LogiCORE™ IP Soft Error Mitigation Controller v3.1

User Guide

UG764 October 19, 2011

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/21/10	1.0	Initial Xilinx release.
12/14/10	2.0	Updated core to v1.2 and ISE to v12.4.
03/01/11	3.0	Updated core to v1.3 and ISE to v13.1.
06/22/11	4.0	Updated core to v2.1 and ISE Design Suite to v13.2. Added support for Spartan-6 devices.
10/19/11	5.0	Updated core to v3.1 and ISE Design Suite to v13.3. Added support for Kintex-7 and Virtex-7 (excluding SSI) devices.

Table of Contents

Chapter 4: Example Design Overview

Chapter 5: Generating the Solution

Chapter 6: Building the Solution

Chapter 7: Design Constraints

Chapter 8: Applying the Solution

Chapter 9: Customizing the Solution

Chapter 10: Additional Considerations

Schedule of Figures

Chapter 1: Introduction

Chapter 2: Soft Error Overview

Chapter 3: Core Overview

Chapter 4: Example Design Overview

Chapter 5: Generating the Solution

Chapter 6: Building the Solution

Chapter 7: Design Constraints

Chapter 8: Applying the Solution

Chapter 9: Customizing the Solution

Chapter 10: Additional Considerations

Schedule of Tables

Chapter 1: Introduction

Chapter 2: Soft Error Overview

Chapter 3: Core Overview

Chapter 4: Example Design Overview

Chapter 5: Generating the Solution

Chapter 6: Building the Solution

Chapter 7: Design Constraints

Chapter 8: Applying the Solution

About This Guide

The LogiCORE™ IP Soft Error Mitigation Controller User Guide provides information about the Soft Error Mitigation (SEM) Controller. This guide describes how to design with the core by outlining the key interfaces, detailing its behaviors, and providing an operational overview.

Guide Contents

This manual contains the following chapters:

- Chapter 1, Introduction describes the core and related information, including licensing, recommended design experience, additional resources, technical support, and submitting feedback to Xilinx.
- Chapter 2, Soft Error Overview describes soft errors, surveys the reliability issues associated with them, and proposes techniques for mitigating them.
- Chapter 3, Core Overview overviews the function of the SEM Controller and the interfaces it exposes.
- Chapter 4, Example Design Overview overviews the function of the SEM Controller system-level design example and the interfaces it exposes.
- Chapter 5, Generating the Solution provides instructions for generating the SEM Controller solution.
- Chapter 6, Building the Solution provides details on building the SEM Controller solution.
- Chapter 7, Design Constraints describes the user constraints file provided with the SEM Controller.
- Chapter 8, Applying the Solution provides insight into the SEM Controller solution and how to apply it from three different levels, using a bottom-up approach.
- Chapter 9, Customizing the Solution provides details about customizing the SEM Controller.
- Chapter 10, Additional Considerations provides information critical to successful application of the SEM Controller in a complete design.

Additional Resources

To find additional documentation, see the Xilinx website at:

www.xilinx.com/support/documentation/index.htm.

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

www.xilinx.com/support/mysupport.htm.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *design_name*
Helvetica bold	Commands that you select from a menu	**File → Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *design_name*
	References to other manuals	See the *User Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Dark Shading	Items that are not supported or reserved	This feature is not supported
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**`lowpwr =`**{**`on`**|**`off`**}
Vertical bar |	Separates items in a list of choices	**`lowpwr =`**{**`on`**|**`off`**}
Angle brackets < >	User-defined variable or in code samples	<directory name>

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		.
		.
		.

Horizontal ellipsis . . .	Repetitive material that has been omitted	**`allow block`** *block_name loc1 loc2 ... locn;*

Convention	Meaning or Use	Example
Notations	The prefix '0x' or the suffix 'h' indicate hexadecimal notation	A read of address 0x00112975 returned 45524943h.
	An '_n' means the signal is active low	**usr_teof_n** is active low.

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Blue, underlined text	Hyperlink to a website (URL)	Go to www.xilinx.com for the latest speed files.

XILINX®

Chapter 1

Introduction

This chapter introduces the Soft Error Mitigation (SEM) Controller core and provides related information, including recommended design experience, additional resources, technical support, and submitting feedback to Xilinx. The SEM Controller is designed to support both Verilog and VHDL design environments.

Soft error mitigation is an emerging concern for commercial FPGA devices although it has been a long-time consideration for aerospace and high-reliability/high-availability applications. This guide covers the implementation and use of the SEM Controller.

About the Core

The SEM Controller is a Xilinx CORE Generator™ IP core, included in the latest IP update on the Xilinx IP Center. For detailed information about the core, see:

www.xilinx.com/products/intellectual-property/SEM.htm

The SEM Controller can detect, correct, and classify soft errors in the Configuration Memory of an FPGA device. The solution does not prevent soft errors in Configuration Memory; rather, it provides designers with a method to better manage the system-level effects of these events. Careful management of these events increases system reliability and availability.

Licensing

The SEM Controller core is a free solution. No license key is required to generate the core through CORE Generator.

System Requirements

The SEM Controller core is not used in isolation, and is intended for integration into a larger design. The system requirements are therefore set by the nature of the larger design.

When the SEM Controller is configured to use the optional error classification function or the optional correction by replace function, the controller requires externally stored data created by the Xilinx implementation tools. The application that creates this data requires a large amount of system RAM. Xilinx recommends the use of a 64-bit operating system with 16 Gb or more of system RAM.

Recommended Design Experience

The SEM Controller operates at a relatively low frequency. The implementation of the controller in Xilinx FPGA devices is not challenging. However, evaluation of the suitability

of this solution for use in a system and proper application of the solution requires careful analysis and detailed understanding of the system requirements. These considerations are design specific and outside the scope of this document.

Contact your local Xilinx representative for a closer review of your specific requirements.

Additional Core Resources

For detailed information and updates about the SEM Controller, see the following documents, located on the SEM Controller product page at:

www.xilinx.com/products/ipcenter/SEM.htm

- *LogiCORE IP Soft Error Mitigation Controller Data Sheet*
- *LogiCORE IP Soft Error Mitigation Controller Release Notes*

Technical Support

For technical support, go to www.xilinx.com/support. Questions are routed to a team with expertise using the SEM Controller.

Xilinx will provide technical support for use of this product as described in the *Soft Error Mitigation Controller User Guide*. Xilinx cannot guarantee timing, functionality, or support of this product for designs that do not follow these guidelines.

Feedback

Xilinx welcomes comments and suggestions about the SEM Controller and the accompanying documentation.

SEM Controller

For comments or suggestions about the SEM Controller, please submit a WebCase from www.xilinx.com/support/clearexpress/websupport.htm. Be sure to include the following information:

- Product name
- Core version number
- Explanation of your comments

Documentation

For comments or suggestions about the SEM Controller documentation, please submit a WebCase from www.xilinx.com/support/clearexpress/websupport.htm. Be sure to include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

References

1. UG470, *7 Series FPGAs Configuration User Guide*
2. UG360, *Xilinx Virtex-6 FPGA Configuration User Guide*
3. UG380, *Xilinx Spartan-6 FPGA Configuration User Guide*
4. UG116, *Xilinx Device Reliability Report*
5. WP286, *Continuing Experiments of Atmospheric Neutron Effects on Deep Submicron Integrated Circuits*
6. XAPP962, *Single-Event Upset Mitigation for Xilinx FPGA Block Memories*

XILINX®

Chapter 2

Soft Error Overview

This chapter describes soft errors, surveys the reliability issues associated with them, and proposes techniques for mitigating them. A variety of solutions are possible, and the selection of the best solution is highly application dependent. The SEM Controller is the best solution for highly demanding commercial applications. However, most applications are well served using the integrated silicon soft error mitigation features alone. Xilinx recommends careful consideration of system-level requirements in the decision to use and selection of soft error mitigation solutions.

Overview

Ionizing radiation is capable of inducing undesired effects in most silicon devices. Broadly, an undesired effect resulting from a single event is called a single event effect (SEE). In most cases, these events do not permanently damage the silicon device; SEEs that result in no permanent damage to the device are called soft errors. However, soft errors have the potential to reduce reliability.

Xilinx devices are designed to have an inherently low susceptibility to soft errors. However, Xilinx also recognizes that soft errors are unavoidable within commercial and practical constraints. As a result, Xilinx has integrated soft error detection and correction capability into many device families.

In many applications, soft errors can simply be ignored. In applications where higher reliability is desired, the integrated soft error detection and correction capability is usually sufficient. In demanding applications, the SEM Controller can ensure an even higher level of reliability.

If a soft error occurs, one or more memory bits are corrupted. The memory bits affected may be in the device configuration memory (which determines the behavior of the design), or may be in design memory elements (which determine the state of the design). The following four memory categories represent a majority of the memory in a device:

- **Configuration Memory.** Storage elements used to configure the function of the design loaded into the device. This includes function block behavior and function block connectivity. This memory is physically distributed across the entire device and represents the largest number of bits. Only a fraction of the bits are essential to the proper operation of any specific design loaded into the device.
- **Block Memory**. High capacity storage elements used to store design state. As the name implies, the bits are clustered into a physical block, with a number of blocks distributed across the entire device. Block Memory represents the second largest number of bits.
- **Distributed Memory**. Medium capacity storage elements used to store design state. This type of memory is present in certain configurable logic blocks (CLBs) and is

distributed across the entire device. Distributed Memory represents the third largest number of bits.

- **Flip Flops**. Low capacity storage elements used to store design state. This type of memory is present in all configurable logic blocks (CLBs) and is distributed across the entire device. Flip Flops represent the fourth largest number of bits.

An extremely small number of additional memory bits exist as internal device control registers and state elements. Soft errors occurring in these areas may result in regional or device-wide interference that is referred to as a single-event functional interrupt (SEFI). Due to the small number of these memory bits, the frequency of SEFI events is considered negligible in this discussion, and these infrequent events are not addressed by the SEM Controller.

Soft error mitigation for the design state in Block Memory, Distributed Memory, and Flip Flops can be performed in the design itself, by applying standard techniques such as error detection and correction codes or redundancy. Soft errors in unused storage elements (those physically present in the device, but unused by the design) are simply ignored. Designers concerned about reliability must assess risk areas in the design and incorporate mitigation techniques for the design state as warranted.

Soft error mitigation for the design function in Configuration Memory is performed using error detection and correction codes.

Configuration Memory is organized as an array of frames, much like a wide static RAM. In many device families, each frame is protected by ECC, with the entire array of frames protected by CRC in all device families. The two techniques are complementary; CRC is incredibly robust for error detection, while ECC provides high resolution of error location.

The SEM Controller builds upon the robust capability of the integrated logic by adding optional capability to classify Configuration Memory errors as either “essential” or “non-essential.” This leverages the fact that only a fraction of the Configuration Memory bits are essential to the proper operation of any specific design.

Without error classification, all Configuration Memory errors must be considered “essential.” With error classification, most errors will be assessed “non-essential” which eliminates false alarms and reduces the frequency errors that require a potentially disruptive system-level mitigation response.

Additionally, the SEM Controller extends the built-in correction capability to accelerate error detection and provides the capability to handle multi-bit errors.

If the extended features offered by the SEM Controller like error injection, error classification, correction by enhanced repair, and correction by replace are not required, the integrated soft error detection and correction capability in the silicon should be sufficient for SEU mitigation. See UG470, *7 Series FPGAs Configuration User Guide*, UG360, *Virtex-6 FPGA Configuration User Guide* and UG380, *Spartan-6 FPGA Configuration User Guide* for information on how to use the built-in error detection and correction capability.

Reliability Estimation

As a starting point, a designer’s specification for system reliability should highlight critical sections of the system design and provide a value for the required reliability of each sub-section. Reliability requirements are typically expressed as failures in time (FIT), which is the number of design failures that can be expected in 10^9 hours (approximately 114,155 years).

When more than one instance of a design is deployed, the probability of a soft error affecting any one of them increases proportionately. For example, if the design is shipped

in 1,000 units of product, the nominal FIT across all deployed units is 1,000 times greater. This is an important consideration because the nominal FIT of the total deployment can grow large and may represent a service or maintenance burden.

The nominal FIT is different from the probability of an individual unit being affected. Also, the probability of a specific unit incurring a second soft error is determined by the FIT of the individual design and not the deployment. This is an important consideration when assessing suitable soft error mitigation strategies for an application.

The FIT associated with soft errors must not be confused with that of product life expectancy, which considers the replacement or physical repair of some part of a system.

Xilinx device FIT data is reported in UG116, *Device Reliability Report*. To render an estimate, the device FIT data must be combined with additional design-specific data that indicates how many bits of each memory type are used.

Observations

The data in UG116, *Device Reliability Report*, reveals the overall infrequency of soft errors in devices. Although some memory types clearly contribute more than others, it is important to note that the failure rates involved are so small that most designs need not include any form of soft error mitigation.

The contribution to FIT from Flip Flops is negligible based on the Flip Flop's very low FIT/Mbit and small quantity. However, this does not discount the importance of protecting the design state stored in Flip Flops. If any state stored in Flip Flops is highly important to design operation, the design must contain logic to detect, correct, and recover from soft errors in a manner appropriate to the application.

The contribution to FIT from Distributed Memory and Block Memory can be large in designs where these resources are highly utilized. As previously noted, the FIT contribution can be substantially decreased by using soft error mitigation techniques in the design. For example, Block Memory resources include built-in error detection and correction circuits that can be used in certain Block Memory configurations. For all Block Memory and Distributed Memory configurations, soft error mitigation techniques may be applied using programmable logic resources.

The contribution to FIT from Configuration Memory is large. Without using an error classification technique, all soft errors in Configuration Memory must be considered "essential," and the resulting contribution to FIT will eclipse all other sources combined. Use of error classification reduces the contribution to FIT by no longer considering most soft errors as failures; if a soft error has no effect, it can be corrected without any disruption.

In designs requiring the highest level of reliability, classification of soft errors in Configuration Memory is essential. This capability is provided by the SEM Controller.

Strategies for Mitigating Soft Errors in Configuration Memory

A soft error in Configuration Memory has the potential to significantly change the design, and unlike design state, the Configuration Memory is not periodically overwritten by a new value. The strategies in this section focus on mitigating the effects of soft errors in Configuration Memory. The strategies discussed are:

- Scheduled Maintenance Strategy, page 22
- Emergency Maintenance Strategy, page 22

- Running Repairs Strategy, page 23
- Combined Strategy, page 24

Based on the infrequent occurrence of soft errors, designers must carefully consider the disadvantages and costs of adopting a particular strategy as well as its advantages.

Scheduled Maintenance Strategy

When there is a soft error in a Configuration Memory bit, the effect on the design can be instantaneous or irregularities might not be noticed for a significant time. For example, an error affecting clocks or resets can have an effect almost immediately, but a change to a circuit used to display the hours on a clock might not become apparent for hours.

Some parts of an application are more important than others; it is the important parts to which precautions are applied. It is useful to assess the time a soft error takes to affect a function and to consider the consequences that a failure can have. An important question to ask is if the system is required to maintain operation after a soft error, or is it only required to fail safely? The answer determines the appropriate precautions to take.

When a device is reconfigured, all Configuration Memory bits are written, regardless of the previous state. As a result, previously existing soft errors are removed. Although there are some applications that operate continuously for very long periods of time, very few are required to operate continuously without interruption for the entire product lifetime.

Most applications experience relatively frequent power cycles or periods of inactivity when maintenance can be performed. The scheduled maintenance strategy is intended to fully exploit these opportunities. The best use of the scheduled maintenance strategy exploits every available opportunity to reconfigure the device during normal operation, and reconfigure when it is not playing an active role in the system. No attempt is made to determine if a soft error has occurred; the reconfiguration simply repairs any corruption, if it exists. This is the same concept as performing regular maintenance on a vehicle, where certain parts are replaced at regular intervals even if they appear to be perfectly serviceable.

Even if all errors are corrected by the next scheduled reconfiguration, what happens for the period of time between a soft error and the next reconfiguration? This is when precautions must either maintain service or fail safely as the application requires.

Scheduled reconfiguration corrects any errors that have occurred, and if a design contains precautions to cope with soft errors, this scheme can be acceptable in many applications.

Emergency Maintenance Strategy

The concept behind the emergency maintenance strategy is the same as a vehicle's "check engine" indicator, which offers notification that a serious issue may exist. The appropriate response is to investigate the problem immediately, rather than wait until the next scheduled maintenance. This means advancing the next reconfiguration of the device when a soft error is detected in Configuration Memory.

The key to the emergency maintenance strategy is detecting if a soft error has occurred in the Configuration Memory. Recent Xilinx devices provide an integrated soft error detection capability for this purpose. When this is used for detection only, it facilitates simple implementation of an emergency maintenance strategy using a CRC-based error detection signal. A description of this feature and how to use it is provided in UG470, *7 Series FPGAs Configuration User Guide*, UG360, *Virtex-6 FPGA Configuration User Guide* and UG380, *Spartan-6 FPGA Configuration User Guide*.

The time to detect the presence of a soft error in Configuration Memory with a CRC-based method depends on the rate at which the integrated soft error detection circuit scans the memory array, as well as the location of the soft error with respect to the scanning process. While the best case represents a fairly immediate detection, the worst case may require nearly two full device scans, with the average time to detection being one device scan. If the maximum detection time can be tolerated before a reconfiguration of the device is initiated, then emergency maintenance using the integrated soft error detection avoids the requirement for any special circuitry. However, shorter detection times are possible.

The SEM Controller uses a detection scheme based on both CRC and ECC. It can be used in a detection only mode. With this scheme, almost all soft errors are detected within one device scan because the ECC checks are performed with much finer granularity than the CRC check. The average detection time is half of a full device scan.

Error detection performed by either solution indicates a change in the Configuration Memory and does not (and cannot) indicate a change to design state. If parts of the design contain important state information, these parts still need precautionary circuits of their own.

In general, use the emergency maintenance strategy when the system can tolerate a configuration fault in the design for the duration of the soft error detection time, until reconfiguration is initiated. The system must also be tolerant of the device down time during reconfiguration.

Running Repairs Strategy

The running repairs strategy is useful in applications where it is desirable to maintain operation following a soft error, while carrying out localized repair of Configuration Memory content rather than performing a full device reconfiguration.

Given the high probability that a soft error will have no immediate effect on a particular design, this strategy avoids the interruption of service associated with the emergency maintenance strategy.

The integrated soft error detection capability of some Xilinx devices also includes soft error correction capability. When enabled, this feature allows the device to not only detect errors, but also correct the most common type, which are single-bit errors. A description of this feature and how to use it is provided in UG470, *7 Series FPGAs Configuration User Guide*, UG360, *Virtex-6 FPGA Configuration User Guide*, and UG380, *Spartan-6 FPGA Configuration User Guide*.

