UNSYS digital

The Journal of Instrumentation, Automation and Systems

Technical Implementation of Dual Mode Fault Tolerance

Haryono, Jazi Eko Istiyanto, Agus Harjoko, and Agfianto Eko Putra

Department of Computer Science and Electronics, Gadjah Mada University, Yogyakarta, Indonesia

Abstract—Field Programmable Gate Array (FPGA) is susceptible from hazardous radiation that leads to be in error state. In order to avoid that condition, we apply a fault tolerance technique. Most of the fault tolerances today are only using one mode, which means the fault tolerance that is applied will run all the time without changing its design. It does not consider the condition whether the hazard radiation will occur more frequently or not. As researches have shown, in the orbit, the hazard radiation happens in the South Atlantic Anomaly (SAA)frequently. Therefore, this project creates a new methodology in implementation of fault tolerance by using dual mode. When radiation is happened frequently, we apply more robust fault tolerance; if not frequent, we apply simple fault tolerance. A robust fault tolerance will use more resources, and simple fault tolerance will use less resources. Configuration in FPGA is done by Dynamic Partial Reconfiguration (DPR), which means the transition from robust to simple fault tolerance or vice versa is done while the system is running. This paper will talk about the technical implementation of dual mode fault tolerance by presenting systematically order and important aspect to implement the design successfully. The paper shows a result that dual mode fault tolerance can be configured in FPGA successfully.

Keywords—FPGA, Fault Tolerance, Dynamic Partial Reconfiguration.

I. Introduction

The effects of radiation cause errors in electronic circuits [1]. One such effect is the Single Event Effects (SEE), which causes changes in the value of the memory / flip-flop (SEU) or a combination of logic [1]. There are several methods to prevent a fault in the system such as Triple Modular Redundancy (TMR), Duplex System, and Error Detection and Correction Code (EDAC) such as Hamming Code, Quasi Cyclic Code, etc. The error mitigation is a very active research issue at this time; many research teams are developing methods for increasing reliability of FPGA based systems [2].

Subsystem satellite of On-Board Computer (OBC) must be robust, because OBC has an important role in satellite. According to [3], OBC have to monitor, control, acquire, analyze, make decision, and execute a command. Because of its important role, the OBC should have a good fault tolerance. In satellite, resources are limited that we need to use the resources efficiently. In our work, a fault tolerance system technique that considers radiation environment was developed. Most of fault tolerances today are only using one mode, which means the fault tolerance that is applied will run all the time without changing its configuration. It does not consider the condition whether the hazard radiation will occur more frequently or not. Research has shown that in the orbit, hazard radiation happens in South Atlantic Anomaly (SAA) frequently [4].

Online Checkers were applied in [5] where the module is duplicated. However in the implementation, checker requires more resources to cover the entire existing functionality and becomes more complex. TMR with combination design was applied in [6]. As quoted by the research in [7], in the orbit, TMR design is not enough to mitigate the entire fault that occurs; the fault can happen to two modules at the same time. Fault Tolerance using nine redundancies that makes more intensive in using the resources of FPGA was developed in [8]. Another approach of fault tolerance is found in [9] by triplication the Logic Unit (ALU), along with using TMR. Research in [10] proposed a design that has advantages in efficiency of resource usage because the FPGA can be reconfigured at runtime in accordance with the needs of the system using DPR implementation. But it has not discussed about the implementation of fault tolerance.

Knowing the state of the art of fault tolerances above, we therefore create a new methodology in implementation of fault tolerance by using dual mode. Dual mode design is expected to be efficient in using resources and maintain the robustness. This design considers two conditions: when the hazard radiation occurs more frequently and less frequently. When it is more frequent, we apply more robust fault tolerance; if it is not frequent, we apply simple fault tolerance. For robust fault tolerance, Five Modular Redundancy (FMR) was applied, and for simple fault tolerance, Triple Modular Redundancy (TMR) was applied. This paper will talk about the technical implementation of dual mode fault tolerance, the tools, codes, and important aspect to implement dual mode fault tolerance successfully.

II. Design of Dual Mode Fault Tolerance and Tools

The basic idea is to dynamically change the fault tolerance design while running so we create two designs of the fault tolerance. First design uses TMR and the second design uses FMR. When the hazard radiation is happening frequently we then switch to use FMR, if not then we switch to use TMR. **Figure 1** is the design of the dual mode fault tolerance. In TMR

Corresponding author: Haryono (e-mail:haryono81@gmail.com).
This paper was submitted on February 12, 2014; revised on June 25, 2014; and accepted on June 25, 2014.

state, module 4 and module 5 will be removed from FPGA configuration if module 4 and module 5 exist in FMR state.

We use Virtex 6 FPGA from Xilinx. Integrated Software Environment (ISE) is used to create a module using *VHSIC Hardware Description Language* (VHDL) code. Xilinx Platform Studio (XPS) is used to create Dynamic Partial Reconfiguration (DPR) platform and add Intellectual Property (IP). We use Microblaze processor to execute reconfiguration while system is running, Xilinx Development Kit is used to program the Microblaze processor using c code, to decide when the reconfiguration is done, and to Plan Ahead for Dynamic Reconfiguration planning.

