

Functional Safety for FPGA-Based Hardware Designs

Authors

Khalid Khan

Applications Engineer, Sr Staff,
Synopsys

DeAndre Hoskins

Applications Engineer, Staff,
Synopsys

Introduction

Advances in design and manufacturing technology allow increased factory automation, where tasks are automatically performed by sophisticated equipment such as industrial robots. Manufacturing processes require fail-safe mechanisms to prevent human injury or costly downtime. With increasing sophistication and automation of the manufacturing processes, there is increasing need for error detection and recovery methods.

In addition to industrial applications, the rise of autonomous driving and driver assistance is creating a demand for functional safety design automation. For example, Advanced Driver Assistance Systems (ADAS) include features such as adaptive cruise control, automatic braking, and lane departure warning systems. These features require safe operation of vehicles in the event of random hardware or system failures. These types of failures require safety features that can detect errors and provide warnings to the driver, or automatically correct and maintain the vehicle in a safe operating state.

The design of industrial and automotive applications requires the integration of functional safety measures to ensure the system or equipment operates correctly. This includes safe handling of operator errors, hardware failures or environmental changes. These overall functional safety requirements are defined by the ISO 26262 standard for automotive and the IEC 61508 standard for industrial applications. Due to the re-programmability, long lifespans and high processing bandwidth, FPGAs are being used for industrial and automotive designs. This is requiring FPGA designers to provide for the addition of functional safety features within their designs.

Introduction: IEC 61508 Functional Safety for Industrial

The International Electrotechnical Commission (IEC) 61508 safety standard is the international functional safety standard for industrial applications. It defines “risk” as the likelihood of a hazardous event, and the severity of the possible consequences due to a hazardous event. Reducing possible risks to an “acceptable” level requires the application of Electrical/Electronic/Programmable Electronic (E/E/PES) safety functions. The standard requires an analysis of the specified equipment being designed for potential hazards and risk assessment by using a framework of qualitative analysis of hazards that could affect the design, and then categorizing the likelihood of the occurrence of these hazards. Hazard ratings range from Frequent up to Improbable, with probability values assigned to these categories. Hazard consequences have categories that range from Negligible (minor injuries at worst), to Catastrophic (with multiple loss of life). These two components: qualitative risk combined with hazard consequence comprise a risk-class matrix. Using this risk-class matrix, a corresponding level of desired safety is defined by Safety Integrity Levels (SIL 1-4), which are a target to attain during the development of the application.

Automotive: ISO 26262 Functional Safety

The ISO 26262 Functional Safety specification shares many characteristics with IEC 61508 in the need for systematic design processes that include verification at each stage of development. One difference between the standards is how the safety levels are defined. For ISO 26262, the safety levels are defined as ASILs (Automotive Safety Integrity Levels). There are four ASIL levels, A-D, which are a product of three variables:

- Severity
- Probability of exposure
- Controllability

ASILs aren't specific probability measures, but rather specify the requirements and safety measures needed to avoid residual risk. An ASIL is determined based on the severity of a hazard, the probability of exposure, and the controllability. ISO 26262 describes assessing non-functional causes for failure of safety-related hardware components and part of that safety analysis should identify the following for safety goals:

- Single-point faults or residual faults
- Multiple-point faults
- Evidence of a safety mechanism to maintain a safe state

When designing an FPGA that targets either an industrial application or ADAS system, the methodologies set forth in the IEC 61508 and ISO 26262 specifications can guide the developers in building functional safety. These documents don't specify how a product is developed, but rather a structured method for analyzing potential hazards and measuring the effects of addressing these potential hazards.

There Are Many Types of FPGAs

There are SRAM, anti-fuse and flash-based FPGAs, which vary significantly in their core architecture, and therefore varies what FPGA designers need to look at for error detection and correction. Designers consider multiple factors when choosing an FPGA to implement in a design, such as capacity, speed, per unit cost and built-in error protection offered. The design techniques used for improving the functional safety depend on the type of FPGA utilized and the type of circuit to be protected. Each protection scheme can have significant impact on area and timing for a given design. So, it is important for a designer to understand the functional safety level and how best to achieve it by trading off integration of FPGA architecture functionality versus triplication and other schemes.

The hardware design techniques available can be categorized in terms of type of protection available:

1. Error detection
2. Error detection and correction

Also, the techniques can be categorized based on the nature of circuit being protected:

3. General combinational and sequential logic
4. State machines
5. RAMs

The following sections give details of these techniques, their effectiveness on different types of FPGAs and how Synplify synthesis helps users automate the integration of these techniques into a design.

