance. *Journal of Computer Systems Science and Engineering*, Vol. 8, No. 4, pp. 234-244.

Chielle E., (2016). Selective Software-Implemented Hardware Fault Tolerance Techniques to Detect Soft Errors in Processors with Reduced Overheads, *Porto Alegre: Programa de PósGraduação em Microeletrônica*.

Coulouris G., Dollimore J. and Kindberg T. (2001) Distributed Systems: Concepts and Design, *4th Edition, Pearson Education* Ltd., New York.

Digilent Co, 2016. http://store.digilentinc.com/fpga-programmable-logic/system-boards/

Dugan J.B. and Lyu m.R. (1995). Dependability modeling for fault-tolerant software and systems. *John Wiley & Sons* Ltd., pp. 109-138.

Du B.. et al. (2015). On-line Test of Control Flow Errors: A new Debug Interface-based Approach, *IEEE Transactions on Computers*, vol. PP, no. 99.

Giadomenico F., Bondavalli A., JXu J. (1995). Hardware and Software fault tolerance: adaptive architectures in distributed computing environments. TR. *B415, IEI-CNR*.

Husi G., Szász Cs., Hashimoto H. (2014). Application of reconfigurable hardware technology in the development and implementation of building automation systems. *Environmental Engineering and Management,* Vol. 13/11, ISSN: 1582-9596, http://omicron.ch.tuiasi/EEMJ.

Johnson B.W. (1989). Design and Analysis of Fault-tolerant Digital Systems, *Addison-Wesley series in electrical and computer engineering*, ISBN: 0-201-07570-9.

Khan F.G., Qureshi K., and Nazir B. (2010). Performance evaluation of fault tolerance techniques in grid computing systems. *Computer and Electrical Engineering,* Vol. 36, No. 6, pp. 1110-1122.

Kumar A., Agrawal M., and Garg S.C. (1986). Reliability analysis of a two-unit redundant system with critical human error. *Micro. and Reliability*, Vol. 26, pp. 867-871.

Leach R.J. (2008). Setting checkpoints in legacy code to improve fault-tolerance, *Journal of Systems and Software,* Vol. 81, No. 6, pp. 920-928.

Levitin G. (2006). Reliability and performance analysis of hardware-software systems with fault-tolerant software components. Reliability Engineering and System Safety, Vol. 91, pp. 570-579.

Lewis E. (2001). A load-capacity interference model for common mode failure in 1-out-of-2G system, IEEE Transactions on Reliability, Vol. 50, pp. 47-53.

Lyu M.R. (1995). Software Fault Tolerance, *John Wiley* .

McAllister D.F. and Scott R.K. (1991). Cost modeling of fault-tolerant software. *Information and Software Technology*, Vol. 33, No. 8, pp. 594-603.

Poledna S. (1994). Replica determinism in distributed real-time systems: A brief survey, *Real-Time Systems* 6(3), pp. 289-316.

Randell B., Xu J. (1995). The evolution of the recovery block concept in Software Fault Tolerance, *Whiley,* pp. 1-2.

Rampratap T.Z., (2016). Modeling for Fault Tolerance in Cloud Computing Environment, Journal of Computer Sciences and Applications, Vol. 4, No. 1, 9-13, DOI: 10.12691/jcsa-4-1-2, http://pubs.sciepub.com/jcsa/4/1/2

Rink N., Castrillon J. (2017). Trading Fault Tolerance for Performance in AN Encoding, CF’17 proceedings of the Computing Frontiers Conference, pp. 183-190, ACM New York, DOI:10.1145/3075564.3075565.

Sari A. and Akkaya M. (2015), Fault Tolerance Mechanisms in Distributed Systems. *Int. Journal of Communications, Network and System Sciences,* Vol. 8, pp. 471-482. http://dx.doi.org/10.4236/ijcns.2015.812042.

Sharma U., (2012). A Novel Approach for Providing Fault Tolerance to FPGA-Based Reconfigurable Systems, *IACSIT Int. Journal of Engineering and Technology*, Vol. 4, No. 6, DOI: 10.7763/IJET.2012.V4.492 pp. 821-825.

Scarpino F. (1997). VHDL and AHDL Digital System Implementation. *Prentice Hall*, ISBN-10: 0138570876.

Shin B., (2016). Fault Tolerant Control and Localization for Autonomous Driving: Systems and Architecture, *Technical Report No. UCB/EECS-2016-83,* http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-83.html

Sulekha R. (2011). Thesis on Software and Hardware Reliability of Fault Tolerant Systems, *PhD Thesis, Shobhit Institute of Technology and Engineering, A Deemed-To-Be University Modipura*n, Meerut- India.

Sinca R., Szász Cs. (2017, May). High-reliability Electronic Systems Development and Implementation for Safety Applications, *Journal of Electrical and Electronics Engineering*, P-ISSN: 1844-6035, Vol. 10, nr. 1, pp. 73-78.

Sinca R., Szász Cs. (2017). Fault-tolerant digital systems development usinng triple modular redundancy, *Int. Rev. Appl. Sci. Eng. 8(2017) 1, 3-7, DOI:10.1556/1848.2017.8.*

Teng X. and Pham H. (2002). A software reliability growth model for N-version programming systems, *IEEE transactions on Reliability,* vol. 51, No. 3, pp. 311-321.

Valdes J. and Zequeira R. (2006). On the optimal allocation of two active redundancies in a two-component series system. *Operations Research Letters*, 34/1, pp. 49-52.

Wattanapingskorn N. and Coit D.W. (2007). Fault-tolerant embedded system design and optimization considering reliability estimation uncertainry. *Reliability Engineering and System Safety*, Vol. 92, No 4, pp. 184-192.

Wu Y., Wang Y., and Fernandez E.B. (1994). A uniform approach to software and hardware fault tolerance. *Journal of Systems and Software*, Vol. 29, pp. 117-127.

Yamachi H., Tsujimura Y., Kambayashi Y., and Yamamoto H. (2006). Multi-objective genetic algorithm for solving N-version program design problem. *Reliability Engineering and System Safety,* 91( )9. pp. 1083-1094.

Yang L., and Meng X. (2011). Reliability analysis of a warm standby reparaible system with priority in use. Applied mathematical Modelling, Vol. 35, No. 9, pp. 4295-4303.

Yang M., G. Hua, Y. Feng, J. Gong, (2017). Software Fault-tolerance Techniques for Spacecraft Control, *Wiley Online Library*, DOI: 10.1002/9781119107392.ch