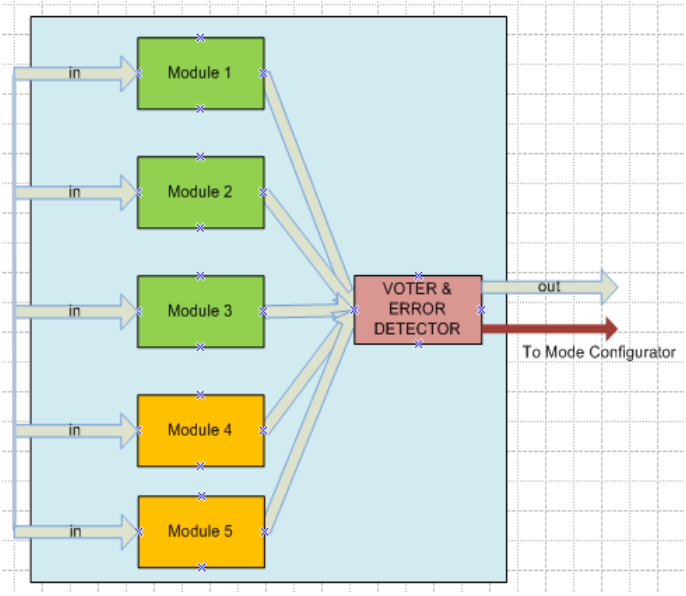


Figure 1The design of the dual mode fault tolerance

Switching between FMR and TMR is done automatically. It is triggered by an error that occurs in the module consecutively. We can know the output of the module so that we can compare to each other which one is error. In this design, we setup with five times, if Error Detector detects an error in some module five times consecutively then the system moves to the FMR mode, at here we consider there are many radiation occurs. If not detected the error at five times every calculation of the module will then move to the TMR mode. The settings of how many errors are detected respectively can be changed according to the needs of the system.

III. Create A Module using VHSIC Hardware Description Language (VHDL) Code

To represent our model of fault tolerance, we create a module that will handle the data more safely. We therefore give an Error Detection and Correction (EDAC) functions in our module. The module is created by the ISE Project Navigator using VHDL code, and at the end, we generated Netlist file that contains information (NGC) file. We only create one module that can be used to reconfigure all modules in an FPGA. We placed the component of the module in a `user_logic.vhdl`, it is done when we are creating IP, then we initiated it five times.

XILINX ISE is used to create the module. Because it is easier to test, we can focus only the VHDL code that we want to test. The clock is an important part that is used to start the processing. The Reset port is used to reset our variable data, to ensure that all the data reverts back to the initial state when we reset. *Input Module* is everything that coming to the module and *Output Module* is the data result which has been processed.

The module should be able to handle the data properly. The data can be an error, may be flip from the original data due to radiation, so we apply the module by Error Detection and Correction (EDAC). EDAC that we are using is Extended Hamming Code (8,4). Hamming codes have a minimum distance of 3, which means that the decoder can detect and correct a single error, but it cannot distinguish a double bit error of some codeword from a single bit error of a different codeword [11]. Therefore, we use Extended Hamming Code that can correct single bit errors and detect two-bit errors. Detail about extended hamming code that is very nicely presented, about how to construct and repair, from theory to technical aspect that inspires this implementation can be found in [11] and [9].

The scenario of the experiment is when the data is received by the module, the module will *encode* the data by Extended hamming code, from encode data the module will *decode* the data. The `output0` will be the *encode data* and the `output1` will be the *decoding/actual data*, the actual data is gotten from the encoded data, because we want to make sure that the Hamming code is properly working. To make the analysis easy, we create *variable* in the hamming code processing. When the code is hit, this variable is assigned immediately using *signal* that will appear after the process is finished. Parity is calculated from actual data that is coming to the module; encoding data is nothing but the actual data plus parity. The output will be the 8 bits, four bits are actual data and the other 4 bits are parity.

The module is able to handle in reading the encoding data/codeword. By calculating the syndrome from the *encoding data***,** we can know which position is an error. If *the syndrome* is zero, it means no error; if *syndrome* in index '0' is zero and the other index (1-3) is not zero, it means double error; and if syndrome in index '0' is one, it means error is happening and can be repaired [11].

IV. Create A Voter Unit and Error Detector

Voter unit is a component that handles to select which data that is correct; the voter unit uses TMR and FMR technique. Switching from FMR to TMR or vice versa can be done while the system is running. Ports that are used are shown in the following VHDL code:

```
Port  (
Clk : in STD_LOGIC;
Reset : in STD_LOGIC;
TmrState  : in std_logic;
ModuleState  : in std_logic_vector(4 downto 0);
ErrorDetectorOutput: out std_logic_vector(4 downto 0);
```

```
Input1 : in  std_logic_vector(159 downto 0);
Input2 : in  std_logic_vector(159 downto 0);
Input3 : in  std_logic_vector(159 downto 0);
Input4 : in  std_logic_vector(159 downto 0);
Input5 : in  std_logic_vector(159 downto 0);
FtResultOut : out  std_logic_vector(159 downto 0)  );
```

`TmrState` is input that indicate whether in TMR mode or FMR mode. `ModuleState` is indicating which module that is active. `ErrorDetectorOutput` will give information which module that is an error, in the future will be used to repair the module when it is in error. `Input1` to `Input5` is input data for each module, as we instantiate the module in a five times each instantiation will be mapped to this input. Last is `FtResultOut`, it is the final data that we expected to be the correct one.