Redundancy-Based Protection

One of the commonly used methods for protection is to add redundancy. Triple Modular Redundancy (TMR) technique involves triplicating the logic and using single or multiple majority voter(s) to determine the correct output. This scheme provides protection if one of the copies of the triplicate logic is affected by an error, the remaining two copies help determine the correct output. The amount of logic to be triplicated depends on the type of logic that is being protected. E.g. Triplication of registers uses less area than triplication of entire blocks of combinational logic.

Using TMR in SRAM-Based FPGAs

In case of SRAM-based FPGAs, configuration memories are more susceptible to soft errors. In comparison, the FIT (failure in time) rate for register is relatively small. Configuration memories are used for routing matrix and for configuring logic cell functionality. A fault in routing can affect both the combinational and sequential logic. So TMR for SRAM-based FPGAs require triplication of blocks of logic containing the lookup tables and registers and the addition of voters at the module outputs.

Logic blocks can also contain sequential feedback loops. A fault introduced in sequential feedback loops can retain an error over multiple clock cycles even if the enclosing logic block is subjected to TMR. So it's important to add a majority voter at the output of registers which are part of feedback loops. Distributed TMR, a feature available in Synplify allows designers to triplicate blocks of logic and add majority voters to the output of blocks and in the sequential feedback loops.