The SEM Controller expands on the error correction capability in Xilinx devices. The controller can repair single-bit errors using the same method as the integrated function. The controller can repair two-bit adjacent errors using a CRC-based enhanced repair algorithm. Or, the core can be configured to replace Configuration Memory content instead of repair it, enabling correction of soft errors of arbitrary size as long as the location of the errors can be identified. This provides extensive correction capability.

Although the running repairs strategy can detect and repair soft errors while the design is active, any soft errors that occur will exist for a short period of time until their correction. A simple, yet very conservative approach is to assume that all soft errors can affect the proper operation of the design. One potential solution is to issue a reset to all important circuits upon detection of an error and then remove the reset after the error has been corrected. Although this interrupts normal operation, it will be of significantly shorter duration than the time required for a full reconfiguration of the device and does not require the additional resource usage associated with design redundancy.

To further minimize interruption of normal operation, the SEM Controller offers an optional soft error classification capability. When this capability is enabled and a soft error is corrected, the core performs a look up to determine if the bit(s) affected by the soft error were essential to the proper operation of the design. If the correction involved any essential bits, then the error is considered "essential" and may warrant additional action. If the error is "non-essential," the soft error will have no effect on the design operation, and the error can be corrected while the design continues to operate normally.

Combined Strategy

Systems requiring the very highest reliability benefit from using a combination of scheduled maintenance, emergency maintenance, and running repairs strategies. This multi-layered approach continues the redundancy concept by using each strategy to provide cover for the next.

When considering the most beneficial combinations for a system, the operation of an aircraft is a useful analogy when reliability is paramount. Once in service, regular maintenance of the aircraft includes the replacement of parts even if there is no visible evidence of a problem. This prevents failures due to wear over time as well as fixing defects missed during in situ inspections. Reconfiguring a device at regular intervals offers similar advantages by correcting possible unseen errors (however remote a possibility) and ensuring a clean start for each period of operation.

Before each flight, checks are made to ensure all important systems on the aircraft are working properly. If the check reveals a fault, the aircraft is grounded until the fault is corrected. Using the built-in error detection to continuously check the Configuration Memory performs a similar function. If a system is in a standby mode and an error is detected, an emergency reconfiguration can be invoked to repair the fault. This avoids the potential of accumulating errors over time and ensures the device is in perfect condition when it enters an operational mode.

Once an aircraft is in flight, any failure is undesirable. An active warning light alerts the crew to the nature of the fault so that they can take appropriate actions. These actions vary depending on the severity and location of the failure. It might be possible to contain the problem and continue to the planned destination, or a diversion and an emergency landing might be required. The most challenging scenario is when the plane must maintain flight because there is no suitable landing site, for example, in the middle of the ocean. In all cases, once the plane has landed, the aircraft is removed from service and repaired before being flown again. This applies even if the fault was considered to be temporary and resolved by the crew during the flight.

In a Xilinx device, the warning is provided when the error detection logic identifies a soft error. In most cases, the soft error does not affect the operation of the design. However, the cases that do impact operation require that appropriate action is taken. If the device can be safely reconfigured immediately, this is the obvious response to the warning. If continuous operation must be maintained, then redundancy and localized repairs (using capability of the SEM Controller) will be required.

XILINX®

Chapter 3

Core Overview

This chapter overviews the function of the SEM Controller and the interfaces it exposes. The controller is highly flexible and may be designed into a variety of applications.

The SEM Controller provides users with a method to better manage the system-level effects of soft errors in Configuration Memory. Intelligent management of these events increases reliability and availability and reduces maintenance and downtime costs. The controller has been designed to directly support the "running repairs" strategy although using it to implement other strategies is clearly possible.

As the term "mitigation" implies, the SEM Controller does not prevent soft errors. The controller does not operate on soft errors in Block Memory, Distributed Memory, or Flip Flops. Soft error mitigation in these resources must be addressed by inclusion of precautionary measures in the design such as redundancy or error detection and correction codes.

Overview

The SEM Controller implements five main functions: initialization, error injection, error detection, error correction, and error classification. All functions, except initialization and detection, are optional; desired functions are selected during the IP core configuration and generation process in the Xilinx CORE Generator.

The controller initializes by bringing the integrated soft error detection capability of the FPGA into a known state after the FPGA enters user mode. After this initialization, the controller endlessly loops, observing the integrated soft error detection status. When an ECC or CRC error is detected, the controller evaluates the situation to identify the Configuration Memory location involved.

Once this is complete, the controller may optionally correct the soft error by repairing it or by replacing the affected bits. The repair methods are active partial reconfiguration to perform a localized correction of Configuration Memory using a read-modify-write scheme. These methods use algorithms to identify the error in need of correction. The replace method is also active partial reconfiguration with the same goal, but this method uses a write-only scheme to replace Configuration Memory with original data. This data is provided by the implementation tools and stored outside the controller.

The controller may optionally classify the soft error as essential or non-essential using a lookup table. The lookup table is stored outside the controller and is fetched as required during execution of error classification. This data is also provided by the implementation tools and stored outside the controller.

When the controller is idle, there is an option to accept input from the user to inject errors into Configuration Memory. This function is useful for testing the integration of the controller into a larger system design. Using the error injection capability, system

verification and validation engineers may construct test cases to ensure the complete system responds to soft error events as expected.

Controller Interfaces

The SEM Controller is the kernel of the soft error mitigation solution. Figure 3-1 shows the SEM Controller ports. These ports are clustered into six groups. Shading indicates port groups that only exist in certain configurations.

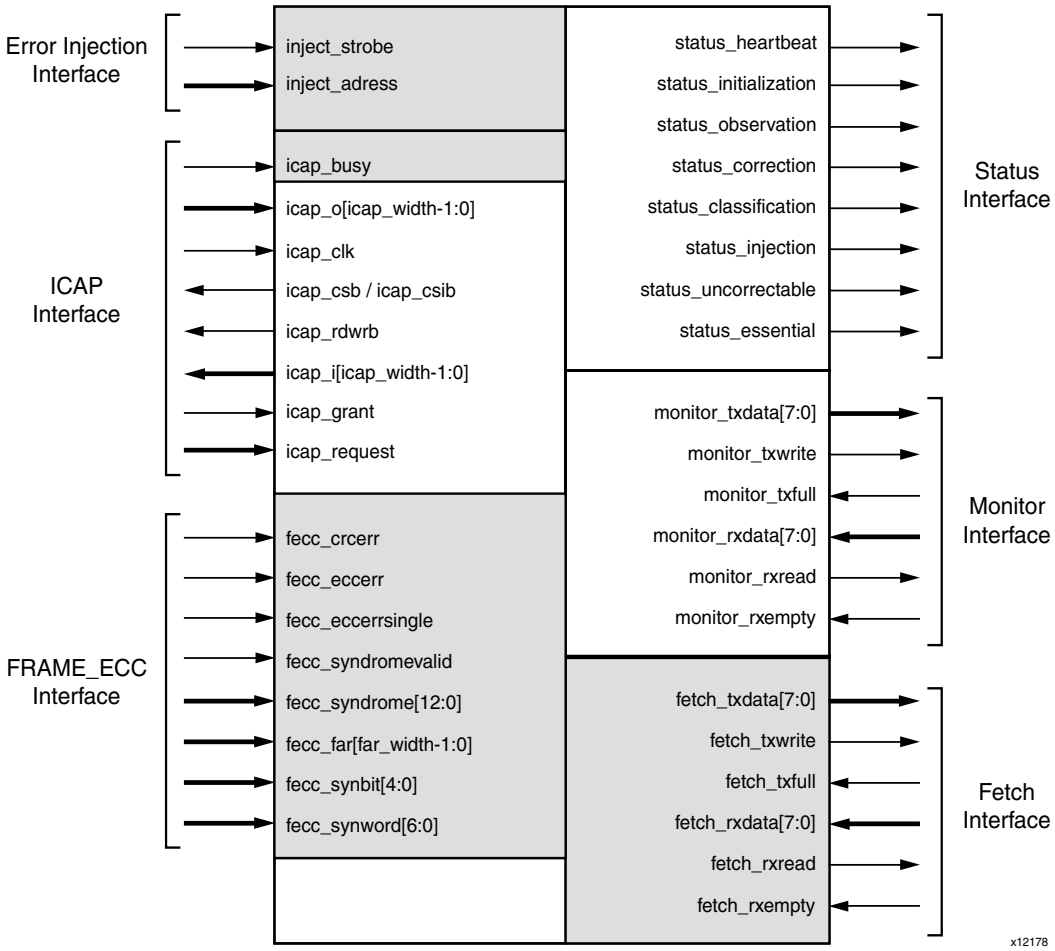

Figure 3-1: **SEM Controller Ports**

The SEM Controller has no reset input or output. It automatically initializes itself with an internal synchronous reset derived from the de-assertion of the global GSR signal.

The controller is a fully synchronous design using `icap_clk` as the single clock. All state elements are synchronous to the rising edge of this clock. As a result, all interfaces are also synchronous to the rising edge of this clock.

ICAP Interface

The ICAP Interface is a point-to-point connection between the SEM Controller and the ICAP primitive. The ICAP primitive enables read and write access to the registers inside the FPGA configuration system. The ICAP primitive and the behavior of the signals on this interface are described in UG470, *7 Series FPGAs Configuration User Guide*, UG360, *Virtex-6*

FPGA Configuration User Guide, and UG380, *Spartan®-6 FPGA Configuration User Guide*. For 7 series devices, `icap_busy` is not part of the ICAP Interface.

Table 3-1: **ICAP Interface Signals**

Name	Sense	Direction	Description
icap_busy	HIGH	IN	Receives BUSY output of ICAP. For 7 series devices, icap_busy is not part of the ICAP Interface.
icap_o[icap_width-1:0]	HIGH	IN	Receives O output of ICAP. The variable icap_width is equal to 32 for 7 series and Virtex-6 devices and 16 for Spartan-6 devices.
icap_csib / icap_csb	LOW	OUT	Drives CSIB (7 series) / CSB (Virtex-6 and Spartan-6) input of ICAP.
icap_rdwrb	LOW	OUT	Drives RDWRB input of ICAP.
icap_i[icap_width-1:0]	HIGH	OUT	Drives I input of ICAP. The variable icap_width is equal to 32 for 7 series and Virtex-6 devices and 16 for Spartan-6 devices.
icap_clk	EDGE	IN	Receives the clock for the design. This same clock also must be applied to the CLK input of ICAP. The clock frequency must comply with the ICAP input clock requirements as specified in the target device data sheet.
icap_request	HIGH	OUT	This signal is reserved for future use. Leave this port OPEN.
icap_grant	HIGH	IN	This signal is reserved for future use. Tie this port to VCC.

FRAME_ECC Interface

The FRAME_ECC Interface is a point-to-point connection between the SEM Controller and the FRAME_ECC primitive. The FRAME_ECC primitive is an output-only primitive that provides a window into the soft error detection function in the FPGA configuration system. The FRAME_ECC primitive and the behavior of the signals on this interface are described in UG470, *7 Series FPGAs Configuration User Guide*, and UG360, *Virtex-6 FPGA Configuration User Guide*.

For Spartan-6 devices, the FRAME_ECC primitive and the soft error detection functionality are implemented in soft logic within the SEM Controller. This logic implements the same capabilities as present in Virtex-6 FPGAs. As a result, no FRAME_ECC Interface exists on implementations of the SEM Controller for Spartan-6 devices.

Table 3-2: **FRAME_ECC Interface Signals**

Name	Sense	Direction	Description
fecc_crcerr	HIGH	IN	Receives CRCERROR output of FRAME_ECC.
fecc_eccerr	HIGH	IN	Receives ECCERROR output of FRAME_ECC.

Table 3-2: **FRAME_ECC Interface Signals *(Cont'd)***

Name	Sense	Direction	Description
fecc_eccerrsingle	HIGH	IN	Receives ECCERRORSINGLE output of FRAME_ECC.
fecc_syndromevalid	HIGH	IN	Receives SYNDROMEVALID output of FRAME_ECC.
fecc_syndrome[12:0]	HIGH	IN	Receives SYNDROME output of FRAME_ECC.
fecc_far[far_width-1:0]	HIGH	IN	Receives FAR output of FRAME_ECC. The variable for far_width is 26 for 7 series devices and 24 for Virtex-6 devices.
fecc_synbit[4:0]	HIGH	IN	Receives SYNBIT output of FRAME_ECC.
fecc_synword[6:0]	HIGH	IN	Receives SYNWORD output of FRAME_ECC.

Status Interface

The Controller Status Interface provides a convenient set of decoded outputs that indicate, at a high level, what the controller is doing.

Table 3-3: **Status Interface Signals**

Name	Sense	Direction	Description
status_heartbeat	HIGH	OUT	The heartbeat signal is active while status_observation is true. This output will issue a single-cycle high pulse at least once every 128 clock cycles for 7 series and Virtex-6 devices, and at least once every 512 clock cycles for Spartan-6 devices. This signal may be used to implement an external watchdog timer to detect "controller stop" scenarios that may occur if the controller or clock distribution is perturbed by soft errors. When status_observation is false, the behavior of the heartbeat signal is unspecified.
status_initialization	HIGH	OUT	The initialization signal is active during controller initialization, which occurs one time after the design begins operation.
status_observation	HIGH	OUT	The observation signal is active during controller observation of error detection signals. This signal remains active after an error detection while the controller queries the hardware for information.
status_correction	HIGH	OUT	The correction signal is active during controller correction of an error or during transition through this controller state if correction is disabled.
status_classification	HIGH	OUT	The classification signal is active during controller classification of an error or during transition through this controller state if classification is disabled.
status_injection	HIGH	OUT	The injection signal is active during controller injection of an error. When an error injection is complete, and the controller is ready to inject another error or return to observation, this signal returns inactive.

Table 3-3: **Status Interface Signals**

Name	Sense	Direction	Description
status_essential	HIGH	OUT	The essential signal is an error classification status signal. Prior to exiting the classification state, the controller will set this signal to reflect whether the error occurred on an essential bit(s). Then, the controller exits classification state.
status_uncorrectable	HIGH	OUT	The uncorrectable signal is an error correction status signal. Prior to exiting the correction state, the controller will set this signal to reflect the correctability of the error. Then, the controller exits correction state.

The `status_heartbeat` output provides an indication that the controller is active. Although the controller mitigates soft errors, it can also be disrupted by soft errors. For example, the controller clock may be disabled by a soft error. If the `status_heartbeat` signal stops, the user can take remedial action.

The `status_initialization`, `status_observation`, `status_correction`, `status_classification`, and `status_injection` outputs indicate the current controller state. The `status_uncorrectable` and `status_essential` outputs qualify the nature of detected errors.

Two additional controller state may be decoded from the five controller state outputs. If all five signals are low, the controller is idle (inactive but ready to resume). If all five signals are high, the controller is halted (inactive due to fatal error).

Error Injection Interface

The controller Error Injection Interface provides a convenient set of inputs to command the controller to inject a bit error into configuration memory.

Table 3-4: **Error Injection Interface Signals**

Name	Sense	Direction	Description
inject_strobe	HIGH	IN	The error injection control is used to indicate an error injection request. The inject_strobe signal should be pulsed high for one cycle concurrent with the application of a valid address to the inject_address input. The error injection control must only be used when the controller is idle.
inject_address [inject_width-1:0]	HIGH	IN	The error injection address bus is used to specify the parameters for an error injection. The value on this bus is captured at the same time inject_strobe is sampled active. For 7 series devices, the variable inject_width is 40, and for Virtex-6 and Spartan-6 devices, the variable is 36.

The user must provide an error injection address and command on `inject_address[inject_width-1:0]` and assert `inject_strobe` to indicate an error injection.

In response, the controller will inject a bit error. The controller confirms receipt of the error injection command by asserting `status_injection`. When the injection command has completed, the controller de-asserts `status_injection`. To inject errors affecting multiple bits, a sequence of error injections can be performed.

For more information on error injection commands, see Error Injection Interface in Chapter 8.

Monitor Interface

The controller Monitor Interface provides a mechanism for the user to interact with the controller.

The controller is designed to read commands and write status information to this interface as ASCII strings. The status and command capability of the Monitor Interface is a superset of the Status Interface and the Error Injection Interface. The Monitor Interface is intended for use in processor based systems.

Table 3-5: **Monitor Interface Signals**

Name	Sense	Direction	Description
monitor_txdata[7:0]	HIGH	OUT	Parallel transmit data from controller.
monitor_txwrite	HIGH	OUT	Write strobe, qualifies validity of parallel transmit data.
monitor_txfull	HIGH	IN	This signal implements flow control on the transmit channel, from the shim (peripheral) to the controller.
monitor_rxdata[7:0]	HIGH	IN	Parallel receive data from the shim (peripheral).
monitor_rxread	HIGH	OUT	Read strobe, acknowledges receipt of parallel receive data.
monitor_rxempty	HIGH	IN	This signal implements flow control on the receive channel, from the shim (peripheral) to the controller.

Fetch Interface

The controller Fetch Interface provides a mechanism for the controller to request data from an external source.

During error correction and error classification, the controller may need to fetch a frame of configuration data or a frame of essential bit data. The controller is designed to write a command describing the desired data to the Fetch Interface in binary. The external source must use the information to fetch the data and return it to the Fetch Interface.

Table 3-6: **Fetch Interface Signals**

Name	Sense	Direction	Description
fetch_txdata[7:0]	HIGH	OUT	Parallel transmit data from controller.
fetch_txwrite	HIGH	OUT	Write strobe, qualifies validity of parallel transmit data.
fetch_txfull	HIGH	IN	This signal implements flow control on the transmit channel, from the shim (peripheral) to the controller.
fetch_rxdata[7:0]	HIGH	IN	Parallel receive data from the shim (peripheral).
fetch_rxread	HIGH	OUT	Read strobe, acknowledges receipt of parallel receive data.
fetch_rxempty	HIGH	IN	This signal implements flow control on the receive channel, from the shim (peripheral) to the controller.

XILINX®

Chapter 4

Example Design Overview

This chapter overviews the function of the SEM Controller system-level design example and the interfaces it exposes. The system-level design example encapsulates the controller and various shims that serve to interface the controller to other devices. These shims may include I/O Pins, ChipScope, I/O Interfaces, Memory Controllers, or application-specific system management interfaces.

As delivered, the system-level design example is not a reference design, but an integral part of the total solution and fully verified by Xilinx. While designers do have the flexibility to modify the system-level design example, the recommended approach is to use it as delivered.