The code inside the voter will check module state. Because the voter can switch while the system is run, we make the data in `Input4` to be '1' or high all so that the voter condition will automatically switch to TMR technique if `TmrState` is '1'.The following VHDL code shows about this technique:

```
FtResult<=
(PR_Input1 and PR_Input2 and PR_Input3 ) or
       (PR_Input1 and PR_Input2 and PR_Input4 ) or
       (PR_Input1 and PR_Input2 and PR_Input5 ) or
       (PR_Input1 and PR_Input3 and PR_Input4 ) or
       (PR_Input1 and PR_Input3 and PR_Input5 ) or

       (PR_Input1 and PR_Input4 and PR_Input5 ) or
       (PR_Input2 and PR_Input3 and PR_Input4 ) or
       (PR_Input2 and PR_Input3 and PR_Input5 ) or
       (PR_Input2 and PR_Input4 and PR_Input5 ) or
       (PR_Input3 and PR_Input4 and PR_Input5 );
```

Error detector will be responsible to detect and determine which module that is erroneous after we know the correct value. In other words, we can know which one that is not correct by simply comparing the result and the input with each input from each module. The following is the VHDL code that employs this technique:

```
ErrorDetector(0) <=  '0' when FtResult = PR_Input1 else '1';
ErrorDetector(1) <=  '0' when FtResult = PR_Input2 else '1';
ErrorDetector(2) <=  '0' when FtResult = PR_Input3 else '1';
ErrorDetector(3) <=  '0' when FtResult = PR_Input4 else '1';
ErrorDetector(4) <=  '0' when FtResult = PR_Input5 else '1';
```

V. Generate The NGCFileof The Module Unit and Voter Unit

After we finish from the VHDL code, we then analyze using simulation. We add the new source by VHDL test bench, then switch to simulation. After everything is run with the proper data, we then generate the NGC file for our module. This NGC file will be used by the XPS. We have to make sure when doing Dynamic Reconfiguration is removing the Add I/O buffers. This can be found in the Process Running Design by right click process properties in the Synthesize, then uncheck the `iobuf` at Xilinx Specific Option. If we do not do this stage, the error will appear during implementation in the XPS project; we cannot serialize the same buffer. To make generating NGC easy and to test the behavior before going to XPS, we can just make one ISE project that contains Module Unit and Voter Unit VHDL; those components will have Top Module. Top Module will do the task as that in `user_logic.vhdl`. We can generate the NGC directly in this project by selecting the component either Module Unit or Voter Unit, but we need to make the component that is selected as Top Module.

VI. Create Dynamic Partial Reconfiguration (DPR) Platform and Create Intellectual Property (IP) in The XPS Project

When switching between modes and fixing the error module, the dual mode fault tolerance must not interrupt the system while running. Therefore we use DPR that is offered by Xilinx. The technical documentation to achieve DPR from Xilinx can be found in [13]. Here, we are stretching about the DPR system that is related to achieving dual mode fault tolerance and some parts that have not mentioned from the document.

By using, XPS we create a project that place our components (voter unit and module unit), and create the structure of which the dual mode fault tolerance will work. Microblaze processor is added in the system assembly view, having a task to manage when the DPR will be done and write the bit data to FPGA RAM that contain the module unit.

Adding IP is not included in the DPR document, but is relatively important to be successful in implementing DPR. The document that talks about adding the IP can be found in the [13]. The IP will handle about how to place our components and communicate with each other. The components that we have made are placed in the `user_logic.vhdl`. `user_logic.vhdl` is a file that will be generated automatically by XPS when we create an IP. `user_logic.vhdl` allows us to add our component and map the ports according to our design. The important aspect and short step that we have to do after adding the IP are as the following.

Step 1: Adding the port in *mpd file***.** If we want to connect our IP to external port, we need to add the name of the port in the file.

Step 2: *Rescanning the project.* This makes our port that is added in mpd file is shown in the Assembly View under Ports Tab at the IP that is added.

Step 3: Making the ports that are shown *to be external*.

Step 4: Adding the port in *the ucf file* according to the name that is shown in the most left column that has been given.

Step 5: Adding a port in the `"Name of IP".vhdl` and then map the port in `user_logic.vhdl` instantiation.

Step 6: In the `user_logic.vhdl`,adding the port in entity, then finally we can use it.