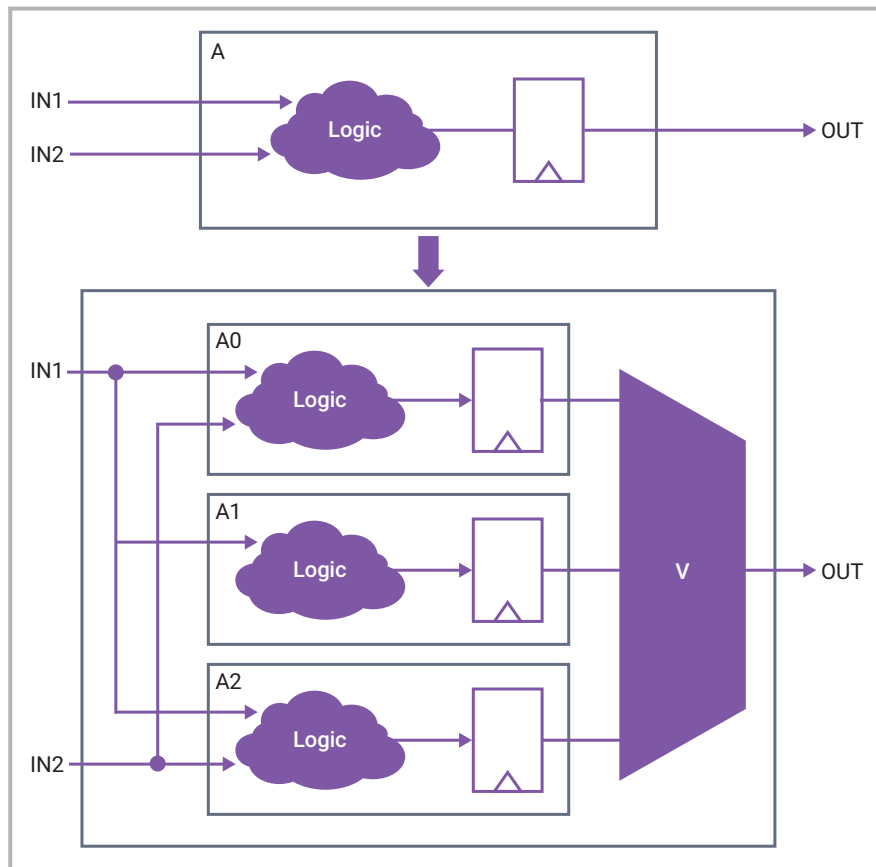


Figure 1: Distributed TMR with no feedback loops

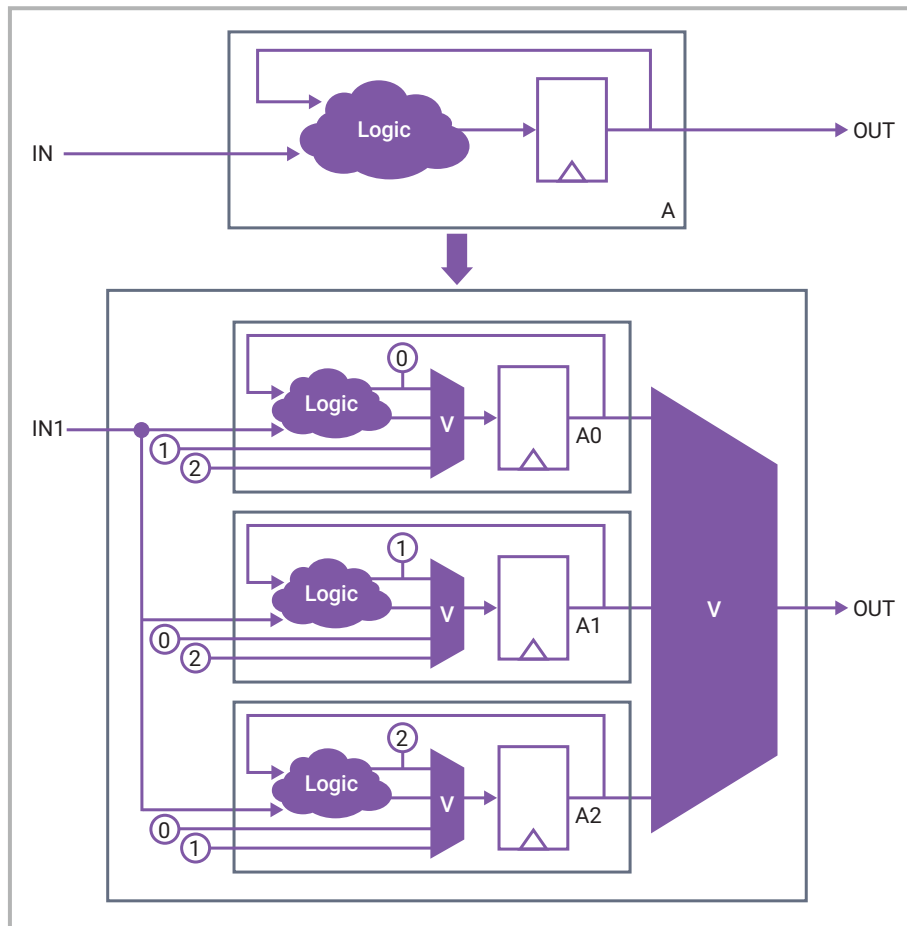


Figure 2: Distributed TMR with feedback loops

Users can have multiple interconnected blocks which require protection and by using Synplify, the designer can have the triplication and voter circuitry added automatically.

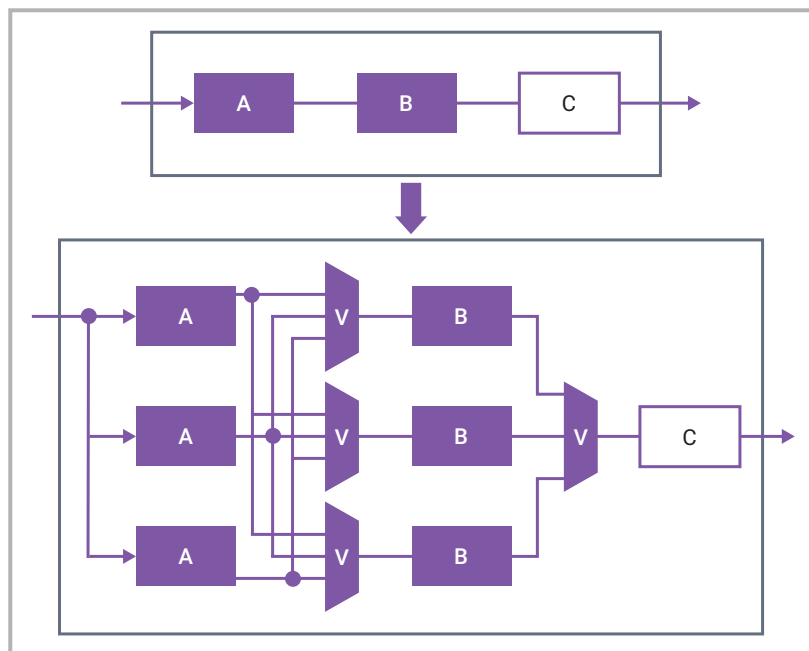


Figure 3: Multiple module TMR with triplicated voters

Some users require extra protection for every path in the design. Synplify provides a method for this level of protection and can add majority voter at every register in the triplicated copies of the block that are subjected to distributed TMR.