Overview

In addition to serving as an instantiation vehicle for the controller itself, the system-level design example incorporates five main functions:

- The FPGA configuration system primitives and their connection to the controller.
- The MON shim, a bridge between the controller and a standard RS-232 port. The resulting interface may be used to exchange commands and status with the controller. This interface is designed for connection to processors.
- The EXT shim, a bridge between the controller and a standard SPI bus. The resulting interface may be used to fetch data by the controller. This shim is only present in certain controller configurations and is designed for connection to standard SPI flash.
- The HID shim, a bridge between the controller and an interface device. The resulting interface may be used to exchange commands and status with the controller. This shim is only present in certain controller configurations.

Figure 4-1 shows a block diagram of the system-level design example. The blocks drawn in gray only exist in certain configurations.

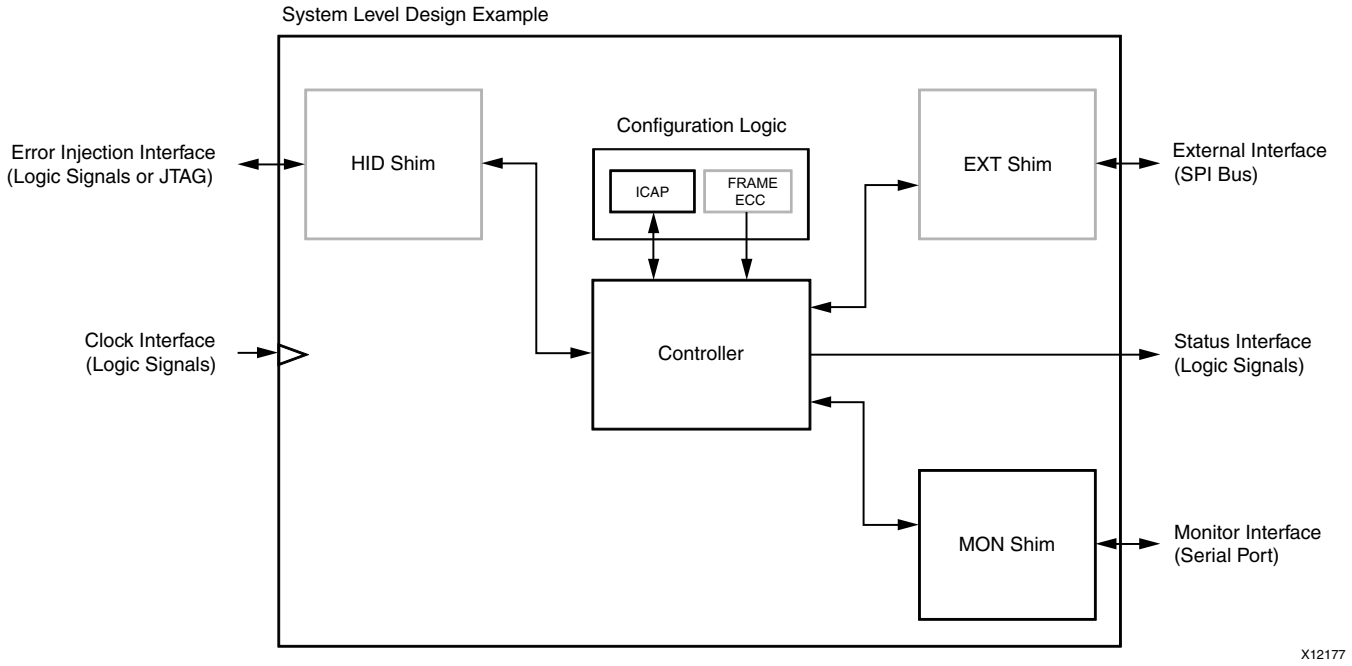

Figure 4-1: **Example Design Block Diagram**

The system-level design example is provided to allow flexibility in system-level interfacing. To support this goal, the system-level design example is provided as RTL source code, unlike the controller itself.

Example Design Interfaces

Figure 4-2 shows the example design ports. The ports are clustered into six groups. The groups shaded in gray only exist in certain configurations.

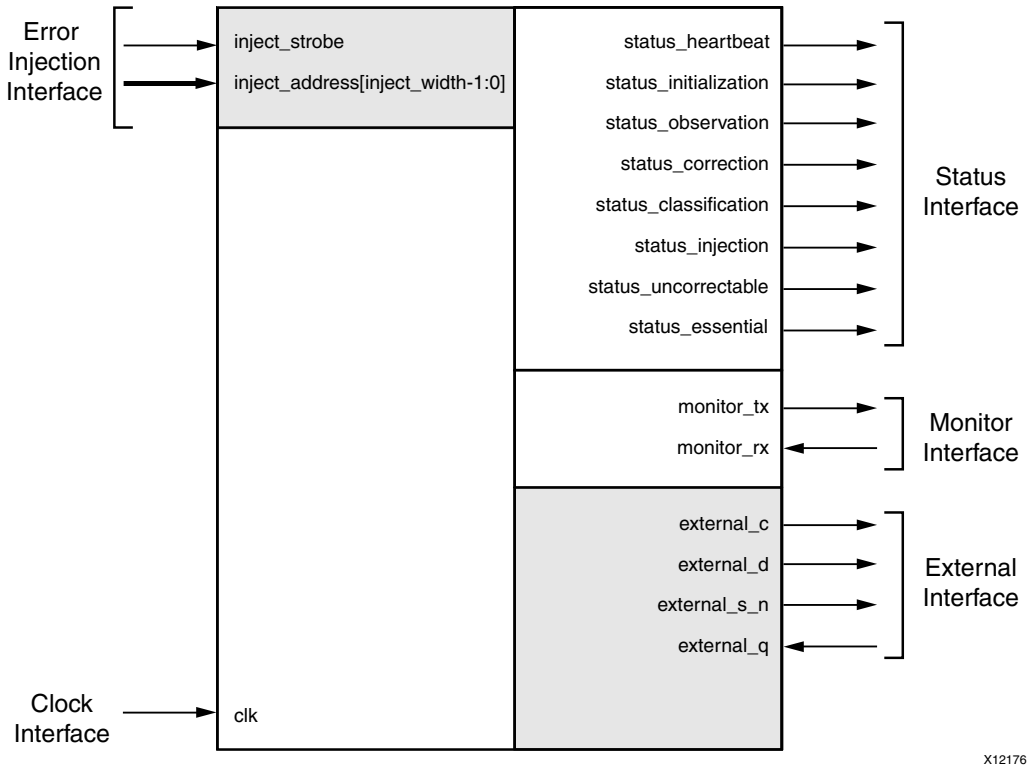

Figure 4-2: **Example Design Ports**

The system-level design example has no reset input or output. The controller automatically initializes itself. The controller then initializes the shims, as required.

The system-level design example is a fully synchronous design using `clk` as the single clock. All state elements are synchronous to the rising edge of this clock. As a result, the interfaces are generally synchronous to the rising edge of this clock.

Status Interface

The Status Interface is a direct pass-through of the controller Status Interface. See Chapter 3, Core Overview for a description of this interface.

Clock Interface

The Clock Interface is used to provide a clock to the system-level design example. Internally, the clock signal is distributed on a global clock buffer to all synchronous logic elements.

Table 4-1: **Clock Interface Details**

Name	Sense	Direction	Description
clk	EDGE	IN	Receives the master clock for the system-level design example.

Monitor Interface

The Monitor Interface is always present. The MON shim in the system-level design example is a UART. This shim serializes status information generated by the controller (a byte stream of ASCII codes) for serial transmission. Similarly, the shim de-serializes command information presented to the controller (a bit stream of ASCII codes) for parallel presentation to the controller.

The shim uses a standard serial communication protocol. The shim contains synchronization and over sampling logic to support asynchronous serial devices that operate at the same nominal baud rate. Refer to Chapter 8, Applying the Solution for additional information.

The resulting interface is directly compatible with a wide array of devices ranging from embedded microcontrollers to desktop computers. External level translators may be necessary depending on system requirements.

Table 4-2: **Monitor Interface Details**

Name	Sense	Direction	Description
monitor_tx	LOW	OUT	Serial transmit data from system-level design example.
monitor_rx	LOW	IN	Serial receive data to system-level design example.

External Interface

The External Interface is present when the controller requires access to external data. When present, the EXT shim in the system-level design example is a fixed-function SPI bus master. This shim accepts commands from the controller that consist of an address and a byte count. The shim generates SPI bus transactions to fetch the requested data from an external SPI flash. The shim formats the returned data for the controller to pick up.

The shim uses standard SPI bus protocol, implementing the most common mode (CPOL = 0, CPHA = 0, often referred to as "Mode 0"). The SPI bus clock frequency is locked to one half of the master clock for the system-level design example. See Chapter 8, Applying the Solution for information on external timing budgets.

The resulting interface is directly compatible with a wide array of standard SPI flash. External level translators may be necessary depending on system requirements.

Table 4-3: **External Interface Details**

Name	Sense	Direction	Description
external_c	EDGE	OUT	SPI bus clock for an external SPI flash.
external_d	HIGH	OUT	SPI bus "master out, slave in" signal for an external SPI flash.
external_s_n	LOW	OUT	SPI bus select signal for an external SPI flash.
external_q	HIGH	IN	SPI bus "master in, slave out" signal for an external SPI flash.

Error Injection Interface

The Error Injection Interface is present when the controller supports error injection and the HID shim is set to I/O Pins. This interface is a direct pass-through of the controller Error Injection Interface. See Chapter 3, Core Overview for a description of this interface.

When the controller supports error injection and the HID shim is set to ChipScope, or when error injection is disabled, this interface is absent.

Additional Information for I/O Pin Interfaces

When the system-level design example is implemented as delivered, its interfaces are mapped to external I/O pins with specific electrical characteristics. These are summarized in Table 4-4.

Table 4-4: **SEM Controller I/O Pin Interface Details**

Name	Driver Type	Pull	Load	Supply	Drive / Slew
Clock Interface					
clk	LVCMOS	NONE	N/A	2.5V/1.8V	N/A
Status Interface					
status_heartbeat	LVCMOS	NONE	UNSPECIFIED	2.5V/1.8V	4 mA / SLOW
status_initialization	LVCMOS	NONE	UNSPECIFIED	2.5V/1.8V	4 mA / SLOW
status_observation	LVCMOS	NONE	UNSPECIFIED	2.5V/1.8V	4 mA / SLOW
status_correction	LVCMOS	NONE	UNSPECIFIED	2.5V/1.8V	4 mA / SLOW
status_classification	LVCMOS	NONE	UNSPECIFIED	2.5V/1.8V	4 mA / SLOW
status_injection	LVCMOS	NONE	UNSPECIFIED	2.5V/1.8V	4 mA / SLOW
status_essential	LVCMOS	NONE	UNSPECIFIED	2.5V/1.8V	4 mA / SLOW
status_uncorrectable	LVCMOS	NONE	UNSPECIFIED	2.5V/1.8V	4 mA / SLOW
Monitor Interface					
monitor_tx	LVCMOS	NONE	10 pF	2.5V/1.8V	4 mA / SLOW
monitor_rx	LVCMOS	NONE	N/A	2.5V/1.8V	N/A
External Interface (Optional)					
external_c	LVCMOS	NONE	10 pF	2.5V/1.8V	8 mA / FAST
external_d	LVCMOS	NONE	10 pF	2.5V/1.8V	8 mA / FAST
external_s_n	LVCMOS	NONE	10 pF	2.5V/1.8V	8 mA / FAST
external_q	LVCMOS	NONE	N/A	2.5V/1.8V	N/A
Error Injection Interface (Optional)					
inject_strobe	LVCMOS	NONE	N/A	2.5V/1.8V	N/A
inject_address [inject_width-1:0]	LVCMOS	NONE	N/A	2.5V/1.8V	N/A

XILINX®

Chapter 5

Generating the Solution

This chapter provides instructions for generating the SEM Controller solution. The generation instructions are annotated with detailed information about each of the available options.

Creating a Project

First, create a project using the Xilinx CORE Generator software. For detailed information on starting and using the CORE Generator software, see the Xilinx CORE Generator User Guide. Perform the following steps:

1. Start the CORE Generator software.
2. Choose **File** > **New Project** from the menu.
3. Using the file requestor dialog box:
   a. Navigate to the desired project directory.
   b. Modify the project file name, if desired.
   c. Click **Save**.
4. Set the Part Options in the Project Options dialog box:
   a. Select the target family, device, package, and speed grade.

      Example: Kintex-7, xc7k325t-ffg900-1

      Note: If an unsupported family is selected, the IP core will not appear in the IP Catalog.

   b. Click **Apply**.
5. Set the Generation Options in the Project Options dialog box:
   a. For Flow, select Design Entry in either VHDL or Verilog.
   b. For Flow Settings, select either ISE (for XST) or Synplicity (for Synplify).
   c. Click **Okay**.

After creating the project, the IP core will be available for selection in the IP Catalog, located at **FPGA Features and Design** > **Soft Error Mitigation** > **Soft Error Mitigation**.

Configuring the Solution

Locate the IP core in the IP Catalog and click it once to select it. Important information regarding the solution is displayed in the main window. Please review this information before proceeding.

In the Actions section of the main window, click on the Customize and Generate link. This launches page one of the solution configuration dialog box, shown in Figure 5-1.

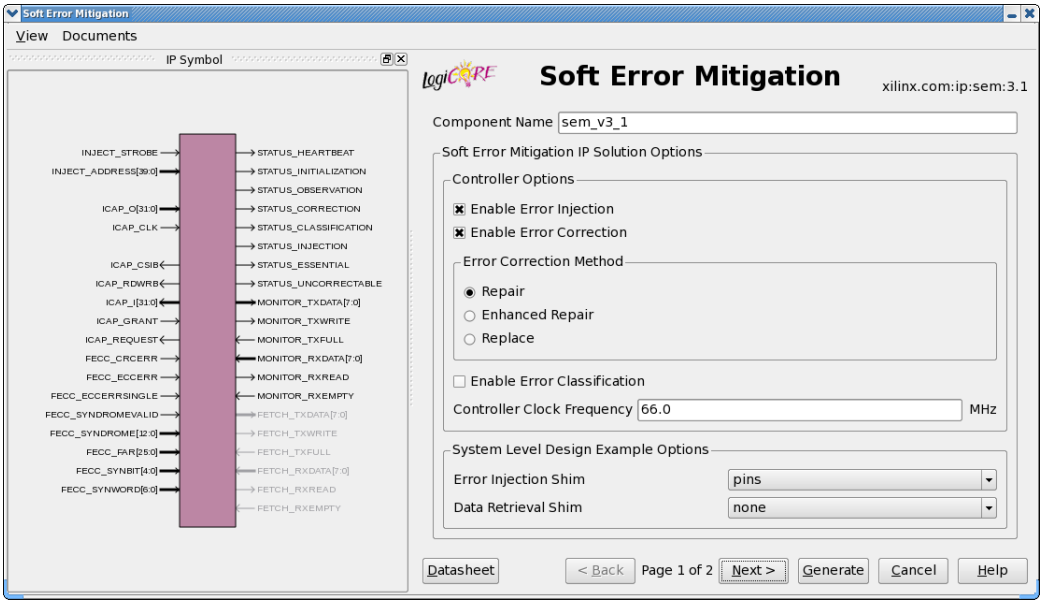

Figure 5-1: **Solution Configuration Dialog Box Page 1**

Review each of the available options, and modify them as desired so that the SEM Controller solution meets the requirements of the larger project into which it will be integrated. The following sub-sections discuss the options in detail to serve as a guide.

Component Name and Symbol

The name of the generated component is set by the Component Name field. The name "sem_v3_1" is used in this example.

The Component Symbol occupies the left half of the dialog box and provides a visual indication of the ports that will exist on the component, given the current option settings. Ports that will be generated are drawn in black. This diagram is automatically updated when the option settings are modified.

Controller Options: Enable Error Injection

The Enable Error Injection checkbox is used to enable or disable the error injection feature. Error injection is a design verification function that provides a mechanism for users to create errors in Configuration Memory that model a soft error event. This is useful during integration or system level testing to verify that the controller has been properly interfaced with system supervisory logic and that the system responds as desired when a soft error event occurs.

If error injection is enabled, the Error Injection Interface is generated (as indicated by the Component Symbol) and the controller will perform error injections in response to commands from the user. These commands may be applied to the Error Injection Interface or to the Monitor Interface.

If error injection is disabled, the Error Injection Interface is removed (as indicated by the Component Symbol) and the controller will not perform any error injections. Note that the Monitor Interface continues to exist even if the Error Injection Interface is removed. If error

injection commands are applied to the Monitor Interface, the controller will parse the commands but otherwise ignore them.

Controller Options: Enable Error Correction

The Enable Error Correction checkbox is used to enable or disable the error correction feature. No matter what setting is used, the controller will monitor the error detection circuits and report error conditions.

If error correction is enabled, the controller will attempt to correct errors that are detected. The method by which corrections are performed is selectable. Most errors are correctable, and upon successful correction, the controller signals that a correctable error has occurred and was corrected. For errors that are not correctable, the controller signals that an uncorrectable error has occurred.

If error correction is disabled, the controller does not attempt to correct errors. At the first error event, the controller will stop scanning after reporting the error condition. Further, when error correction is disabled, the error classification feature is also disabled.

Controller Options: Error Correction Method

With error correction enabled, the error correction method is selectable. The available methods are correction by repair, correction by enhanced repair, and correction by replace. The controller will correct errors using the method selected by the user. The correction possibilities are:

Correction by Repair

- Correction by repair for one-bit errors. The ECC syndrome is used to identify the exact location of the error in a frame. The frame containing the error is read, the relevant bit inverted, and the frame is written back. This is signaled as correctable.

Correction by Enhanced Repair

- Correction by repair for one-bit and two-bit adjacent errors. For one-bit errors, the behavior is identical to correction by repair. For two-bit errors, an enhanced CRC-based algorithm capable of correcting two-bit adjacent errors is used. The frame containing the error is read, the relevant bits inverted, and the frame is written back. This is signaled as correctable.

Correction by Replace

- Correction by replace for one-bit and multi-bit errors. The frame containing the error is read. The controller requests replacement data for the entire frame from the Fetch Interface. When the replacement data is available, the controller compares the damaged frame with the replacement frame to identify the location of the errors. Then, the replacement data is written back. This is signaled as correctable.

The difference between repair, enhanced repair, and replace methods becomes clear when considering what happens in the uncommon case of errors involving more bits than can be corrected by the repair or enhanced repair methods. In these cases, the repair or enhanced repair methods will not yield a successful correction. However, if such errors are corrected by the replace method, the error is correctable regardless of how many bits were affected.