Placing the component and map it in the `user_logic.vhdl`, we add the components that we have created (module unit and voter unit). Following is the VHDL that place the component in the `user_logic.vhdl`**:**

```
COMPONENT VoterUnit
PORT(
   Clk : in STD_LOGIC;
   Reset : in STD_LOGIC;
   FmrOrTmrState  : in std_logic;
   ModuleState: in std_logic_vector(4 downto 0);
   ErrorDetectorOutput: out std_logic_vector(4 downto
0);
   Input1 : in  std_logic_vector(159 downto 0);
   Input2 : in  std_logic_vector(159 downto 0);
   Input3 : in  std_logic_vector(159 downto 0);
   Input4 : in  std_logic_vector(159 downto 0);
   Input5 : in  std_logic_vector(159 downto 0);
   FtResultOut : out  std_logic_vector(159 downto 0)
);
END COMPONENT;

COMPONENT ModuleUnit
PORT(
   Clk : in STD_LOGIC;
   Reset : in STD_LOGIC;
   ModulInput: in std_logic_vector(3 downto 0);
   ModulOutput0: out std_logic_vector(7 downto 0);
   ModulOutput1: out std_logic_vector(3 downto 0)
);
END COMPONENT;
```

The DPR document does not mention to copy our NGC files, but the components that are added should be available in the implementation directory of the XPS project. We need to copy manually the NGC files that have been created because to enable DPR, we need to generate the *BitStream* from XPS. Without those NGC files, the Generate *BitStream* will fail. After Generate *BitStream*, `asystem_bd` file will be created; it is used later when creating the system'*sace* file.

For component initiation, we can initiate the component as many as we want.Module unit is initiated five times to enable the FMR. In the initiation code, we map the port according to the signal that related. Following is some initiations code that we have created:

```
Inst_m1: ModuleUnit PORT MAP(
   Clk            => Bus2IP_Clk,
   Reset          => Bus2IP_Resetn,
   ModulInput     => InputToModule1(3 downto 0),\
   ModulOutput0 => OutputFromModule1(7 downto 0),
   ModulOutput1 => OutputFromModule1(11 downto 8)
   );
Inst_m2: ModuleUnit PORT MAP(
   Clk            => Bus2IP_Clk,
   Reset          => Bus2IP_Resetn,
   ModulInput     => InputToModule2(3 downto 0),
   ModulOutput0   => OutputFromModule2(7 downto 0),
   ModulOutput1 => OutputFromModule2(11 downto 8)
   );
```

Bus2IP_Clk is required to clock the component so that component know when should work and work only is needed. *Inst_m1* and *Inst_m2* will be the portion for Dynamic Partial Reconfiguration. We can remove the connection or establish the connection of the module unit by making the instantiated object as a dynamic partitioned part.

Mapping the output from each module to the input voter unit, the voter should vote which one is the correct data from five modules in FMR mode, and three modules in TMR mode, and find the module that is faulty. Following is the code that maps the output of voter unit:

```
InputToVoter1 <= OutputFromModule1;
InputToVoter2 <= OutputFromModule2;
InputToVoter3 <= OutputFromModule3;
InputToVoter4 <= OutputFromModule4;
InputToVoter5 <= OutputFromModule5;
```

It is important to consider how the communication between the processor and the IP. The IP will have a specific address that will be used by the processor to read and write the data in a 32-bit memory location. When creating the IP, we have to decide how many numbers of software accessible register, one number is having 32-bit memory allocation. We assign the output of the Voter Unit to this address so that we can analyze the result by sending the data to a computer through RS232. Here, we have two numbers of software accessible register so when we want to access the data we need to know the BASEADDR of our IP. Following is the Memory Register Mapping that is reference from Xilinx automation code. `C_BASEADDR` is the IP address from IP creation:

```
"10"     :  C_BASEADDR + 0x0
"01"     :  C_BASEADDR + 0x4
```

VII. Program TheMicroblazeusing C Code in Xilinx Development Kit (SDK) to Operate A DPR

After generating the *Bitstraem* from XPS, we are exporting the design to SDK. This will give prompt to use an SDK application to put our C code and all the *Bitstream* will be exported to our SDK project. We can find the c code template to do DPR in the `UG744_design_files.zip` that is informed in the DPR Document from Xilinx. We add the functionality to check the module for error and then do reconfiguration. Following is the code to check the module that is an error:

```
intErrorModule=Xil_In32(XPAR_DUALMODEFT_0_BASEADDR);
```

Xil_In32 function is used to perform an input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address. Here we read at the address `XPAR_DUALMODEFT_0_BASEADDR` where our IP is residing. The following VHDL code informs that Error Detector data is placed in the first (`"10"`) accessible register:

```
caseslv_reg_read_sel is
   when "10" => slv_ip2bus_data(4 downto 0) <=
```

```
        ErrorDetectorOutput(4 downto 0);
    when "01" => slv_ip2bus_data(11 downto 0) <=
        FtResult(11 downto 0);
    when others => slv_ip2bus_data <= (others => '0');
end case;
```

If we want to access the `FtResult`, we need to add the address of 4 because it is placed in the second (`"01"`) accessible register, which will be `XPAR_DUALMODEFT_0_BASEADDR+4` as Xilinx has described how to access this memory data.

To see the result of our application and to make sure everything is configured properly before we go to the next step in Plan Ahead, we can test the project by loading the bit into the device using of Xilinx Tool because Plan Ahead will consume much times. Based on the experience that we have done, we will be able to get the `system.ace` correctly if we use `.elf` file from *release*, to get `.elf` from *release*, we need to *Build All* the application projects.