On the other hand, some users cannot afford the cost of voters at the sequential feedback loops in the design. In this case, the user can apply distributed TMR with voter logic only at the output of modules, which is called block TMR.

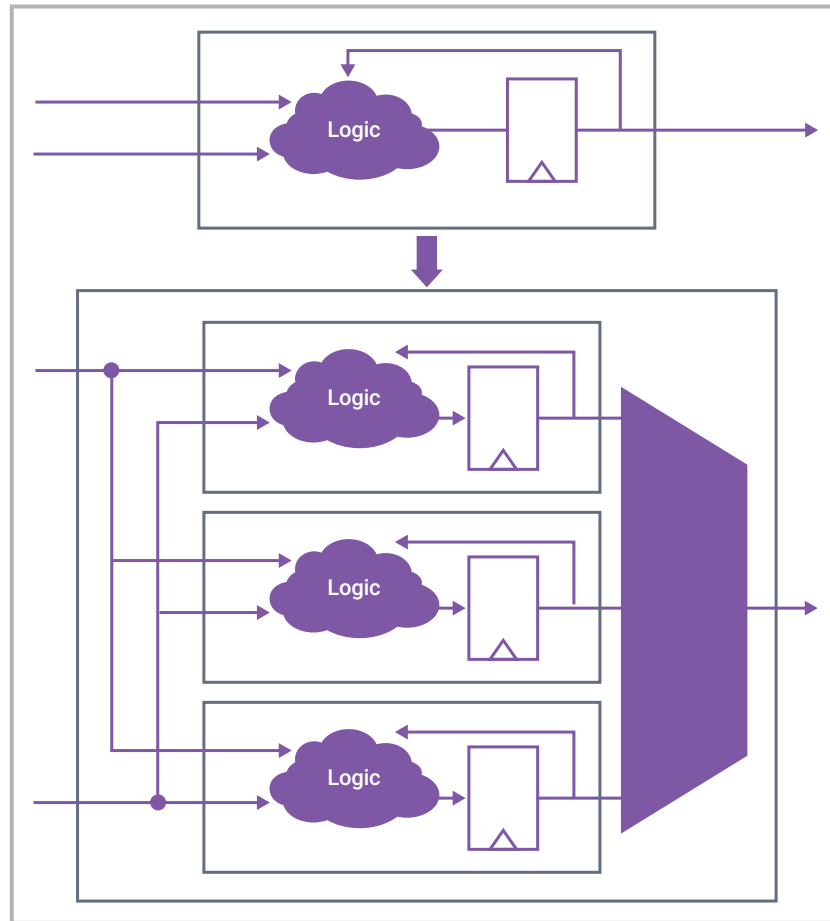


Figure 4: Block TMR—no voter on feedback path loop

With block TMR, if an error affects a register in feedback loop, it can cause accumulation, which can make a design non-functional. The designer can incorporate ways to detect and mitigate such errors like resetting the block into the design. Synplify provides designers a mechanism for detecting such errors and we will discuss this in the error monitoring section later in this white paper.

TMR for Antifuse Or Flash-Based FPGAs

In case of antifuse or flash-based FPGAs, registers are susceptible to errors and routing is not affected. So, it is important for the designer to protect the registers to help improve functional safety. In this case, the designer can integrate local TMR into the design to triplicate the registers and add voter logic.

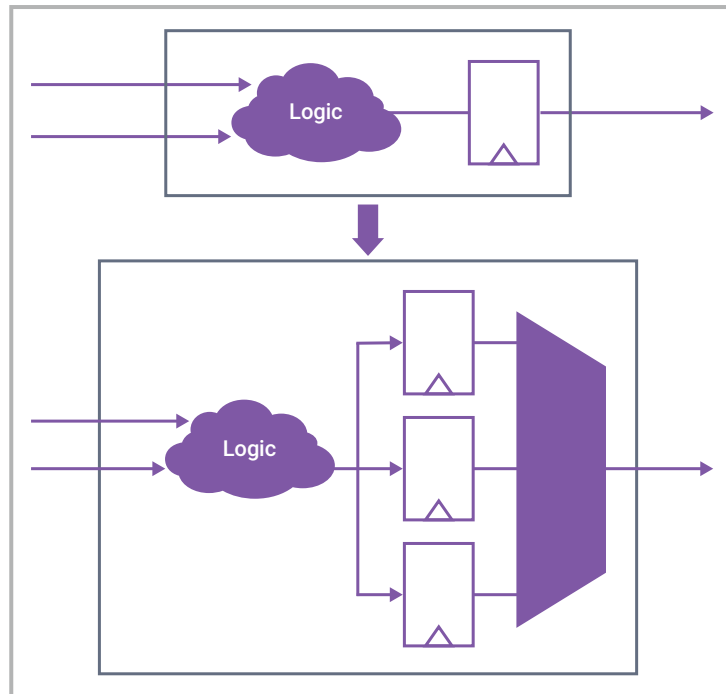


Figure 5: Local Triple Modular Redundancy to add error detection and mitigation for registers