The fundamental tradeoff between the methods is error correction success rate versus the cost of adding or increasing data storage requirements. Using correction by repair as the baseline for comparison:

- Correction by enhanced repair provides superior correction capability through use of additional internal Block RAM.
- Correction by replace provides ultimate correction capability through the addition of external SPI Flash.

EasyPath devices are not compatible with the error correction by replace method.

Controller Options: Enable Error Classification

The Enable Error Classification checkbox is used to enable or disable the error classification feature. Error classification is automatically disabled if error correction is disabled.

The error classification feature uses the Xilinx Essential Bits technology to determine whether a detected and corrected soft error has affected the function of a user design. Essential Bits are those bits that have an association with the circuitry of the design. If an Essential Bit changes, it will change the design circuitry. However it may not necessarily affect the function of the design.

Without knowing which bits are essential, the system must assume any detected soft error has compromised the correctness of the design. The system level mitigation behavior will often result in disruption or degradation of service until the FPGA configuration is repaired and the design is reset or restarted.

For example, if bitgen reports that 20% of the Configuration Memory is essential to an operation of a design, then only 2 out of every 10 soft errors (on average) actually merits a system-level mitigation response. The error classification feature is a table lookup to determine if a soft error event has affected essential Configuration Memory locations. Use of this feature reduces the effective FIT of the design. The cost of enabling this feature is the external storage required to hold the lookup table. When error classification is enabled, the Fetch Interface is generated (as indicated by the Component Symbol) so that the controller has an interface through which it can retrieve external data.

If error classification is enabled, and a detected error has been corrected, the controller will look up the error location. Depending on the information in the table, the controller will either report the error as essential or non-essential. If a detected error cannot be corrected, this is because the error cannot be located. Therefore, the controller conservatively reports the error as essential since it has no way to look up data to indicate otherwise.

If error classification is disabled, the controller unconditionally reports all errors as essential since it has no data to indicate otherwise.

Note that error classification need not be performed by the controller. It is possible to disable error classification by the controller, and implement it elsewhere in the system using the essential bit data provided by the implementation tools and the error report messages issued by the controller through the Monitor Interface.

Controller Options: Controller Clock Frequency

The controller clock frequency is set by the Clock Frequency field. The error mitigation time decreases as the controller clock frequency increases. Therefore, the frequency should be as high as practical, up to a limit posed by the Configuration Memory system. The dialog box will warn if the desired frequency exceeds the capability of the target device.

For designs that require a data retrieval interface to fetch external data for error classification or error correction by replace, an additional consideration exists. The example design implements an external memory interface that is synchronous to the

controller. The controller clock frequency therefore also determines the external memory cycle time. The external memory system must be analyzed to determine its minimum cycle time, as it may limit the maximum controller clock frequency.

Instructions on how to perform this analysis are located in Interface Level in Chapter 8. However, this analysis requires timing data from implementation results. Therefore, Xilinx recommends the following:

1. Generate the solution using the default frequency setting.
2. Extract the required timing data from the implementation results.
3. Complete the timing budget analysis to determine maximum frequency.
4. Re-generate the solution with a frequency at or below the calculated maximum frequency of operation.

Example Design Options: Error Injection Shim

For configurations that support error injection, the example design provides two options for a shim to external control of the Error Injection Interface:

- Direct control through physical pins
- Indirect control through JTAG using Xilinx ChipScope

When selecting ChipScope to control the Error Injection Interface, the ChipScope ICON and ChipScope VIO cores are not included. They must be generated (separately) with their output products placed in the example design directory. The necessary ChipScope core configurations are described at the end of this chapter.

Example Design Options: Data Retrieval Shim

For configurations that require a data retrieval interface to fetch external data for error classification or error correction by replace, the controller Fetch Interface must be bridged to an external storage device. The example design provides a shim to an external SPI Flash device as the only option. If the data retrieval interface is not required, no shim is generated.

Reviewing the Configuration

Proceed to page two of the of the solution configuration dialog box by clicking Next. This page is shown in Figure 5-2.

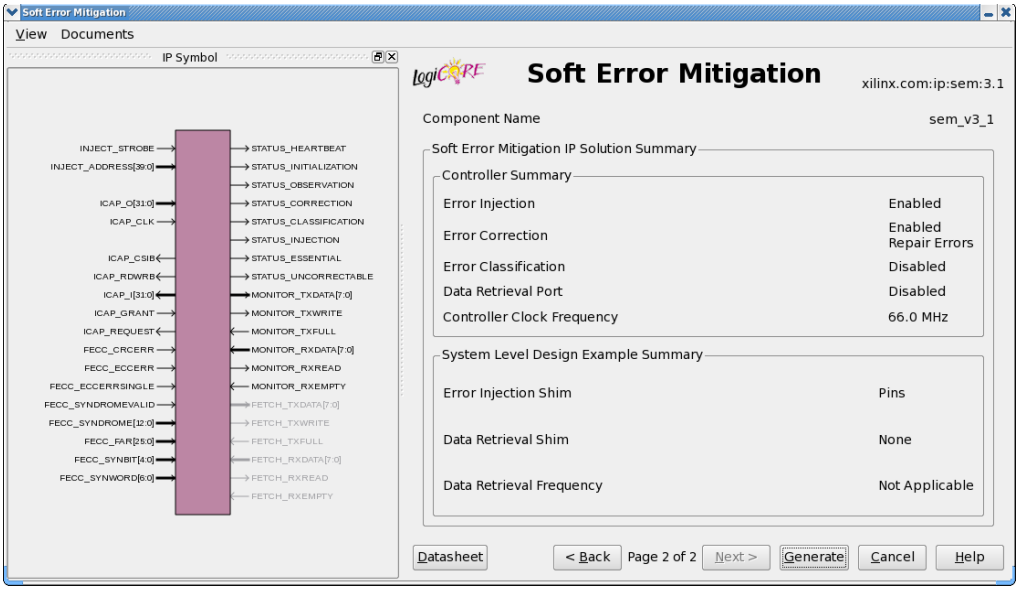

Figure 5-2: **Solution Configuration Dialog Box Page 2**

Review the summary to confirm each option is correct. Return to the previous page, if necessary, to correct or change options prior to generation. After the options are reviewed and correct, proceed to generate the solution.

Generating the Solution

Click Generate to begin the generation process. The CORE Generator software will generate and deliver a customized solution based on the provided option settings. When this process has completed, a final dialog box with important information regarding the solution will appear. Please review this information before exiting the CORE Generator software to review the delivered files.

Reviewing the Deliverables

The SEM Controller deliverables are organized in the directory structure shown below. Click a directory name to go to the description of the directory and the files it contains.

<project directory>
Top-level project directory; name is user-defined

<project directory>/<component name>
Release notes file

<component name>/doc
Product documentation

<component name>/example design

Verilog or VHDL design files

<component name>/implement
Implementation script files

<component name>/implement/results
Results directory, created after implementation scripts are run, and contains implement script results

<component name>/implement/synplify
Synthesis results when Synplify is used

<component name>/implement/xst
Synthesis results when XST is used

<project directory>

The <project directory> contains all the CORE Generator project files.

Table 5-1: **Project Directory**

Name	Description
<project_dir>	
<component_name>.xco	Configuration options file.
<component_name>.ngc	SEM Controller implementation netlist.
<component_name>.{v|vhd}	SEM Controller simulation netlist.
<component_name>.{veo|vho}	SEM Controller instantiation template.
<component_name>_flist.txt	List of files delivered with the solution.

Back to Top

<project directory>/<component name>

The component name directory contains the release notes in the readme file provided with the core, which can include tool requirements, updates, and issue resolution.

Table 5-2: **Component Name Directory**

Name	Description
<project_dir>/<component_name>	
sem_v3_1_readme.txt	Release notes file.

Back to Top

<component name>/doc

The doc directory contains the PDF documentation provided with the core.

Table 5-3: **Doc Directory**

Name	Description
<project_dir>/<component_name>/doc	
ds796_sem.pdf	Soft Error Mitigation Controller Data Sheet
ug764_sem.pdf	Soft Error Mitigation Controller User Guide

Back to Top

<component name>/example design

The example design directory contains the example design files provided with the core.

Table 5-4: **Example Design Directory**

Name	Description
<project_dir>/<component_name>/example_design	
sem_cfg.{v|vhd}	Example design configuration logic.
sem_example.{v|vhd}	Example design top level.
sem_example.ucf	Example design constraints file.
sem_mon.{v|vhd}	Example design MON shim.
sem_mon_fifo.{v|vhd}	Example design MON shim FIFO sub-block.
sem_mon_piso.{v|vhd}	Example design MON shim PISO sub-block.
sem_mon_sipo.{v|vhd}	Example design MON shim SIPO sub-block.
sem_ext.{v|vhd}	Example design EXT shim. This file is only generated if error classification or error correction by replace are enabled.
sem_ext_byte.{v|vhd}	Example design EXT shim byte transfer sub-block. This file is only generated if error classification or error correction by replace are enabled.
sem_hid.{v|vhd}	Example design HID shim. This file is only generated if error injection using ChipScope is enabled.
icon_bscan_bufg.{v|vhd}	Example design HID shim sub-block. This file is only generated for Virtex-6 or Spartan-6 if error injection using ChipScope is enabled.

Back to Top

<component name>/implement

The implement directory contains the core implementation script files.

Table 5-5: **Implement Directory**

Name	Description
<project_dir>/<component_name>/implement	
implement.bat	Windows batch file implementation script.
implement.sh	Linux batch file implementation script.
makedata.tcl	Bitgen data formatter for essential bit lookup table and correction by replace data. This file is only generated if error classification or error correction by replace are enabled.
synplify.prj	Synplify synthesis script, only generated if the project option flow setting is Synplicity.
xst.prj	XST synthesis project, only generated if the project option flow setting is ISE.
xst.scr	XST synthesis script, only generated if the project option flow setting is ISE.

Back to Top

<component name>/implement/results

The results directory is created by the implement script. The implementation results are placed in the results directory.

<component name>/implement/synplify

The synplify directory is created by the Synplify script. The synthesis results are placed in the synplify directory.

<component name>/implement/xst

The xst directory is created by the XST script. The synthesis results are placed in the xst directory.

Generating ChipScope Files

This section describes requirements for ChipScope files necessary to support the optional error injection feature in the specific configuration where ChipScope is selected as the error injection shim.

7 Series Devices

The ChipScope ICON core must be generated for the target device, using the default core name “chipscope_icon”, with the following parameters:

```
ENABLE_JTAG_BUFG = TRUE
NUMBER_CONTROL_PORTS = 1
USE_EXT_BSCAN = FALSE
USE_SOFTBSCAN= FALSE
```

```
USE_UNUSED_BSCAN = FALSE
```

The USER_SCAN_CHAIN may be set as desired. After the ICON core is generated, the `chipscope_icon.ngc` and `chipscope_icon.{v|vhd}` files must be copied to the example design directory.

The ChipScope VIO core must be generated for the target device, using the default core name "chipscope_vio", with the following parameters:

```
ENABLE_ASYNCHRONOUS_INPUT_PORT = FALSE
ENABLE_ASYNCHRONOUS_OUTPUT_PORT = FALSE
ENABLE_SYNCHRONOUS_INPUT_PORT = TRUE
ENABLE_SYNCHRONOUS_OUTPUT_PORT = TRUE
INVERT_CLOCK_INPUT = FALSE
SYNCHRONOUS_INPUT_PORT_WIDTH = 8
SYNCHRONOUS_OUTPUT_PORT_WIDTH = 41
```

After the VIO core is generated, the `chipscope_vio.ngc` and `chipscope_vio.{v|vhd}` files must be copied to the example design directory. The instantiation and interconnection of the ICON and VIO components inside the HID shim may be inspected, if desired. The mapping of the VIO synchronous input and synchronous output ports is:

```
sync_in[7] receives status_heartbeat
sync_in[6] receives status_uncorrectable
sync_in[5] receives status_essential
sync_in[4] receives status_injection
sync_in[3] receives status_classification
sync_in[2] receives status_correction
sync_in[1] receives status_observation
sync_in[0] receives status_initialization
sync_out[40] drives inject_strobe
sync_out[39:0] drives inject_address[39:0]
```

Virtex-6 and Spartan-6 Devices

The ChipScope ICON core must be generated for the target device, using the default core name "chipscope_icon", with the following parameters:

```
ENABLE_JTAG_BUFG = TRUE
NUMBER_CONTROL_PORTS = 1
USE_EXT_BSCAN = FALSE
USE_SOFTBSCAN= FALSE
USE_UNUSED_BSCAN = FALSE
```

The USER_SCAN_CHAIN may be set as desired. After the ICON core is generated, the `chipscope_icon.ngc` and `chipscope_icon.{v|vhd}` files must be copied to the example design directory.

The ChipScope VIO core must be generated for the target device, using the default core name "chipscope_vio", with the following parameters:

```
ENABLE_ASYNCHRONOUS_INPUT_PORT = FALSE
ENABLE_ASYNCHRONOUS_OUTPUT_PORT = FALSE
ENABLE_SYNCHRONOUS_INPUT_PORT = TRUE
ENABLE_SYNCHRONOUS_OUTPUT_PORT = TRUE
INVERT_CLOCK_INPUT = FALSE
SYNCHRONOUS_INPUT_PORT_WIDTH = 9
SYNCHRONOUS_OUTPUT_PORT_WIDTH = 37
```

After the VIO core is generated, the `chipscope_vio.ngc` and `chipscope_vio.{v|vhd}` files must be copied to the example design directory.

The instantiation and interconnection of the ICON and VIO components inside the HID shim may be inspected, if desired. The mapping of the VIO synchronous input and synchronous output ports is:

```
sync_in[8] is reserved, and is tied low
sync_in[7] receives status_heartbeat
sync_in[6] receives status_uncorrectable
sync_in[5] receives status_essential
sync_in[4] receives status_injection
sync_in[3] receives status_classification
sync_in[2] receives status_correction
sync_in[1] receives status_observation
sync_in[0] receives status_initialization
sync_out[36] drives inject_strobe
sync_out[35:0] drives inject_address[35:0]
```

Using ChipScope Analyzer

A project must be created in the ChipScope Analyzer software. The status signals received by the synchronous input port should be represented with LEDs. The inject_address bus output value should be represented as HEX for convenient data entry. The inject_strobe control output must be represented as a single PULSE output for proper operation.

XILINX®

Chapter 6

Building the Solution

After generating the solution, the resulting controller netlist and example design files can be used with a standard design flow.

Simulation of designs that instantiate the controller is supported. In other words, including the controller in a larger project does not adversely affect ability to run simulations of functionality unrelated to the controller. However, it is not possible to observe the controller behaviors in simulation. Simulation of a design including the controller will compile, but the controller will not exit the initialization state. Hardware-based evaluation of the controller behaviors is required.

For this reason, no simulation test bench is provided with the example design. Alternatively, customers can use ISim Hardware Co-simulation to simulate their design. Please contact Xilinx for more information.

Synthesis of designs that instantiate the controller is supported for Synopsys Synplify and Xilinx XST. The synthesis output can be processed using the Xilinx implementation tools. This chapter outlines the example design synthesis and implementation scripts.

Implementing the Example Design

To implement the example design, open a console window and type the following:

Windows

```
ms-dos> cd <project directory>\<component name>\implement
ms-dos> implement.bat
```

Linux

```
% cd <project directory>/<component name>/implement
% ./implement.sh
```

These commands change directory and execute an implementation script. The script synthesizes the design, performs physical implementation, and generates programming files. The result files are placed in the results directory. The following is an outline of the scripted processing sequence:

1. If the solution requires ChipScope files in the example design directory, the script verifies the files exist. If they do not exist, the script exits.
2. The script removes intermediate and result files from previous implementation runs.
3. The script synthesizes the project using Synopsys Synplify or Xilinx XST based on the project option settings used when the solution was generated.
4. The script performs a physical implementation:
   a. Xilinx ngdbuild application builds a design database

b. Xilinx map application maps the design to the target device
c. Xilinx par application places and routes the design
d. Xilinx trce application performs static timing analysis
e. Xilinx netgen application generates a timing simulation model
f. Xilinx bitgen application generates programming files

5. If the solution requires external data storage to support error classification or error correction by replace, an additional TCL script is called to post-process special bitgen output files into a SPI Flash programming file.
6. The script removes some intermediate files to clean up the results directory.

The script file starts from the example design source and the controller netlist and results in programming files. It is possible to use the Xilinx ISE graphical design environment to implement the example design, provided the implementation tool options used in the script are preserved and the additional TCL script is invoked manually, if it is required.

Creating the External Memory Programming File

When error correction by replace is enabled, an image of the configuration data is required. When error classification is enabled, an image of the essential bit lookup data is required. As a result, one or both of these data sets may be required. The data sets are identical in size, with their size a function of the target device. The data sets are generated by the bitgen application.

The format of the data is required to be binary, using the full data set(s) generated by bitgen. The external storage must be byte addressable. A small table is required at offset zero:

- Byte 0: 32-bit pointer to start of replacement data, byte 0 (least significant byte)
- Byte 1: 32-bit pointer to start of replacement data, byte 1
- Byte 2: 32-bit pointer to start of replacement data, byte 2
- Byte 3: 32-bit pointer to start of replacement data, byte 3 (most significant byte)
- Byte 4: 32-bit pointer to start of essential bit data, byte 0 (least significant byte)
- Byte 5: 32-bit pointer to start of essential bit data, byte 1
- Byte 6: 32-bit pointer to start of essential bit data, byte 2
- Byte 7: 32-bit pointer to start of essential bit data, byte 3 (most significant byte)
- The remaining bytes are reserved, filled with zero

A pointer value of 0x00000000 is used if a particular block of data is not present. After the table, the essential bit data and replacement data may be located at any offset provided each data set is contiguous.

The TCL script, which post processes the bitgen output files, generates three outputs:

- An Intel hex data file (MCS) for programming SPI Flash devices
- A raw binary data file (BIN) for programming SPI Flash devices
- An initialization file (VMF) for loading SPI Flash simulation models

XILINX®

Chapter 7

Design Constraints

The SEM Controller and the system-level design example require the specification of physical implementation constraints to yield a functional result that meets performance requirements. These constraints are provided with the system-level design example in a user constraints file (UCF).

To achieve consistent implementation results, the UCF provided with the solution must be used. These constraints have been tested in hardware and provide consistent results. For additional details on the definition and use of a UCF or specific constraints, see the applicable Constraints Guide available through the documentation page for the ISE Design Suite.

Constraints may require modification to integrate the solution into a larger project, or as a result of changes made to the system-level design example. Modifications should only be made with a thorough understanding of the effect of each constraint. Additionally, support is not provided for designs that deviate from the provided constraints.