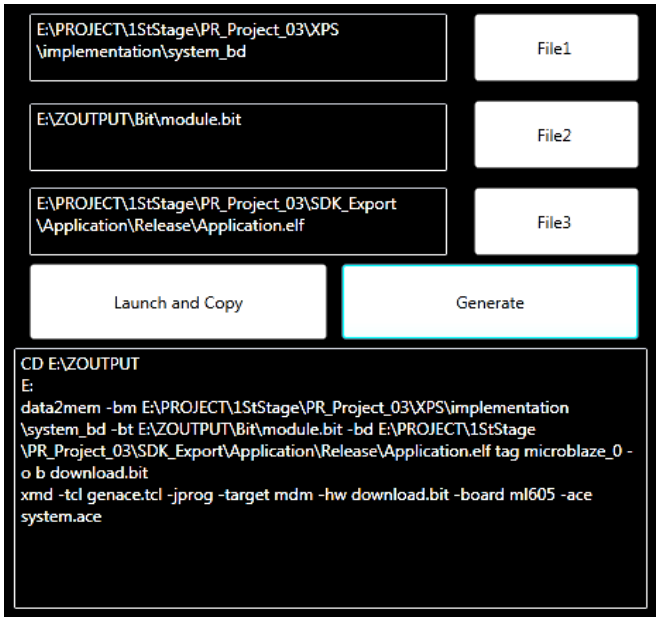


Figure 2 Creating an image file form

VIII. Design Dynamic Reconfiguration using Plan Ahead

The next step is creating the project for the Dynamic Reconfiguration floor plan using Plan Ahead. The important stages we need to care are:

1) In the Plan Ahead, we import all the NGC files that have been created in the XPS to the project. NGC files from our component (Voter Unit and Module Unit) are not included, but they will be included later when we make reconfigurable module. There is opportunity to load the NGC files then.
2) Setting partition for the *netlist* or NGC files. When setting the partition, we have option either to make the partition is filled by *blank* module or by available *netlist*file. We create two files: blank and available netlist. Those will be our bit file to be written up in the RAM FPGA during DPR processing.
3) Placing the reconfiguration module in the device in FPGA. There will be available FPGA area form that can be drag and drop when we select the Set *PblockSize*.
4) Creating the strategies. This stage needs to be done carefully when adding data in*More Options*field in DPR Document using`../../../edk/implementation/system.bmm`. Based on our experience, it cannot be done so we need to change to exact directory, for example:*-bm E:\PROJECT\\XPS\implementation\system.bmm.*
5) Creating the *Design Runs* to make the bit generation plan.

IX. Creating An Image File

Image files contain data reconfiguration for FPGA. Image files will be placed in the Compact flash, they are *system.ace* and partial reconfiguration bit files. To make easy in the process in creating the image files, because we are going to generate the image files for many times: for testing, re-evaluating, and fixing purposes, we create the application to do this stage using C#. **Figure 2** is the application form that will generate the *system.ace* image easily. Three important files that we have to have are *system_bd*, *module* (static file), and the *elf* file.

X. Testing The Dual Mode Fault Tolerance

This chapter will discuss the testing, some comparison with other fault tolerances, and discussion about fault tolerance performance under the proposed dual mode approach. We classify them into four topics:

1. *Measurement with respect to efficiency*: By knowing the resource usage by fault tolerance, it will be known how much efficient of the design. The measurement calculation is done by calculating the power that is used by the FPGA core to the amount of resource used by the FPGA.
2. *Speed measurement*: To measure the speed of mitigation process for error module.
3. *Testing dynamic configuration from TMR to FMR or vice versa*: The system should be able to shift from TMR to the FMR mode or vice versa without disturbing the state of the system. This is to determine whether a system fault tolerance that is made can switch automatically.
4. *Testing the robustness by giving Fault Injection*: This stage is the most decisive test. By providing fault injection to the system, we will know the robustness of the system.

Measurement with respect to efficiency: The greater in using the resources on the FPGA, the greater Power is needed. Calculations were performed using the Xilinx Power Estimator (XPE) for Virtex 6. Figure 3 is a resource calculation on a single module using XPE if a module consists of 1573 LUTs Logic, 103 Distributed RAM, and 1456 flip-flop.

From the results of power calculation, we require 0.010 W for each module. FMR will be activated 20% of the total time. 20% is an estimation of the SAA location obtained on the location of the earth. Equations (1) to (7) are the calculation of the efficiency in one orbit (100 minutes/orbit) when satellite passing the SAA location:

$$\text{TMR Time} = 100 \times \frac{80}{100} = 80 \text{ minutes} = 4800 \text{ seconds} \quad (1)$$

$$\text{Power FMR} = 0.01 \times 5 = 0.05 \text{ W} \quad (2)$$

$$\text{FMR Energy} = \text{FMR Power} \times 4800 \text{ seconds} \quad (3)$$

$$= 0.05 \times 4800 = 240 \text{ W·s} \quad (4)$$

$$\text{TMR Power} = 0.01 \times 3 = 0.03 \text{ W} \quad (5)$$

$$\text{TMR Energy} = \text{TMR Power} \times 4800 \text{ seconds} \quad (6)$$

$$= 0.03 \times 4800 = 144 \text{ W·s} \quad (7)$$

TABLE I Comparison between some fault tolerances in using resources

Technique	Power	Efficiency
TMR without mitigation	0.03 w	Not there
TMR with mitigation	0.03 w	Not there
NMR without mitigation	0.09 w	Not there
Dual mode (FMR and TMR) with mitigation (Proposed design)	TMR: 0.03 w FMR: 0.05 w	96 Ws per orbit