Safe Guarding State Machines

Finite state machines (FSM) are critical components of any design and errors on any register belonging to a state machine can impact the state of the whole design. Protecting FSM registers is critical for reliable design execution, and allowing the state machine to operate in its legal states. A soft error can create an illegal state within an FSM which could cause deadlock and therefore the design to fail. Although the techniques described above such as distributed TMR or local TMR can help FSMs as well, it causes larger area and worse timing. To address these types of issues Synplify provides integrated methods to detect errors on a state machine register and reset to a valid state.

```
always @ (posedge clk)
    if (rst)
        present_state <= s0;
    else
        present_state <= next_state;
always @ (in1, present_state)
    case (present_state)
        s0 : . . . . . ;
        s1 : . . . . . ;
        s2 : . . . . . ;
        s3 : . . . . . ;
        default : next_state <= s3;
    endcase
```

- **Safe encoding:** With safe encoding, if an FSM gets into illegal state because of SEU, a reset signal is applied to the state machine to bring it into a reset state. The reset signal can be asynchronous or synchronous. Synplify supports `syn_encoding = safe` as an attribute on FSM. In the example, the FSM has 4 states (s0, s1, s2 and s3). If the FSM goes into an illegal state because of an error, logic is added to reset the state machine to state s0.
- **Safe state machine:** If an FSM gets into illegal state, it goes into the default state described in the RTL in the following clock cycle. In the state machine shown above, the default state is s3 and the designer needs to implement the necessary logic to mitigate the effect of the errors.
- **Hamming-3 encoding:** This is used to encode the state of an FSM and allow detection and correction of a one-bit error. If one of the state registers changes due to an error, hamming-3 encoding identifies the correct state. An option to detect bit errors in FSM and correct it in the following clock cycle is supported.

The three methods described above provide protection to finite state machines and designers can select one of these methods to add error detection and mitigation to their design automatically based on the functional requirements.

Safe Guarding Block RAMs

Just like configuration bits, Block RAM contents are also susceptible to errors and need to be protected. One approach is to do distributed or block TMR for the module containing Block RAM. Another method is to triplicate each Block RAM and add voter logic. If the FPGA being used supports Error Code Correction (ECC) logic for Block RAMs the designer can use it. Synplify provides supports for all 3 methods described above to protect Block RAM contents. The TMR of Block RAM provides correction for multiple bit errors, but requires three times more Block RAM resources. Since FPGAs have limited number of Block RAMs, this may not be feasible approach for all applications.

ECC provides one-bit correction and is built-in to the memories which requires no additional Block RAMs. This reduces area but due to the lower protection level may not be applicable to certain configurations.

- **Using TMR for Block RAM:** To use TMR for Block RAM, a user can enclose RAM in a module and apply Block or distributed TMR or specify local TMR for the Block RAM.
- **Applying ECC to BRAM:** ECC can be enabled for a Block RAM using `syn_ramstyle` attribute available in Synplify. Depending on the target technology, Synplify will use appropriate Block RAM configuration.

Error Detection and Monitoring Techniques

For certain applications, detection of an error is critical but if the area is too big, a designer may use custom methods to correct errors. Such methods can include applying global reset to the design and scrubbing configuration memories on the FPGA. In these cases, duplication of logic susceptible to an error could be sufficient. A comparator can be built on the output of duplicated logic to detect error and designers can use an error flag to implement their custom error correction method. The area penalty using this method is lower compared to full triplication techniques as there are only two copies of the modules instead of three. Synplify provides methods to duplicate modules and add comparator logic automatically to FPGA designs for state machines, memory, and logic. It also provides ways to access to the comparator output to monitor errors and trigger necessary error mitigation logic.

Summary

Functional safety can be implemented in FPGA designs through features in Synplify. Whether the application is industrial equipment or automotive design, the functional safety Synplify will help designers address functional safety as identified in the IEC 61508 or ISO 26262 specifications.