Contents of the User Constraints File

Although the UCF delivered with each generated solution shares the same overall structure and sequence of constraints, the contents may vary based on options set at generation. The sections that follow define the structure and sequence of constraints using a Virtex-6 device implementation as an example.

Device Selection Constraint

The first section of the UCF specifies the exact device for the implementation tools to target, including the specific part, package, and speed grade. The device in the UCF reflects the device chosen in the CORE Generator software project. An example device selection constraint is:

```
CONFIG PART = XC6VLX240T-FF1156-1 ;
```

Although the Controller itself is designed to function in any of the supported devices, some details of the system-level example design depend on the selected part. For this reason, this constraint should not be modified. To target a different device, return to the CORE Generator software project, change the device, and re-generate the solution.

Controller Constraints

The controller, considered in isolation and regardless of options at generation, is a fully synchronous design. Fundamentally, it only requires a clock period constraint on the master clock input. In the generic UCF, this constraint is placed on the system-level design

example clock input and propagated into the controller. The constraint is discussed in Example Design Constraints, page 52.

The signal paths between the controller and the FPGA configuration system primitives must be considered as synchronous paths. By default, the paths between the ICAP primitive and the controller are analyzed as part of a clock period constraint on the master clock, because the ICAP clock pin is required to be connected to the same master clock signal.

However, the situation is different for the FRAME_ECC primitive (if required), as it does not have a clock pin. Based on the specific use of the FRAME_ECC primitive with the ICAP primitive, it is known that FRAME_ECC primitive pins are synchronous to the master clock signal. Therefore, additional constraints with values derived from the clock period constraint are added:

```
INST "example_cfg/example_frame_ecc" TPSYNC = FECC_SPECIAL ;
TIMESPEC "TS_FECC_SYNC" = FROM "FECC_SPECIAL" TO FFS(*) 7000 ps ;
TIMESPEC "TS_FECC_PADS" = FROM "FECC_SPECIAL" TO PADS(*) 20000 ps ;
```

Example Design Constraints

The example design constraints are organized by interface, rather than constraint type. The first group is for the master clock input to the entire design. It applies an I/O standard and a period constraint. The period constraint value is based on options set at generation:

```
NET "clk" IOSTANDARD = LVCMOS25 | PERIOD = 10000 ps;
```

The second group is for the Status Interface, applying an I/O standard and time name, so that an output timing constraint may be applied to the interface. The output timing constraint is set at two times the period constraint.

```
NET "status_initialization" IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW =
SLOW | TNM = STAPINS ;
NET "status_observation" IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = SLOW
| TNM = STAPINS ;
NET "status_correction" IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = SLOW
| TNM = STAPINS ;
NET "status_classification" IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW =
SLOW | TNM = STAPINS ;
NET "status_injection" IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = SLOW
| TNM = STAPINS ;
NET "status_uncorrectable" IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW =
SLOW | TNM = STAPINS ;
NET "status_essential" IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = SLOW
| TNM = STAPINS ;
NET "status_heartbeat" IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = SLOW
| TNM = STAPINS ;

TIMEGRP "STAPINS" OFFSET = OUT 20000 ps AFTER "clk" ;
```

The third group is for the MON shim, applying an I/O standard and time name, so that input and output timing constraints may be applied to the interface. The input and output timing constraints are set at two times the period constraint.

```
INST "example_mon/example_mon_sipo/sync_reg" IOB = TRUE ;
INST "example_mon/example_mon_piso/pipeline_serial" IOB = TRUE ;

NET "monitor_tx" IOSTANDARD = LVCMOS25 | DRIVE = 4 | SLEW = SLOW | TNM
= SERPINS ;
```

```
NET "monitor_rx" IOSTANDARD = LVCMOS25 | TNM = SERPINS ;

TIMEGRP "SERPINS" OFFSET = IN 20000 ps VALID 40000 ps BEFORE "clk" ;
TIMEGRP "SERPINS" OFFSET = OUT 20000 ps AFTER "clk" ;
```

The following group is for the EXT shim, and is only present when the EXT shim is generated based on options set at generation. It applies an I/O standard and time name, so that input and output timing constraints may be applied to the interface. The input and output timing is of considerable importance, as the actual timing must be used in the analysis of the SPI bus timing budget. However, there is no hard requirement for the input and output timing of the FPGA implementation. It will vary based on the selected device and speed grade.

As such, the input and output timing constraints are arbitrarily set at two times the period constraint. The additional constraint to use IOB flip flops will yield substantially better input and output timing than the constraint values suggest. It is the actual timing obtained from the timing report that should be used in the analysis of the SPI bus timing budget, not the constraint value.

```
INST "example_ext/example_ext_byte/ext_c_ofd" IOB = TRUE ;
INST "example_ext/example_ext_byte/ext_d_ofd" IOB = TRUE ;
INST "example_ext/example_ext_byte/ext_q_ifd" IOB = TRUE ;
INST "example_ext/ext_s_ofd" IOB = TRUE ;

NET "external_c" IOSTANDARD = LVCMOS25 | DRIVE = 8 | SLEW = FAST | TNM
= SPIPINS ;
NET "external_d" IOSTANDARD = LVCMOS25 | DRIVE = 8 | SLEW = FAST | TNM
= SPIPINS ;
NET "external_s_n" IOSTANDARD = LVCMOS25 | DRIVE = 8 | SLEW = FAST | TNM
= SPIPINS ;
NET "external_q" IOSTANDARD = LVCMOS25 | TNM = PADS:SPIPINS ;

TIMEGRP "SPIPINS" OFFSET = IN 20000 ps VALID 40000 ps BEFORE "clk" ;
TIMEGRP "SPIPINS" OFFSET = OUT 20000 ps AFTER "clk" ;
```

The following group is for the HID shim, and is only present when the HID shim is set to I/O Pins. It applies an I/O standard and time name, so that an input timing constraint may be applied to the interface. The input timing constraint is set at two times the period constraint.

```
NET "inject_strobe" IOSTANDARD = LVCMOS25 | TNM = PADS:INJPINS ;
NET "inject_address[0]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[1]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[2]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[3]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[4]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[5]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[6]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[7]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[8]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[9]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[10]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[11]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[12]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[13]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[14]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[15]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
```

```
NET "inject_address[16]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[17]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[18]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[19]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[20]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[21]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[22]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[23]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[24]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[25]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[26]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[27]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[28]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[29]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[30]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[31]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[32]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[33]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[34]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;
NET "inject_address[35]" IOSTANDARD = LVCMOS25 | TNM = INJPINS ;

TIMEGRP "INJPINS" OFFSET = IN 20000 ps VALID 40000 ps BEFORE "clk" ;
```

The following constraints in the UCF implement an area group to place portions of the system-level design example into a bounded region of the selected device. The instances included in the area group depend on the options set at generation. The range values vary depending on device selection. The area group is defined so that component packing is closed to resources from outside the group, but the unused component locations are open to placement of unrelated logic.

The area group forces packing of the soft error mitigation logic into an area physically adjacent to the ICAP site in the device. This has two benefits. The first and most important is to maintain reproducibility in timing results. The second is for improved resource usage; the area group forces tighter packing and generates a resource usage summary that is helpful in estimating the FIT of the system-level design example.

```
INST "example_wrapper/*" AREA_GROUP = "SEM_CONTROLLER" ;
INST "example_mon/*" AREA_GROUP = "SEM_CONTROLLER" ;
INST "example_ext/*" AREA_GROUP = "SEM_CONTROLLER" ;
INST "example_hid/*" AREA_GROUP = "SEM_CS_ICONVIO" ;

AREA_GROUP "SEM_CONTROLLER" RANGE = SLICE_X84Y115:SLICE_X99Y124 ;
AREA_GROUP "SEM_CONTROLLER" RANGE = RAMB18_X4Y46:RAMB18_X4Y49 ;
AREA_GROUP "SEM_CS_ICONVIO" RANGE = SLICE_X84Y100:SLICE_X99Y114 ;

AREA_GROUP "SEM_CONTROLLER" GROUP = CLOSED ;
AREA_GROUP "SEM_CS_ICONVIO" GROUP = CLOSED ;

AREA_GROUP "SEM_CONTROLLER" PLACE = OPEN ;
AREA_GROUP "SEM_CS_ICONVIO" PLACE = OPEN ;
```

The final constraints in the UCF are a template for assigning I/O pin locations to the top-level ports of the system-level example design. These assignments are necessarily board-specific and therefore cannot be automatically generated. To apply these constraints, uncomment them and fill in valid I/O pin locations for the target board.

```
## NET "clk" LOC = " ";
```

```
## NET "external_c" LOC = " " ;
## NET "external_d" LOC = " " ;
## NET "external_q" LOC = " ";
## NET "external_s_n" LOC = " ";

## NET "monitor_tx" LOC = " ";
## NET "monitor_rx" LOC = " ";

## NET "status_initialization" LOC = " ";
## NET "status_observation" LOC = " ";
## NET "status_correction" LOC = " ";
## NET "status_classification" LOC = " ";
## NET "status_injection" LOC = " ";
## NET "status_uncorrectable" LOC = " ";
## NET "status_essential" LOC = " ";
## NET "status_heartbeat" LOC = " ";

## NET "inject_strobe" LOC = " ";
## NET "inject_address[0]" LOC = " ";
## NET "inject_address[1]" LOC = " ";
## NET "inject_address[2]" LOC = " ";
## NET "inject_address[3]" LOC = " ";
## NET "inject_address[4]" LOC = " ";
## NET "inject_address[5]" LOC = " ";
## NET "inject_address[6]" LOC = " ";
## NET "inject_address[7]" LOC = " ";
## NET "inject_address[8]" LOC = " ";
## NET "inject_address[9]" LOC = " ";
## NET "inject_address[10]" LOC = " ";
## NET "inject_address[11]" LOC = " ";
## NET "inject_address[12]" LOC = " ";
## NET "inject_address[13]" LOC = " ";
## NET "inject_address[14]" LOC = " ";
## NET "inject_address[15]" LOC = " ";
## NET "inject_address[16]" LOC = " ";
## NET "inject_address[17]" LOC = " ";
## NET "inject_address[18]" LOC = " ";
## NET "inject_address[19]" LOC = " ";
## NET "inject_address[20]" LOC = " ";
## NET "inject_address[21]" LOC = " ";
## NET "inject_address[22]" LOC = " ";
## NET "inject_address[23]" LOC = " ";
## NET "inject_address[24]" LOC = " ";
## NET "inject_address[25]" LOC = " ";
## NET "inject_address[26]" LOC = " ";
## NET "inject_address[27]" LOC = " ";
## NET "inject_address[28]" LOC = " ";
## NET "inject_address[29]" LOC = " ";
## NET "inject_address[30]" LOC = " ";
## NET "inject_address[31]" LOC = " ";
## NET "inject_address[32]" LOC = " ";
## NET "inject_address[33]" LOC = " ";
## NET "inject_address[34]" LOC = " ";
## NET "inject_address[35]" LOC = " ";
```

XILINX®

Chapter 8

Applying the Solution

This chapter provides insight into the SEM Controller solution and how to apply it from three different levels, using a bottom-up approach. Interface Level, page 57 describes how to connect to the solution. Behavior Level, page 74 continues by describing how to interact with the solution through its interfaces once it has been connected. System Level, page 84 concludes by touching on aspects of integrating the solution into a larger system.

Interface Level

The system-level design example exposes four to six interfaces, depending on the options selected when it is generated. Each interface is described separately. The interface-level descriptions are intended to convey how to connect each interface.

Clock Interface

The following recommendations exist for the input clock. These recommendations are derived from the FPGA data sheet requirements for clock signals applied to the FPGA configuration system:

- Duty Cycle: 45% minimum, 55% maximum
- Jitter: Not Specified

The higher the frequency of the input clock, the lower the mitigation latency of the solution. Therefore, faster is better. There are several important factors that must be considered in determination of the maximum input clock frequency:

- Frequency must not exceed FPGA configuration system maximum clock frequency. Consult the device data sheet for the target device for this information.
- Frequency must not exceed the maximum clock frequency as reported in the static timing analyzer. This is generally not a limiting constraint.

Based on the fully synchronous design methodology, additional considerations arise in clock frequency selections that relate to the timing of external interfaces, if the system-level design example is used:

- For the EXT shim and memory interface:
  - The SPI bus timing budget must be evaluated to determine the maximum SPI bus clock frequency; a sample analysis is presented in External Interface, page 61.
  - The SPI bus clock is the input clock divided by two; therefore, the input clock cannot exceed twice the maximum SPI bus clock frequency.
- For the MON shim and serial interface:

- The input clock and the serial interface baud rate are related by an integer multiple of sixteen. For very high baud rates or very low input clock frequencies, the solution space may be limited if standard baud rates are desired.
- A sample analysis is presented in Monitor Interface, page 60.

After considering these factors, select an input clock frequency that satisfies all requirements.

Status Interface

Direct, logic-signal-based event reporting is available from the Status Interface. The Status Interface may be used for a variety of purposes, but its use is entirely optional. This interface reports three different types of information:

- State: Indicates what the controller is doing.
- Flags: Identifies the type of error detected.
- Heartbeat: Indicates the FPGA configuration system is active.

Only the heartbeat event is unique to the Status Interface. The other information is also available on the Monitor Interface.

In most cases, desired signals from the Status Interface should be brought to I/O pins on the FPGA. The system-level design example brings all of the signals to I/O pins. The Status Interface is the most reliable event reporting mechanism because it has the least amount of logic associated with it. Although it is possible to receive and interpret the status signals inside the FPGA, this is discouraged as it will reduce the overall reliability of the solution.

Externally, the status signals may be connected to indicators for viewing, or to another device for observation. In order to properly capture event reporting, the switching behavior of the Status Interface must be accounted for when interfacing to another device.

The signals in the Status Interface are generated by sequential logic processes in the controller using the clock supplied to the system-level design example. As a result, the pulse widths are always an integer number of clock cycles.

The collective switching behavior of the state signals `status_initialization`, `status_observation`, `status_correction`, `status_classification`, and `status_injection` is illustrated in Figure 8-1. In the figure, the `status_[state]` signal represents the five state signals, as a group, which may be considered an indication of the controller state.

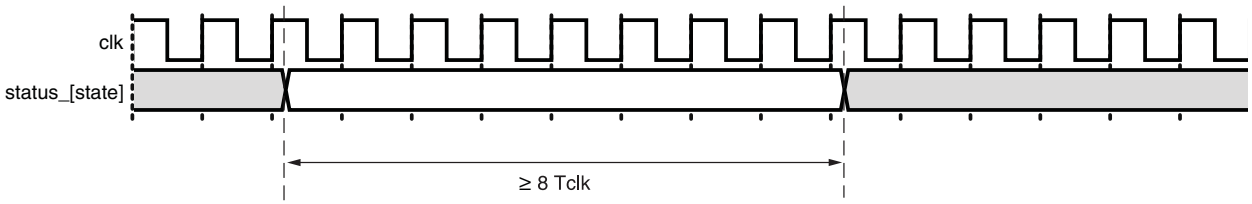

Figure 8-1: **Status Interface State Signals Switching Characteristics**

The switching behavior of the flag signals `status_uncorrectable` and `status_essential` is relative to the exit from the states where these flags are updated, as illustrated in Figure 8-2 and Figure 8-3. The figures illustrate a window of time when the

flags are valid with respect to transitions out of the state in which they may be updated. Specific flag values are not shown in the waveform.

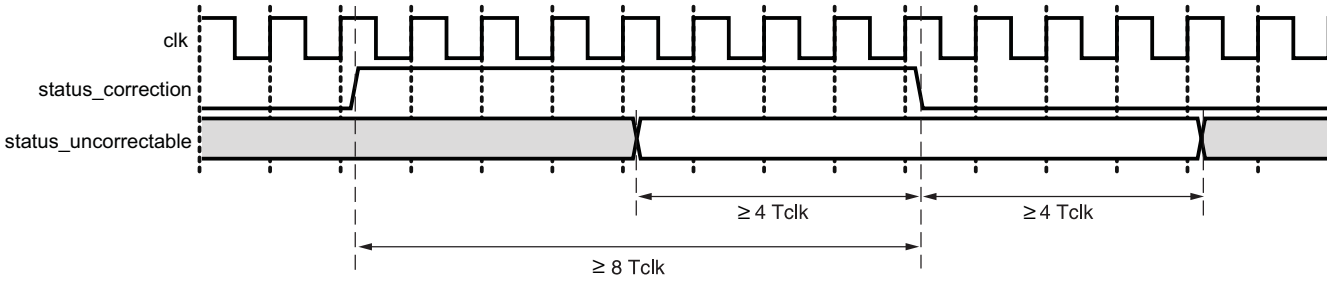

Figure 8-2: **Status Interface Uncorrectable Flag Switching Characteristics**

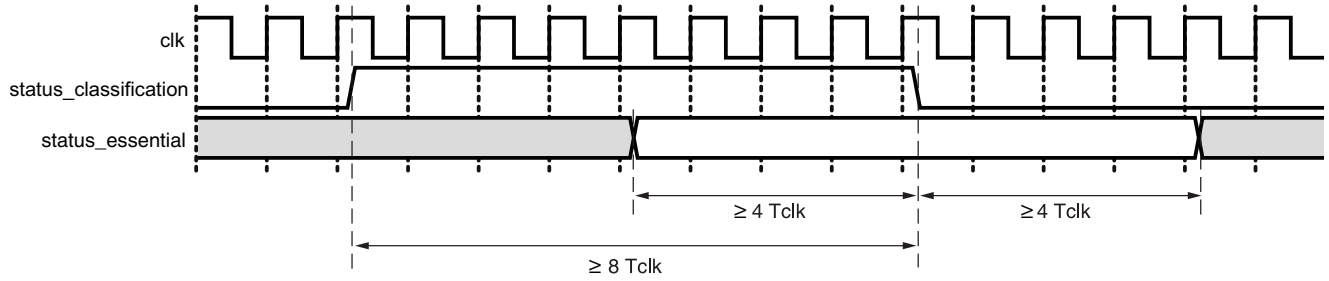

Figure 8-3: **Status Interface Essential Flag Switching Characteristics**

The switching behavior of the heartbeat signal `status_heartbeat` is illustrated in Figure 8-4. This signal is a direct output from the integrated silicon readback process, and is active during the observation state. Upon entering the observation state, the heartbeat signal will become active once the integrated silicon readback process is scanning for errors. The first heartbeat pulse observed during the observation state must be used to arm any circuit that monitors for loss of heartbeat. In other states, the heartbeat signal will be active, but the switching behavior is not guaranteed.

The first heartbeat after entering the observation state should occur within the one readback scan. An entire scan will depend on the device and frequency. See Table 8-7 to determine how long a readback scan will take for the device being used.