Name	Clock (MHz)	LUTs as			FFs	Toggle Rate	Average Fanout	Signal Rate (Mtr/s)	Power (W)
		Logic	Shift Registers	Distributed RAMs					
module1	100.0	1456	0	103	1573	12.5%	3.00	**12.5**	**0.010**
module2	100.0	1456	0	103	1573	12.5%	3.00	**12.5**	**0.010**
module3	100.0	1456	0	103	1573	12.5%	3.00	**12.5**	**0.010**
module4	100.0	1456	0	103	1573	12.5%	3.00	**12.5**	**0.010**
module5	100.0	1456	0	103	1573	12.5%	3.00	**12.5**	**0.010**

Figure 3 Power calculation in each module using XPE

TABLE I is a comparison between some fault tolerances in using resources. It shows that the proposed design has Efficiency: 240 W·s - 144 = 96 W·s per orbit.

TABLE II Speed of mitigation process to error module

Module	Size		Speed (in millisecond)
	In Kilobyte	In Bit	
1	128	1024000	224.93
2	120	960000	209.66
3	81	648000	141.59
4	128	1024000	225.00
5	142	1136000	261.57

Speed measurement: Microblaze processor speed is 100MHz. **TABLE I** is a comparison between some fault tolerances in using resources. It shows that the proposed design has Efficiency: 240 W·s - 144 = 96 W·s per orbit.

TABLE II shows the speed in performing recovery to error module. The speed shows us that each 1 KB requires 1.75 ms; this speed is sufficient for our OBC. The speed is includes reading non-volatile file on a Compact Flash memory and writing to the ICAP port. The size of each module varies depending on the amount of resources. Although required resource is the same, the size of file is different due to the variation when drawing using *pblock* tool in Plan Ahead.

*Testing dynamic configuration from TMR to FMR***:** In this testing, we want to know whether the designed system can switch from TMR to FMR automatically. Switch from TMR to FMR is done when error in a module is detected 5 times consecutively.

Figure 4 shows fault is injected to the module 5 times consecutively after which the system recovers the faulty module immediately. Then after five times error was detected, the system added module 4 and module 5; if module 4 and module 5 is activated, it is in FMR state.

*Testing dynamic configuration from TMR to FMR***:** In this testing we want to know whether the system can switch from FMR to TMR automatically. Switch from FMR to TMR is done when free error in more than five consecutive calculations occurs.

Figure 5 shows a situation after 6 consecutive calculations without error. In *Blank* field, module 4 and 5 are removed or made to be blank. If module 4 and 5 is removed, we considered it is in TMR state. After switch to TMR state, we send again the data, and we see the data still can be encoded and decoded.

*Testing the robustness by giving Fault Injection***:** In this testing we would like to know, whether the system work correctly or not when many faults are injected to the system. **Figure 6** shows part of injection which is made.

When fault is injected, we send the data to the system immediately before the system makes the recovery. After that, we allow the system to recover the erroneous module. This is done in more than 1 hour with more than 3600 times reconfiguration using DPR technique. The data is still valid; the system can decode and encode the data which is sent to the system correctly.

XI. Conclusion

Dual Mode fault tolerance for FPGA has been implemented successfully, the transition from FMR mode to TMR or vice versa can be done without interrupting the system which is run, this proves that we can make another fault tolerant system with various designs, not only dual mode. Voter unit can select the correct data, whether in FMR or TMR mode, and the error detector can detect the module that is an error. Each module can calculate the Ext Hamming Code with having same output or result to each other. We presented systematical steps to successfully implement Dual Mode Fault Tolerance, and showed the important aspects to achieve this project.

The efficiency using dual mode is about 96 W·s per orbit when satellite passing SAA location. Fault injection is done for more than 1 hour with more than 3600 times reconfiguration, and the system still worked correctly without any crash or hang, and the system can encode/decode the data correctly.