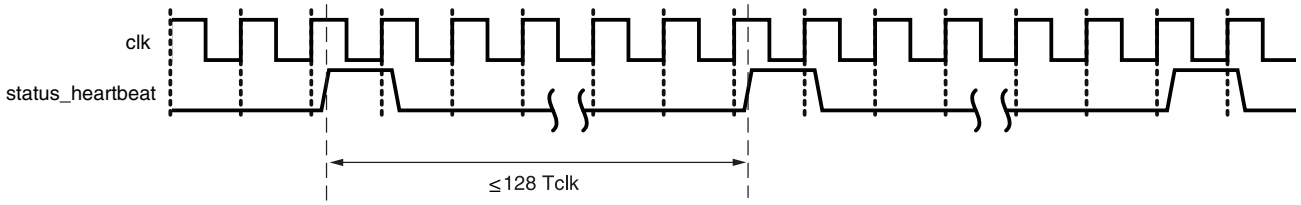

Figure 8-4: **Status Interface Heartbeat Switching Characteristics**

Due to the small pulse widths involved, approaches such as sampling the Status Interface signals through embedded processor GPIO using software polling are not likely to work. Instead, use other approaches such as using counter/timer inputs, edge sensitive interrupt inputs, or inputs with event capture capability.

Monitor Interface

The Monitor Interface consists of two signals implementing an RS-232 protocol compatible, full duplex serial port for exchange of commands and status. The following configuration is used:

- Baud: 9600
- Settings: 8-N-1
- Flow Control: None
- Terminal Setup: VT100
  - TX Newline: CR (Terminal transmits CR [0x0D] as end of line)
  - RX Newline: CR+LF (Terminal receives CR [0x0D] as end of line and expands to CR+LF [0x0D, 0x0A])
  - Local Echo: NO

Any external device connected to the Monitor Interface must support this configuration. Figure 8-5 shows the switching behavior, and is representative of both transmit and receive.

Figure 8-5: **Monitor Interface Switching Characteristics**

Transmit and receive timing is derived from a 16x bitrate enable signal which is created inside the system-level example design using a counter. The behavior of this counter is to start counting from zero, up to and including a terminal count (a condition detected and used to synchronously reset the counter). The terminal count output is also supplied to transmit and receive processes as a time base.

From a compatibility perspective, the advantages of 9600 baud are that it is a standard bitrate, and it is also realizable with a broad range of input clock frequencies. It is used in the system-level design example for these reasons.

From a practical perspective, the disadvantage of such a low bitrate is communication performance (both data rate and latency). This performance can throttle the controller. For this reason, changing to a higher bitrate is strongly encouraged. A wide variety of other bitrates are possible, including standard bitrates: 115200, 230400, 460800, and 921600 baud.

In the system-level example design module for the MON shim, the parameter V_ENABLETIME sets the communication bitrate. The value for V_ENABLETIME is calculated using:

$$\text{V\_ENABLETIME= round to integer}\left[\frac{input\ clock\ frequency}{16 \times nominal\ \ bitrate}\right] - 1 \qquad \textit{Equation 8-1}$$

A rounding error as great as +/- 0.5 may result from the computation of V_ENABLETIME. This error produces a bitrate that is slightly different than the nominal bitrate. A difference of 2% between RS-232 devices is considered acceptable, which suggests a bitrate tolerance of +/- 1% for each device.

Example: The input clock is 66 MHz, and the desired bitrate is 115200 baud.

$$\text{V\_ENABLETIME= round to integer}\left[\frac{66000000}{16 \times 115200}\right] - 1 = 35 \qquad \textit{Equation 8-2}$$

The actual bitrate that results is approximately 114583 baud, which deviates -0.54% from the nominal bitrate of 115200 baud. This is acceptable since the difference is within +/- 1%.

When exploring bitrates, if the difference from nominal exceeds the +/- 1% tolerance, select another combination of bitrate and input clock frequency that yields less error. No additional switching characteristics are specified.

Electrically, the I/O pins used by the Monitor Interface support LVCMOS signaling, which is suitable for interfacing with other devices. No specific I/O mode is required. When full electrical compatibility with RS-232 is desired, then a suitable external level translator must be used.

External Interface

The External Interface consists of four signals implementing a SPI bus protocol compatible, full duplex serial port. This interface is only present when one or both of the following controller options are enabled:

- Error Correction by Replacement
- Error Classification

The implementations of these functions require external storage. The system-level design example provides a fixed-function SPI bus master in the EXT shim to fetch data from a single external SPI flash device. Table 8-1 provides an SPI flash device recommendation for each supported FPGA:

Table 8-1: **External Storage Requirements**

FPGA Device	Error Classification Only	Error Correction by Replacement Only	Error Classification with Error Correction by Replacement
XC7K70T	32 Mbit	32 Mbit	64 Mbit
XC7K160T	64 Mbit	64Mbit	128 Mbit
XC7K325T	128 Mbit	128 Mbit	256 Mbit
XC7K355T	128 Mbit	128 Mbit	256 Mbit
XC7K410T	128 Mbit	128 Mbit	256 Mbit
XC7K420T	128 Mbit	128 Mbit	256 Mbit
XC7K480T	128 Mbit	128 Mbit	256 Mbit
XC7VX330T	128 Mbit	128 Mbit	256 Mbit
XC7VX415T	128 Mbit	128 Mbit	256 Mbit
XC7VX485T	128 Mbit	128 Mbit	256 Mbit
XC7VX550T	256 Mbit	256 Mbit	TBD
XC7V585T	128 Mbit	128 Mbit	256 Mbit
XC7VX690T	256 Mbit	256 Mbit	TBD
XC7VX980T	256 Mbit	256 Mbit	TBD
XC6VCX75T	32 Mbit	32 Mbit	64 Mbit
XC6VCX130T	32 Mbit	32 Mbit	64 Mbit
XC6VCX195T	64 Mbit	64 Mbit	128 Mbit
XC6VCX240T	64 Mbit	64 Mbit	128 Mbit

Table 8-1: **External Storage Requirements *(Cont'd)***

FPGA Device	Error Classification Only	Error Correction by Replacement Only	Error Classification with Error Correction by Replacement
XC6VHX250T	64 Mbit	64 Mbit	128 Mbit
XC6VHX255T	64 Mbit	64 Mbit	128 Mbit
XC6VHX380T	128 Mbit	128 Mbit	256 Mbit
XC6VHX565T	128 Mbit	128 Mbit	256 Mbit
XC6VLX75T	32 Mbit	32 Mbit	64 Mbit
XC6VLX130T	32 Mbit	32 Mbit	64 Mbit
XC6VLX195T	64 Mbit	64 Mbit	128 Mbit
XC6VLX240T	64 Mbit	64 Mbit	128 Mbit
XC6VLX365T	128 Mbit	128 Mbit	256 Mbit
XC6VLX550T	128 Mbit	128 Mbit	256 Mbit
XC6VLX760	256 Mbit	256 Mbit	512 Mbit
XC6VSX315T	128 Mbit	128 Mbit	256 Mbit
XC6VSX475T	128 Mbit	128 Mbit	256 Mbit
XC6SLX4	4 Mbit	N/A	N/A
XC6SLX9	4 Mbit	N/A	N/A
XC6SLX16	4 Mbit	N/A	N/A
XC6SLX25T	8 Mbit	N/A	N/A
XC6SLX45T	16 Mbit	N/A	N/A
XC6SLX75T	16 Mbit	N/A	N/A
XC6SLX100T	32 Mbit	N/A	N/A
XC6SLX150T	32 Mbit	N/A	N/A

The SPI flash device must support the standard "fast read" command 0x0B. If the SPI flash density is larger than 128 Mbit, it must also support a 32-bit addressing extension, enabled with "enable 32-bit addressing" command 0xB7. Creation of a suitable programming file is described in Chapter 6, Building the Solution.

Figure 8-6 shows the connectivity between an FPGA device and a SPI Flash device. Note the presence of level translators (marked "LT"). These are required because commonly available SPI Flash devices use 3.3V I/O, which may not be available depending on the

selected FPGA device or I/O bank voltage. See Table 4-3, page 34 for a description of the signals shown in Figure 8-6.

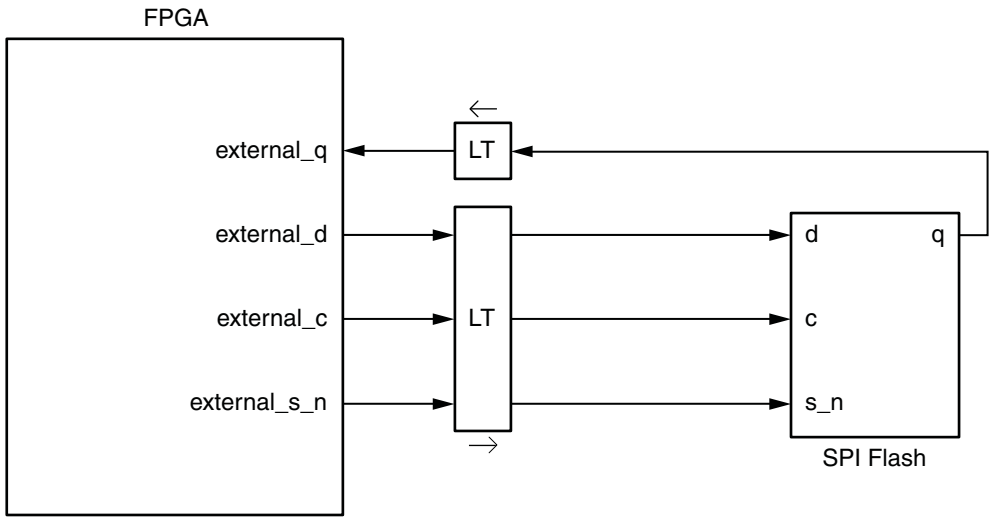

Figure 8-6: **SPI Flash Device Connection, Including Level Translators**

The level translators must exhibit low propagation delay to maximize the SPI bus performance. The SPI bus performance can potentially affect the maximum frequency of operation of the entire system-level design example.

The following sections illustrate how to analyze the SPI bus timing budgets. This is a critical analysis which must be performed to ensure reliable data transfer over the SPI bus. Every implementation should be considered unique and be carefully evaluated to ensure the timing budgets posed in the example are satisfied.

SPI Bus Clock Waveform and Timing Budget

The SPI flash device will have requirements on the switching characteristics of its input clock. This analysis is for the clock signal generated for the SPI flash device by the system-level design example. Completion of this analysis requires board-level signal integrity simulation capability.

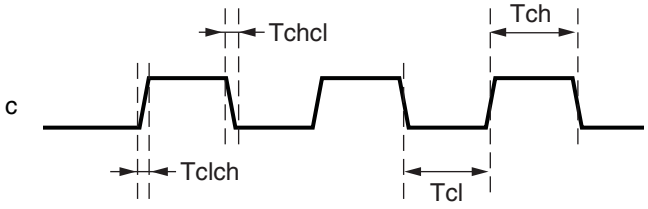

Figure 8-7: **SPI Flash Device Input Clock Requirements**

The following parameters, shown in Figure 8-7, are defined as requirements on the clock input to the SPI flash device:

- T_{clch} = SPI bus clock maximum rise time requirement
- T_{chcl} = SPI bus clock maximum fall time requirement
- T_{cl} = SPI bus clock minimum low time requirement
- T_{ch} = SPI bus clock minimum high time requirement

Based on the physical construction of the SPI bus, the I/O characteristics of the FPGA, and the I/O characteristics of any level translator used, the SPI bus clock signal originating at the FPGA will exhibit maximum rise and fall times (T_{rise} and T_{fall}) at the SPI flash device. Satisfaction of T_{clch} and T_{chcl} requirements by T_{rise} and T_{fall} must be verified. Should T_{clch} and T_{chcl} requirements not be satisfied, avenues of correction include:

- Change I/O slew rate for the system-level design example SPI bus clock output.
- Change I/O drive strength for the system-level design example SPI bus clock output.
- Select an alternate level translator with more suitable I/O characteristics.

Generally, the T_{clch} and T_{chcl} requirements are easy to satisfy. They exist to prohibit exceptionally long rise and fall times that might occur on a true bus with many loads, rather than the point-to-point scheme used with the system-level design example.

The SPI bus clock generated by the system-level design example is the input clock divided by two. Therefore, the SPI bus clock high and low times are nominally equal to T_{clk}. However, considering actual T_{rise} and T_{fall}, also ensure satisfaction of the following:

- $T_{clk} \geq T_{rise} + T_{ch}$
- $T_{clk} \geq T_{fall} + T_{cl}$

Example:

- T_{clch} = 33 ns (from SPI flash datasheet)
- T_{chcl} = 33 ns (from SPI flash datasheet)
- T_{cl} = 9 ns (from SPI flash datasheet)
- T_{ch} = 9 ns (from SPI flash datasheet)
- T_{rise} = 2 ns (from PCB simulation)
- T_{fall} = 2 ns (from PCB simulation)

Given this data, perform the following:

1. Check: Is $T_{clch} \geq T_{rise}$? Is 33 ns ≥ 2 ns? Yes
2. Check: Is $T_{chcl} \geq T_{fall}$? Is 33 ns ≥ 2 ns? Yes
3. Calculate: $T_{clk} \geq T_{rise} + T_{ch}$ requires $T_{clk} \geq 2$ ns + 9 ns, or $T_{clk} \geq 11$ ns
4. Calculate: $T_{clk} \geq T_{fall} + Tcl$ requires $T_{clk} \geq 2$ ns + 9 ns, or $T_{clk} \geq 11$ ns

The rise time requirements are satisfied. These requirements on T_{clk} indicate that the SPI Bus Clock Waveform and Timing Budget will restrict the system-level design example input clock cycle time to be 11 ns or larger.

SPI Bus Transmit Waveform and Timing Budget

The SPI flash device will have requirements on the switching characteristics of its input data with respect to its input clock. This analysis is for data capture at the SPI flash device, when receiving data from the system-level design example.

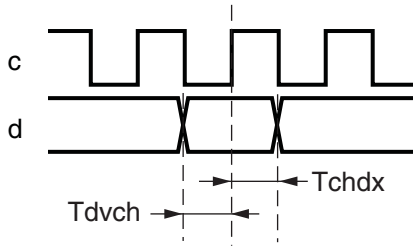

Figure 8-8: **SPI Flash Device Input Data Capture Requirements**

The following parameters, shown in Figure 8-8, are defined as requirements for successful data capture by the SPI flash device:

- T_{dvch} = SPI flash minimum data setup requirement with respect to clock
- T_{chdx} = SPI flash minimum data hold requirement with respect to clock

The analysis assumes minimum propagation delays are zero. This analysis also assumes the following skews are negligible:

- Skew on input clock distribution to FPGA output flip flops.
- Skew on output signal paths from FPGA output flip flops to FPGA pins.
- Skew in PCB level translator channel delays. The level translator on clock and data paths must be matched for this to be true.
- Skew in PCB trace segment delays. The trace delay on clock and data paths must be matched for this to be true
- Duty cycle distortion.

The following parameters are defined as implementation parameters of the EXT shim and PCB:

- T_{clk} = input clock cycle time (`icap_clk`)
- T_{qfpga} = FPGA output delay with respect to `icap_clk`
- T_{w1} = FPGA to level translator PCB trace delay
- T_{w2} = Level translator to SPI flash PCB trace delay
- T_{dly} = Level translator channel delay

The memory system signaling generated by the EXT shim implementation is shown in Figure 8-9.

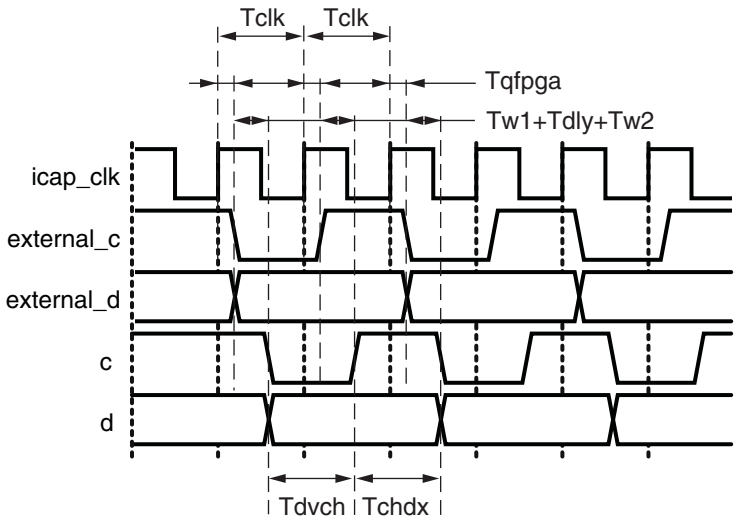

Figure 8-9: **Input Data Capture Timing**

Given the stated assumptions, the delays on both the clock and data paths are identical and track each other over process, voltage, and temperature variations. The following relationships exist:

- $T_{clk} \geq T_{dvch}$
- $T_{clk} \geq T_{chdx}$

Example:

- T_{dvch} = 2 ns (from SPI flash datasheet)
- T_{chdx} = 5 ns (from SPI flash datasheet)

1. Calculate: $T_{clk} \geq T_{dvch}$ requires $T_{clk} \geq 2$ ns
2. Calculate: $T_{clk} \geq T_{chdx}$ requires $T_{clk} \geq 5$ ns

These requirements on T_{clk} indicate that the SPI Transmit Waveform and Timing Budget will restrict the system-level design example input clock cycle time to be 5 ns or larger.

SPI Bus Receive Waveform and Timing Budget

The SPI flash device will exhibit certain output switching characteristics of its output data with respect to its input clock. This analysis is for data capture at the system-level design example, when receiving data from the SPI flash device.

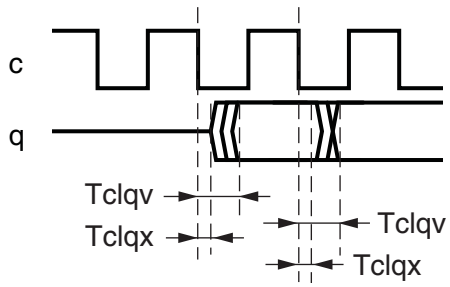

Figure 8-10: **SPI Flash Device Output Data Switching Characteristics**

The following parameters, shown in Figure 8-10, are defined as the output switching behavior of the SPI flash device:

- T_{clqv} = SPI flash maximum output valid with respect to clock
- T_{clqx} = SPI flash minimum output hold with respect to clock

The analysis assumes minimum propagation delays are zero. This analysis also assumes the following skews are negligible:

- Skew on input clock distribution to FPGA output and input flip flops.
- Skew in PCB level translator channel delays. The level translator on clock and data paths must be matched for this to be true.
- Duty cycle distortion.

The following parameters are defined as implementation parameters of the EXT shim and PCB:

- T_{clk} = input clock cycle time (`icap_clk`)
- T_{qfpga} = FPGA output delay with respect to `icap_clk`
- T_{sfpga} = FPGA input setup requirement with respect to `icap_clk`
- T_{hfpga} = FPGA input hold requirement with respect to `icap_clk`
- T_{w1} = FPGA to level translator PCB trace delay
- T_{w2} = Level translator to SPI flash PCB trace delay
- T_{w3} = SPI flash to level translator PCB trace delay
- T_{w4} = Level translator to FPGA PCB trace delay
- T_{dly} = Level translator channel delay

The timing path is a two cycle path for the EXT shim, but a single cycle path to the SPI flash device. For the timing analysis, the clock to out of the SPI flash device is modeled as a combinational delay. Both setup and hold requirements at the FPGA must be considered.

The memory system signaling generated by the EXT shim implementation is shown in Figure 8-11 and Figure 8-12.

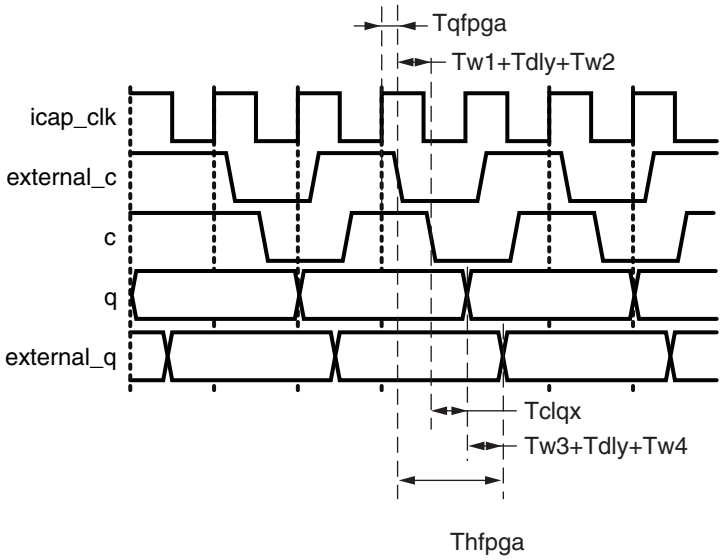

Figure 8-11: **Output Data Capture Timing (Hold Analysis)**

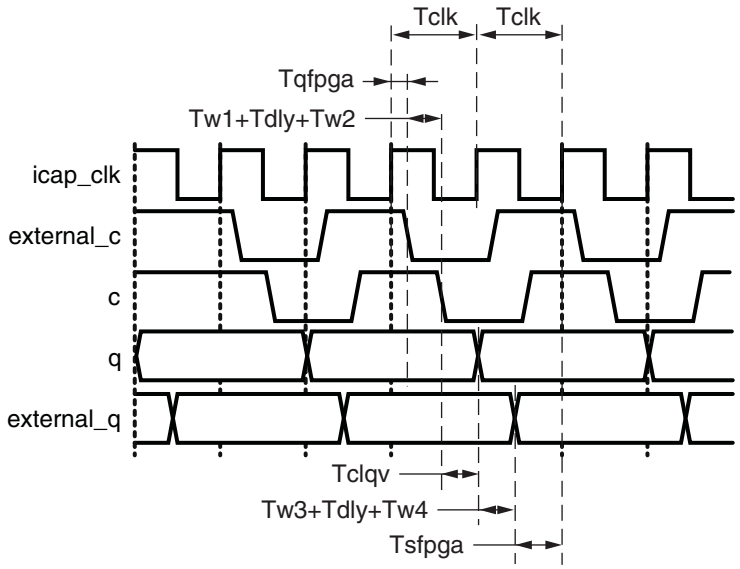

Figure 8-12: **Output Data Capture Timing (Setup Analysis)**

The hold path analysis is a pass/fail test. The hold path analysis must be calculated using minimum delay values, for which the following relationship must be verified:

$$T_{hfpga} \leq T_{qfpga,min} + T_{w1} + T_{dly} + T_{w2} + T_{clqx} + T_{w3} + T_{dly} + T_{w4}$$

Substituting zero as a conservative minimum delay for T_{w1}, T_{w2}, T_{w3}, T_{w4}, and T_{dly} yields:

$$T_{hfpga} \leq T_{qfpga,min} + T_{clqx}$$

The setup path analysis must be calculated using maximum delay values:

$$T_{clk} \geq 0.5*(T_{qfpga,max} + T_{w1} + T_{dly} + T_{w2} + T_{clqv} + T_{w3} + T_{dly} + T_{w4} + T_{sfpga})$$

Example:

- T_{clqv} = 8 ns (from SPI flash datasheet)
- T_{clqx} = 0 ns (from SPI flash datasheet)
- T_{dly} = 3 ns (from level translator datasheet)
- T_{w1} = 1 ns (from board simulation)
- T_{w2} = 1 ns (from board simulation)
- T_{w3} = 1 ns (from board simulation)
- T_{w4} = 1 ns (from board simulation)

The FPGA timing parameters must be obtained from the timing report that results from the implementation of the system-level design example in the FPGA device targeted for use in the application. In order to generate the necessary reports, the timing analyzer must be run using the "-fastpaths" option.

The examples that follow are excerpts from the timing analyzer report generated from a Virtex-6 device implementation of the system-level design example. The purpose of the examples are to illustrate where to find the required information.

Locate T_{qfpga} by searching the timing report for flip flop to pad maximum path timing analysis, where the destination pad is identified as “external_c”.

- T_{qfpga} = I/O Data Path Delay (external_c)
- T_{qfpga} = 3.360 ns, maximum

```
--------------------------------------------------------------------------------
Slack: 13.042ns (requirement - (clock arrival + clock path + data path + uncertainty))
 Source:    example_ext/example_ext_byte/ext_c_ofd (FF)
 Destination:   external_c (PAD)
 Source Clock:   icap_clk rising at 0.000ns
 Requirement:   20.000ns
 Data Path Delay:  3.360ns (Levels of Logic = 1)
 Clock Path Delay:  3.573ns (Levels of Logic = 2)
 Clock Uncertainty: 0.025ns

 Clock Uncertainty:   0.025ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
 Total System Jitter (TSJ): 0.050ns
 Total Input Jitter (TIJ): 0.000ns
 Discrete Jitter (DJ):  0.000ns
 Phase Error (PE):   0.000ns

 Maximum Clock Path at Slow Process Corner:
 clk to example_ext/example_ext_byte/ext_c_ofd
 Location    Delay type   Delay(ns) Physical Resource
             Logical Resource(s)
 ------------------------------------------------- -------------------
 U23.I    Tiopi     0.702 clk clk_IBUFG
 BUFGCTRL_X0Y0.I0  net (fanout=1)  0.938 clk_IBUFG
 BUFGCTRL_X0Y0.O  Tbgcko_O    0.092 example_bufg example_bufg
 OLOGIC_X0Y194.CLK net (fanout=283)  1.841 icap_clk
 ------------------------------------------------- ---------------------------
 Total          3.573ns (0.794ns logic, 2.779ns route)
             (22.2% logic, 77.8% route)

 Maximum Data Path at Slow Process Corner:
 example_ext/example_ext_byte/ext_c_ofd to external_c
 Location    Delay type   Delay(ns) Physical Resource
             Logical Resource(s)
 ------------------------------------------------- -------------------
 OLOGIC_X0Y194.OQ  Tockq     0.625 external_c_OBUF example_ext/example_ext_byte/ext_c_ofd
 E33.O    net (fanout=1)  0.002 external_c_OBUF
 E33.PAD    Tioop     2.733 external_c external_c_OBUF external_c
 ------------------------------------------------- ---------------------------
 Total          3.360ns (3.358ns logic, 0.002ns route)
             (99.9% logic, 0.1% route)
--------------------------------------------------------------------------------
```

Locate T_{qfpga} by searching the timing report for flip flop to pad minimum path timing analysis, where the destination pad is identified as "external_c".

- T_{qfpga} = I/O Data Path Delay (external_c)
- T_{qfpga} = 1.473 ns, minimum

```
--------------------------------------------------------------------------------
Delay (fastest paths): 2.951ns (clock arrival + clock path + data path - uncertainty)
 Source:    example_ext/example_ext_byte/ext_c_ofd (FF)
 Destination:   external_c (PAD)
 Source Clock:   icap_clk rising at 0.000ns
 Data Path Delay:  1.473ns (Levels of Logic = 1)
 Clock Path Delay:  1.503ns (Levels of Logic = 2)
 Clock Uncertainty: 0.025ns

 Clock Uncertainty:   0.025ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
 Total System Jitter (TSJ): 0.050ns
 Total Input Jitter (TIJ): 0.000ns
 Discrete Jitter (DJ):  0.000ns
 Phase Error (PE):   0.000ns

 Minimum Clock Path at Fast Process Corner: clk to example_ext/example_ext_byte/ext_c_ofd
 Location    Delay type   Delay(ns) Physical Resource
             Logical Resource(s)
 ------------------------------------------------- -------------------
 U23.I    Tiopi     0.316 clk
             clk
             clk_IBUFG
 BUFGCTRL_X0Y0.I0  net (fanout=1)  0.367 clk_IBUFG
 BUFGCTRL_X0Y0.O  Tbgcko_O    0.033 example_bufg
             example_bufg
 OLOGIC_X0Y194.CLK net (fanout=283)  0.787 icap_clk
 ------------------------------------------------- ---------------------------
 Total          1.503ns (0.349ns logic, 1.154ns route)
             (23.2% logic, 76.8% route)

 Minimum Data Path at Fast Process Corner: example_ext/example_ext_byte/ext_c_ofd to
external_c
 Location    Delay type   Delay(ns) Physical Resource
             Logical Resource(s)
 ------------------------------------------------- -------------------
 OLOGIC_X0Y194.OQ  Tockq     0.215 external_c_OBUF
             example_ext/example_ext_byte/ext_c_ofd
 E33.O    net (fanout=1)  0.002 external_c_OBUF
 E33.PAD    Tioop     1.256 external_c
             external_c_OBUF
             external_c
 ------------------------------------------------- ---------------------------
 Total          1.473ns (1.471ns logic, 0.002ns route)
             (99.9% logic, 0.1% route)
--------------------------------------------------------------------------------
```

Locate T_{sfpga} by searching the timing report for pad to flip flop maximum path timing analysis, where the source pad is identified as “external_q”.

- T_{sfpga} = I/O Data Path Delay (external_q)
- T_{sfpga} = 4.943 ns, maximum

```
--------------------------------------------------------------------------------
Slack (setup): 18.398ns (requirement - (data path - clock path - clock arrival +
uncertainty))
 Source:    external_q (PAD)
 Destination:   example_ext/example_ext_byte/ext_q_ifd (FF)
 Destination Clock: icap_clk rising at 0.000ns
 Requirement:   20.000ns
 Data Path Delay:  4.943ns (Levels of Logic = 1)
 Clock Path Delay:  3.366ns (Levels of Logic = 2)
 Clock Uncertainty: 0.025ns

 Clock Uncertainty:   0.025ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
 Total System Jitter (TSJ): 0.050ns
 Total Input Jitter (TIJ): 0.000ns
 Discrete Jitter (DJ):  0.000ns
 Phase Error (PE):   0.000ns

 Maximum Data Path at Slow Process Corner:
 external_q to example_ext/example_ext_byte/ext_q_ifd
 Location    Delay type   Delay(ns) Physical Resource
             Logical Resource(s)
 ------------------------------------------------- -------------------
 C33.I    Tiopi     0.766 external_q external_q external_q_IBUF
 ILOGIC_X0Y187.DDLY net (fanout=1)  4.046 external_q_IBUF
 ILOGIC_X0Y187.CLK Tidockd    0.131 fetch_rxdata<0>
             example_ext/example_ext_byte/ext_q_ifd
 ------------------------------------------------- ---------------------------
 Total          4.943ns (0.897ns logic, 4.046ns route)
             (18.1% logic, 81.9% route)

 Minimum Clock Path at Slow Process Corner:
 clk to example_ext/example_ext_byte/ext_q_ifd
 Location    Delay type   Delay(ns) Physical Resource
             Logical Resource(s)
 ------------------------------------------------- -------------------
 U23.I    Tiopi     0.670 clk
             clk
             clk_IBUFG
 BUFGCTRL_X0Y0.I0  net (fanout=1)  0.865 clk_IBUFG
 BUFGCTRL_X0Y0.O  Tbgcko_O    0.087 example_bufg example_bufg
 ILOGIC_X0Y187.CLK net (fanout=283)  1.744 icap_clk
 ------------------------------------------------- ---------------------------
 Total          3.366ns (0.757ns logic, 2.609ns route)
             (22.5% logic, 77.5% route)
--------------------------------------------------------------------------------
```

Locate T_{hfpga} by searching the timing report for pad to flip flop minimum path timing analysis, where the source pad is identified as "external_q".

- T_{hfpga} = I/O Data Path Delay (external_q)
- T_{hfpga} = -1.800 ns, minimum

```
--------------------------------------------------------------------------------
Slack (hold): 0.131ns (requirement - (clock path + clock arrival + uncertainty - data path))
 Source:    external_q (PAD)
 Destination:   example_ext/example_ext_byte/ext_q_ifd (FF)
 Destination Clock: icap_clk rising at 0.000ns
 Requirement:   0.000ns
 Data Path Delay:  1.800ns (Levels of Logic = 1)
 Clock Path Delay:  1.644ns (Levels of Logic = 2)
 Clock Uncertainty: 0.025ns

 Clock Uncertainty:   0.025ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
 Total System Jitter (TSJ): 0.050ns
 Total Input Jitter (TIJ): 0.000ns
 Discrete Jitter (DJ):  0.000ns
 Phase Error (PE):   0.000ns

 Minimum Data Path at Fast Process Corner:
 external_q to example_ext/example_ext_byte/ext_q_ifd
 Location    Delay type   Delay(ns) Physical Resource
             Logical Resource(s)
 ------------------------------------------------- -------------------
 C33.I    Tiopi     0.371 external_q external_q external_q_IBUF
 ILOGIC_X0Y187.DDLY net (fanout=1)  1.432 external_q_IBUF
 ILOGIC_X0Y187.CLK Tiockdd  (-Th)  0.003 fetch_rxdata<0>
             example_ext/example_ext_byte/ext_q_ifd
 ------------------------------------------------- ---------------------------
 Total          1.800ns (0.368ns logic, 1.432ns route)
             (20.4% logic, 79.6% route)

 Maximum Clock Path at Fast Process Corner:
 clk to example_ext/example_ext_byte/ext_q_ifd
 Location    Delay type   Delay(ns) Physical Resource
             Logical Resource(s)
 ------------------------------------------------- -------------------
 U23.I    Tiopi     0.371 clk clk clk_IBUFG
 BUFGCTRL_X0Y0.I0  net (fanout=1)  0.397 clk_IBUFG
 BUFGCTRL_X0Y0.O  Tbgcko_O    0.035 example_bufg example_bufg
 ILOGIC_X0Y187.CLK net (fanout=283)  0.841 icap_clk
 ------------------------------------------------- ---------------------------
 Total          1.644ns (0.406ns logic, 1.238ns route)
             (24.7% logic, 75.3% route)
--------------------------------------------------------------------------------
```

Check:

- Is $T_{hfpga} \leq T_{qfpga,min} + T_{clqx}$?
- Is -1.800 ns ≤ 1.473 ns + 0 ns?
- Is -1.800 ns ≤ 1.473 ns? YES

Calculate:

$$T_{clk} \geq 0.5*(T_{qfpga,max} + T_{w1} + T_{dly} + T_{w2} + T_{clqv} + T_{w3} + T_{dly} + T_{w4} + T_{sfpga})$$

requires

$$T_{clk} \geq 0.5*(3.360 \text{ ns} + 1 \text{ ns} + 3 \text{ ns} + 1 \text{ ns} + 8 \text{ ns} + 1 \text{ ns} + 3 \text{ ns} + 1 \text{ ns} + 4.943 \text{ ns})$$

or

$$T_{clk} \geq 13.152 \text{ ns}$$

The hold requirement is satisfied, and the requirement on T_{clk} indicates that the SPI Receive Waveform and Timing Budget will restrict the system-level design example input clock cycle time to be 13.152 ns or larger.

SPI Bus Timing Budget Conclusions

When the EXT shim and external memory system are present, the SPI bus timing budget must be analyzed to ensure a robust implementation. The result of the analysis will confirm that the external memory system is functional, and reveal any constraints it may pose on the maximum frequency of the system-level design example input clock.

Example:

Using the results from the preceding examples in this section, the memory interface is functional, and the most stringent requirement on T_{clk} is that $T_{clk} \geq 13.152$ ns, as the memory interface will only work when the input clock frequency is 76.037 MHz or lower. Other input clock frequency limits, such as the ICAP maximum clock frequency and the system-level example maximum clock frequency, must also be considered.

Error Injection Interface

The Error Injection Interface consists of an input bus and input strobe, implementing a simple parallel input port. This interface is only present when Error Injection is enabled and I/O Pins are selected. Direct, logic-signal-based error injection is possible from the Error Injection Interface. The use of this interface is entirely optional. This interface accepts four types of commands:

- Command to enter Idle state (halt normal scanning to inject errors)
- Command to enter Observation state (resume normal scanning)
- Inject error at physical frame address
- Inject error at linear frame address

These commands are not unique to the Error Injection Interface; they are also available on the Monitor Interface.

In many cases, desired signals from the Error Injection Interface may be brought to I/O pins on the FPGA. In one configuration, the system-level design example brings all of the signals to I/O pins. It is possible to generate and drive the error injection signals inside the FPGA; in fact, this is exactly what is done when ChipScope is selected rather than I/O pins.

Externally, the error injection signals may be connected to another device for control. In order to properly capture supplied commands, the timing requirements of the Error Injection Interface must be accounted for when interfacing to another device.

The signals in the Error Injection Interface are received by a sequential logic process in the controller using the strobe to enable an input register. The timing requirements shown in Figure 8-13 must be observed to ensure successful data capture.