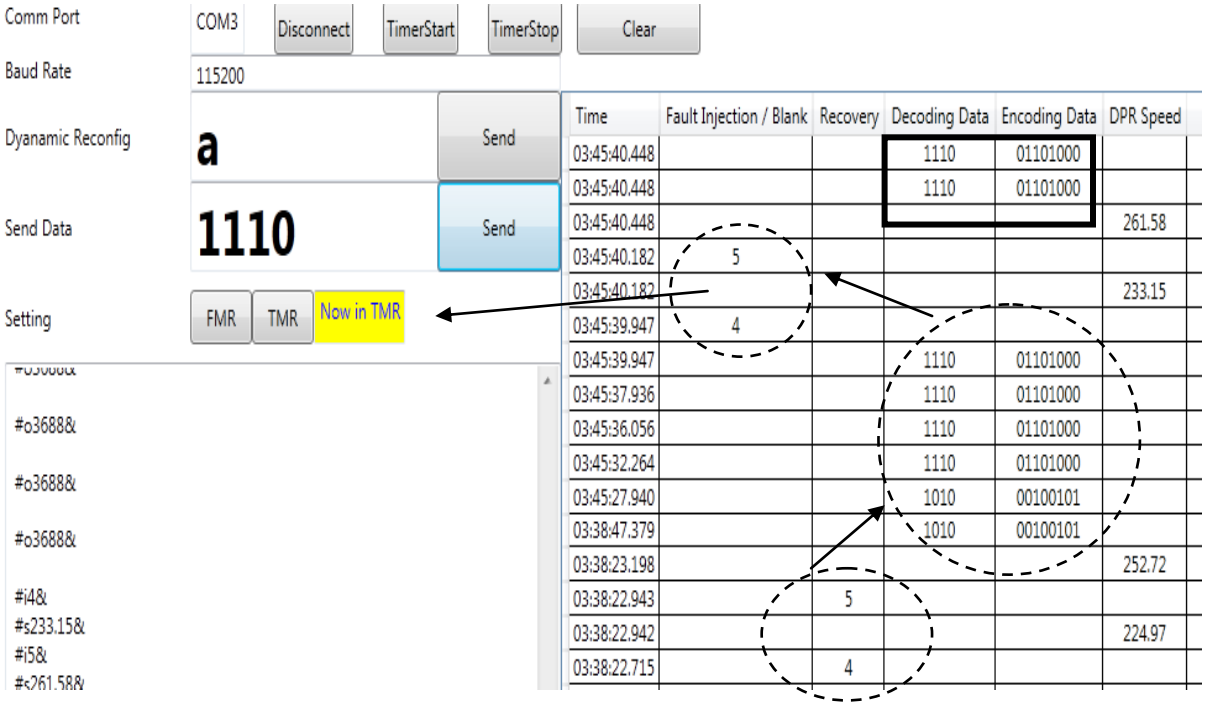


Figure 4 Reconfiguration Testing from TMR to FMR

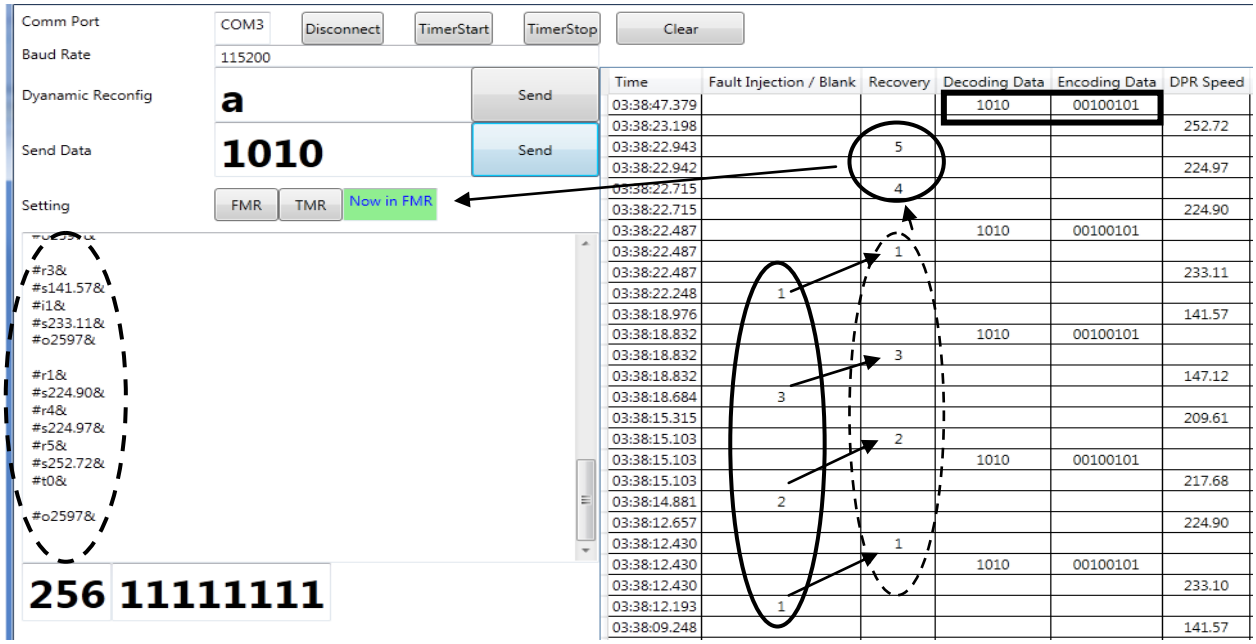
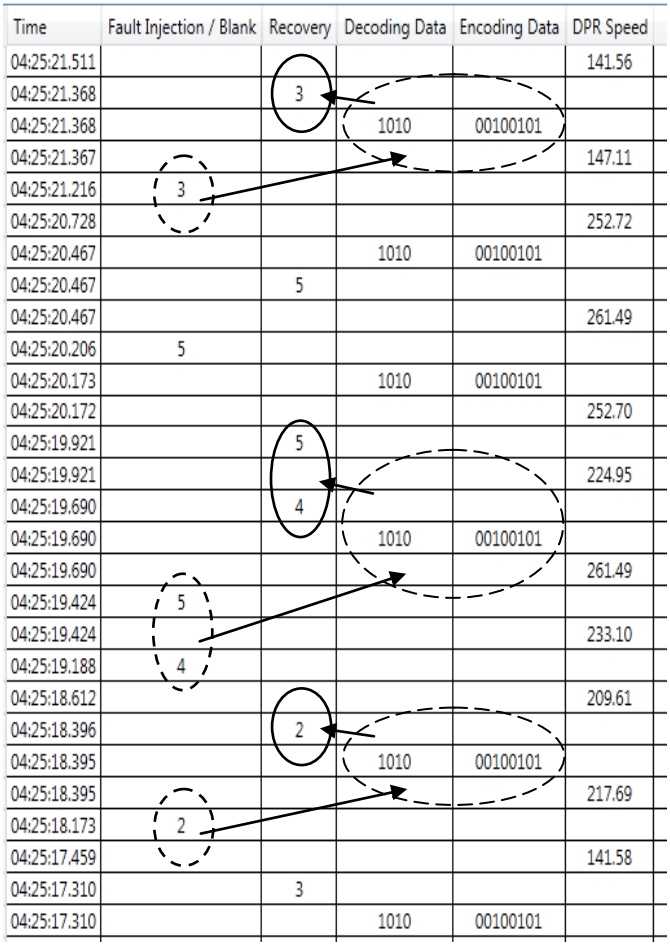


Figure 5 Reconfiguration Testing from FMR to TMR



Time	Fault Injection / Blank	Recovery	Decoding Data	Encoding Data	DPR Speed
04:25:21.511					141.56
04:25:21.368		3			
04:25:21.368			1010	00100101	
04:25:21.367					147.11
04:25:21.216	3				
04:25:20.728					252.72
04:25:20.467			1010	00100101	
04:25:20.467		5			
04:25:20.467					261.49
04:25:20.206	5				
04:25:20.173			1010	00100101	
04:25:20.172					252.70
04:25:19.921		5			
04:25:19.921					224.95
04:25:19.690		4			
04:25:19.690			1010	00100101	
04:25:19.690					261.49
04:25:19.424	5				
04:25:19.424					233.10
04:25:19.188	4				
04:25:18.612					209.61
04:25:18.396		2			
04:25:18.395			1010	00100101	
04:25:18.395					217.69
04:25:18.173	2				
04:25:17.459					141.58
04:25:17.310		3			
04:25:17.310			1010	00100101	

Figure 6 Fault injection testing

ACKNOWLEDGMENT

This project is supported by Satellite Centre - LAPAN and Doctorate Program in Computer Science, Department of Computer Science and Electronics, Faculty of Mathematics and Natural Sciences, Gadjah Mada University. We would like to acknowledge for their support in this project. Thank you very much for Xilinx that publishing about the Technical Partial Reconfiguration document.

REFERENCES

[1] Schrimpf, R. D., Fleetwood, D. M., 2004, Radiation Effects and Soft Errors in Integrated Circuits and Electronic Devices, 34, World Scientific Publishing Wspc, Toh Tuck Link Singapore. CrossRef

[2] Kastil, J., Straka, M., Kotasek, Z., 2012, Methodology for Increasing Reliability of FPGA Design via Partial Reconfiguration, The First Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale (MEDIAN'12), Annecy.

[3] Maral, G., Bousquet, M., 2004, Satellite Communications System, Thomson Press, New Delhi.

[4] Poivey, C., Barth, J.L., LaBel, K.A., Gee, G., Safren, H., 2003, In-flight observations of long-term single-event effect (SEE) performance on Orbview-2 solid state recorders (SSR), Radiation Effects Data Workshop, 2003. IEEE, 21-25 July 2003.

[5] Straka, M., Kotasek, Z., Winter, J., 2008, Digital Systems Architectures Based on On-line Checkers, Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference, Parma.

[6] Straka, M., Kastil, J., Kotasek, Z., 2010, Modern fault tolerant architectures based on partial dynamic reconfiguration in FPGAs, Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium, Vienna.

[7] Bentoutou, Y., 2011, Performance Comparison of Real Time EDAC Systems for Applications On-Board Small Satellites, World Academy Of Science, Engineering and Technology, 131, 66, 709 – 711

[8] Bentoutou, Y., 2011, A Real Time EDAC System for Applications On Board Earth Observation Small Satellites, IEEE Transactions on Aerospace and Electronic Systems, 3, 53, 1022 – 1027.

[9] Shinghal, D. dan Chandra, D., 2011, Design and Analysis of a Fault Tolerant Microprocessor Based on Triple Modular, International Journal of Advances in Engineering & Technology, 1, 1, 21-27.

[10] Savani, V.G., Mecwan, A.I., Gajjar, N.P., 2011, Dynamic Partial Reconfiguration of FPGA for SEU Mitigation and Area Efficiency, International Journal of Advancements in Technology, 2, 2, 285-291.

[11] Wikipedia, Hamming code, [23 January 2014] VIEW ITEM

[12] Ian Elliott, 2002, Advanced Electronic Design Automation, Northumbria University,pp. 15-16[23 January 2014] VIEW ITEM

[13] Xilinx, 2013, Partial Reconfiguration of a Processor Tutorial,[1 January 2014] VIEW ITEM

[14] Xilinx, 2012, Adding Custom IP to an Embedded System Using AXI,[1 January 2014] VIEW ITEM