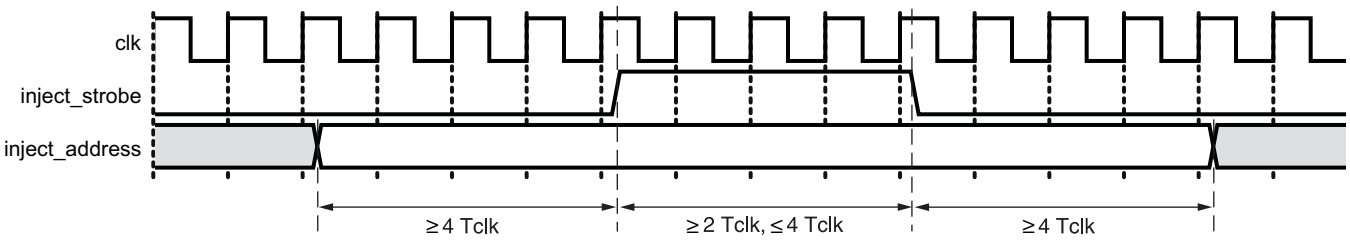

Figure 8-13: **Error Injection Interface Timing Requirements**

The clock signal shown in Figure 8-13 is meant only to illustrate the waveform timescale. While the pulse widths are specified in terms of clock cycles, the external device generating and driving the error injection signals need not be synchronous to the clock signal.

If an error is injected into a frame that is masked or beyond the supported address range for a device or SEU coverage, the error injection command will be ignored and no error will be detected in the observation state.

Behavior Level

The system-level design example exhibits certain high-level functional behaviors during operation, based on the controller design. This section is intended to convey expected behaviors and how to interact with the system-level design example.

Controller Activity

Initialization

During FPGA configuration, the controller is held in reset by the FPGA global set/reset signal. At the completion of configuration, the FPGA configuration system de-asserts the global set/reset signal and the controller initializes. During initialization state, `status_initialization` is true.

This initialization includes some internal housekeeping, as well as directly observable events such as the generation of a status report on the Monitor Interface. Initialization includes:

- A delay to wait for availability of the configuration system through ICAP
- A first readback cycle where the frame-level ECC checksums are computed
- A second readback cycle where the device-level CRC checksum is computed
- An additional readback cycle where an additional checksum is computed (only executed if Virtex-6 or 7 series devices are targeted)
- An additional readback cycle where the frame-level CRC checksums are computed (only executed if correction by enhanced repair is used)

At the completion of initialization, the controller transitions to the observation state.

Observation

The controller spends virtually all of its time in the observation state. During the observation state, `status_observation` is true and the controller observes the FPGA configuration system for indication of error conditions. If no error exists, and the controller receives a command (from either the Error Injection Interface or the Monitor Interface), then the controller will process the received command. Only two commands are supported in the observation state, the "enter idle" and "status report" commands. The controller will ignore all other commands.

The "enter idle" command may be applied through either the Error Injection Interface or the Monitor Interface, and is used to idle the controller so that error injections may be performed. This command causes the controller to transition to the idle state.

The "status report" command is not frequently used; it provides some diagnostic information, and may be helpful as a mechanism to "ping" the controller without idling it. This command is only supported on the Monitor Interface.

In the event an error is detected, the controller reads additional information from the hardware in preparation for a correction attempt. After the controller has gathered the available information, it transitions to the correction state.

Correction

The controller attempts to correct errors in the correction state. The controller always passes through the correction state, even if correction is disabled. During the correction state, `status_correction` is true.

If the error is a CRC-only error, the controller sets `status_uncorrectable` and generates a report on the Monitor Interface. It then transitions to the classification state. If the error is not a CRC-only error, then the behavior of the controller depends on how it has been configured to correct errors.

If the controller is configured for correction by replace, it will generate a replacement data request on the Fetch Interface. In the system-level design example, the EXT shim translates this request into a read of the external memory. The return data is provided to the controller by the EXT shim. The controller then performs active partial reconfiguration to re-write the frame with the correct contents. The controller clears `status_uncorrectable` and generates a report on the Monitor Interface. It then transitions to the classification state.

If the controller is configured for correction by repair or correction by enhanced repair, it will attempt to correct the error using algorithmic methods. If the error is correctable, the controller performs active partial reconfiguration to re-write the frame with the corrected contents and clears `status_uncorrectable`. Otherwise, the controller sets `status_uncorrectable`. In either case, the controller generates a report and then transitions to the classification state.

Classification

The controller classifies errors in the classification state. The controller always passes through the classification state, even if classification is disabled. During the classification state, `status_classification` is true.

All errors signaled as uncorrectable during the correction state are signaled as essential. The only reason an error can be uncorrectable is because it cannot be located. And, if this is the case, the controller cannot look up the error to determine whether it is essential. In these cases, the controller sets `status_essential`, generates a report, and then

transitions to the idle state. Once an uncorrectable error is encountered, the controller does not continue looking for errors. At this point, the FPGA device must be reconfigured.

The treatment of errors signaled as correctable during the correction state depends on the controller option setting. If error classification is disabled, all correctable errors are unconditionally signaled as essential. If error classification is enabled, the controller will generate a classification data request on the Fetch Interface. In the system-level design example, the EXT shim translates this request into a read of the external memory. The return data is provided to the controller by the EXT shim. With this data, the controller then determines whether it is essential. In all cases, the controller generates a report, changes `status_essential` as appropriate, and then transitions to the observation state to resume looking for errors.

Idle

The idle state is similar to the observation state, except that the controller does not observe the FPGA configuration system for indication of error conditions. The idle state is indicated by the de-assertion of all five state bits on the Status Interface. If the controller receives a command (from either the Error Injection Interface or the Monitor Interface), then the controller will process the received command. The error injection commands are only supported in the idle state.

The "enter observation" command may be applied through either the Error Injection Interface or the Monitor Interface, and is used to return the controller to the observation state so that errors may be detected.

The "status report" command is not frequently used; it provides some diagnostic information, and may be helpful as a mechanism to "ping" the controller. This command is only supported on the Monitor Interface.

Any desired set of "error injection" commands may be applied through either the Error Injection Interface or the Monitor Interface. These commands direct the controller to perform error injections. The primary reason the idle state exists is to halt actions taken in response to error detections so that multi-bit errors can be constructed.

Injection

The controller performs error injections in the injection state. The controller always passes through the injection state in response to a valid error injection command issued from the idle state, even if error injection is disabled; this can occur if error injection commands are presented on the Monitor Interface, as the Monitor Interface exists even when error injection is disabled. During the injection state, `status_injection` is true.

The error injection process is a simple read-modify-write to invert one configuration memory bit at an address specified as part of the error injection command.

The controller always transitions from the injection state back to the idle state. Multi-bit errors may be constructed by repeated error injections commands, each resulting in a transition through the injection state. At the end of error injection, the controller must be moved from the idle state back into the observation state.

Fatal Error

The controller enters the fatal error state when it detects an internal inconsistency. Although very unlikely, it is possible for the controller to halt due to soft errors that affect the controller-related configuration memory or the controller design state elements.

The fatal error state is indicated by the assertion of all five state bits on the Status Interface, along with a fatal error report message. This condition is non-recoverable, and the FPGA device must be reconfigured.

Error Injection Interface Commands

The Error Injection Interface commands define what a user may send to the controller through the Error Injection Interface. This command set offers basic capability to inject errors.

Commands are presented by applying a value to the `inject_address` bus, and then asserting the `inject_strobe` signal. Once a command is presented, do not present another command until the Status Interface has confirmed completion of the previous command.

Directed State Changes

The controller may be moved between observation and idle states by a directed state change. The command format is shown in Figure 8-14 through Figure 8-17, with "X" representing a "don't care."

39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 8-14: **7 Series Enter Idle State Command**

39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 8-15: **7 Series Enter Observation State Command**

35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 8-16: **Virtex-6 and Spartan-6 Enter Idle State Command**

35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Figure 8-17: **Virtex-6 and Spartan-6 Enter Observation State Command**

Completion of the command is indicated on the Status Interface by a change in the state outputs indicating the controller has entered the requested state.

Error Injection

There are two addressing schemes to specify the frame address for an error injection. These are linear frame addressing and physical frame addressing. Linear frame addressing is

simple, but the addresses do not provide information about type and physical location of the frame. The error injection command formats for linear frame address are shown in Figure 8-18 through Figure 8-20, with borders marking nibble boundaries and shading marking separate bit fields in the command.

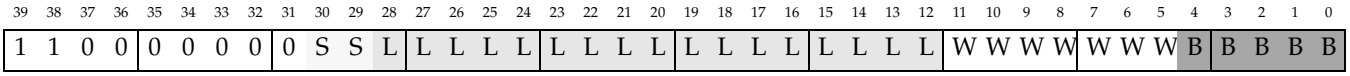

Figure 8-18: **7 Series Error Injection Command (Linear Frame Address)**

Where:

- SS = SLR number (2-bit)
- LLLLLLLLLLLLLLLL = linear frame address (17-bit)
- WWWWWWW = word address (7-bit)
- BBBBB = bit address (5-bit)

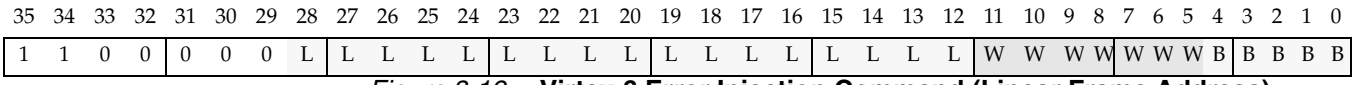

Figure 8-19: **Virtex-6 Error Injection Command (Linear Frame Address)**

Where:

- LLLLLLLLLLLLLLLL = linear frame address (17-bit)
- WWWWWWW = word address (7-bit)
- BBBBB = bit address (5-bit)

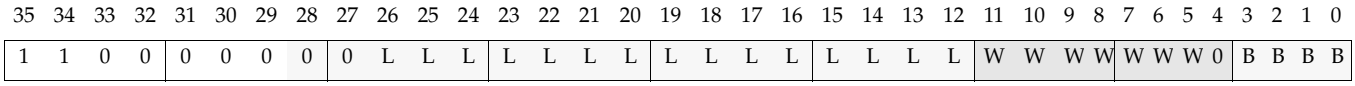

Figure 8-20: **Spartan-6 Error Injection Command (Linear Frame Address)**

Where:

- LLLLLLLLLLLLLLL = linear frame address (15-bit)
- WWWWWWW = word address (7-bit)
- BBBB = bit address (4-bit)

An error injection command using physical frame address has the form shown in Figure 8-21 through Figure 8-23.

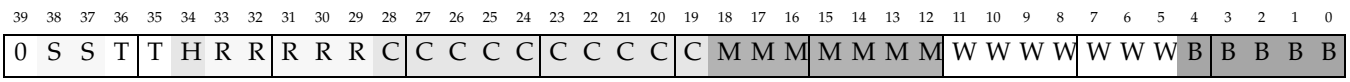

Figure 8-21: **7 Series Error Injection Command (Physical Frame Address)**

Where:

- SS = SLR number (2-bit)
- TT = block type (2-bit)
- H = half address (1-bit)
- RRRRR = row address (5-bit)
- CCCCCCCCCC = column address (10-bit)
- MMMMMMM = minor address (7-bit)
- WWWWWWW = word address (7-bit)
- BBBBB = bit address (5-bit)

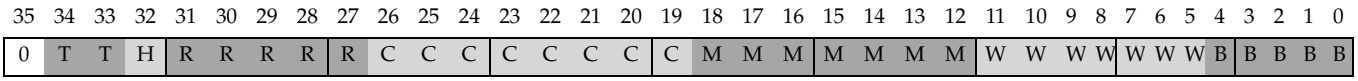

Figure 8-22: **Virtex-6 Error Injection Command (Physical Frame Address)**

Where:

- TT = block type (2-bit)
- H = half address (1-bit)
- RRRRR = row address (5-bit)
- CCCCCCCC = column address (8-bit)
- MMMMMMM = minor address (7-bit)
- WWWWWWW = word address (7-bit)
- BBBBB = bit address (5-bit)

Completion of the command is indicated on the Status Interface by the de-assertion of the `status_injection` output indicating the controller has exited the error injection state.

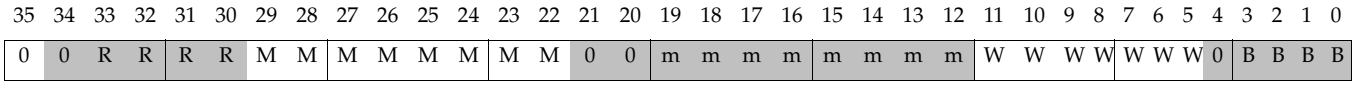

Figure 8-23: **Spartan-6 Error Injection Command (Physical Frame Address)**

Where:

- RRRR = row address (4-bit)
- MMMMMMMM = major address (8-bit)
- mmmmmmmm = minor address (8-bit)
- WWWWWWW = word address (7-bit)
- BBBB = bit address (4-bit)

Monitor Interface Commands

The Monitor Interface commands define what a user may send to the controller through the Monitor Interface. This command set is intended to offer a superset of the "command capability" available from the Error Injection Interface.

Directed State Changes

The controller may be moved between observation and idle states by a directed state change. "I" command is used to enter idle state. "O" command is used to enter observation state.

Status Report

"S" command is used to request a status report from the controller. The status report format is detailed in the next section which describes status messages generated by the controller. The controller will accept this command when in idle or observation states.

Error Injection

"N" command is used to perform an error injection. The controller will only accept this command when in the idle state. The format of the command is:

```
N {9-digit hex value}    Virtex-6 and Spartan-6
N {10-digit hex value}   7 Series
```

Issuing this command is analogous to presenting an error injection command on the Error Injection Interface. The hex value supplied with this command represents the same value that would be applied to the Error Injection Interface.

Monitor Interface Messages

The Monitor Interface messages define what messages a user may expect from the controller through the Monitor Interface. This message set is intended to offer a superset of the "reporting" available from the Status Interface.

Initialization Report

As the controller performs the initialization sequence, it generates the initialization report. This report contains diagnostic information. This report is generated only once when the controller first starts. The 7 series device initialization report looks like this:

```
X7_SEM_V3_1             Name and Version
SC 01                   State Change, Initialization
FS {2-digit hex value}  Core Configuration Information
ICAP OK                 Status: ICAP Available
RDBK OK                 Status: Readback Active
INIT OK                 Status: Completed Setup
SC 02                   State Change, Observation
```

The Virtex-6 and Spartan-6 device initialization reports are virtually identical except the controller reports its name and version as V6_SEM_V3_1 and S6_SEM_V3_1, respectively.

Command Prompt

The command prompt issued by the controller is one of two characters, depending on the controller state. If the controller is in observation state (the default state after initialization completes) the prompt issued is `O>`. If the controller is in idle state, the prompt issued is `I>`.

State Change Report

Any time the controller changes state, the controller also issues a state change report. The report is a single line with the following format:

```
SC {2-digit hex value}
```

The 2-digit hex value is the representation of the Status Interface outputs.

Table 8-2: **State Change Report Decoding**

Report String	State Name
SC 00	Idle
SC 01	Initialization
SC 02	Observation
SC 04	Correction
SC 08	Classification
SC 10	Injection
SC 1F	Fatal Error

Entry into the fatal error state may occur at anytime, even without an explicit state change report. Upon entering this state, the controller issues the following fatal error message:

```
HALT
```

Flag Change Report

Any time the controller changes flags, the controller also issues a flag change report. The report is a single line with the following format:

```
FC {2-digit hex value}
```

The 2-digit hex value is the representation of the Status Interface outputs.

Table 8-3: **Flag Change Report Decoding**

Report String	Condition Name
FC 00	Correctable, Non-Essential
FC 20	Uncorrectable, Non-Essential
FC 40	Correctable, Essential
FC 60	Uncorrectable, Essential

Status Report

A status report provides more information about the controller state. It is a multiple-line report that is generated in response to the `S` command, provided the controller is in the observation or idle states. The 7 series device status report has the following format:

```
MF {8-digit hex value}   Maximum Frame (linear count)
SN {2-digit hex value}   SLR Number
SC {2-digit hex value}   Current State
FC {2-digit hex value}   Current Flags
FS {2-digit hex value}   Feature Set
```

The Virtex-6 and Spartan-6 device status reports are virtually identical except that the Maximum Frame is reported as a 6-digit hex value and the SLR Number is not reported.

Error Detection Report

Upon detection of an error condition, the controller will correct the error as quickly as possible. Therefore, the report information is actually generated after the correction has taken place, assuming it is possible to correct the error. The following scenarios exist:

Diagnosis: CRC error only [cannot identify location or number bits in error]

```
SC 04
CRC
```

Diagnosis: Enhanced repair checksum buffer uncorrectable error

```
SC 04
BFR
```

Diagnosis: 1-bit ECC error, SYNDROME is valid. Controller reports physical frame address, linear frame address, word in frame, and bit in word.

```
SC 04
SED OK
PA {8-digit hex value}
LA {8-digit hex value}
WD {2-digit hex value} BT {2-digit hex value}
```

Diagnosis: 1-bit ECC error, SYNDROME is invalid. Controller reports physical frame address and linear frame address.

```
SC 04
SED NG
PA {8-digit hex value}
LA {8-digit hex value}
```

Diagnosis: 2-bit ECC error. Controller reports physical frame address and linear frame address.

```
SC 04
DED
PA {8-digit hex value}
LA {8-digit hex value}
```

The Virtex-6 and Spartan-6 device error detection reports are virtually identical except that frame addresses are reported as 6-digit hex values.

Error Correction Report

The error correction process varies depending on the controller configuration and the nature of what has been detected and what can be corrected.

The general form of the report for an uncorrectable error, or when correction is disabled is:

```
COR
END
```

Followed by:

```
FC 20          Bit 5, uncorrectable set (stale essential flag)
```

or

```
FC 60          Bit 5, uncorrectable set (stale essential flag)
```

The general form of the report for a correctable error is:

```
COR
{correction list}
END
```

Followed by:

```
FC 00          Bit 5, uncorrectable cleared (stale essential flag)
```

or

```
FC 40          Bit 5, uncorrectable cleared (stale essential flag)
```

The {correction list} is one or more lines providing the word in frame and bit in word of each corrected bit. The list can potentially be thousands of lines. This is the same notation used for the error detection report. Each line of the list is formatted as follows:

```
WD {2-digit hex value} BT {2-digit hex value}
```

Error Classification Report

The error classification process involves looking up each of the errors in a frame to determine if any of them are essential. If one or more are identified as essential, the entire event is considered essential.

The general form of the report for a correctable, non-essential event is:

```
SC 08
CLA
END
FC 00          Bit 6, essential is cleared
```

The general form of the report for a correctable, essential event is:

```
SC 08
CLA
{list}
END
FC 40          Bit 6, essential is set
```

The {list} is one or more lines providing the word in frame and bit in word of each essential bit. The list can potentially be as long as 2,592 lines. This is the same notation used for the error detection report. Each line of the list is formatted as follows:

```
WD {2-digit hex value} BT {2-digit hex value}
```

The general form of the report for an uncorrectable event, or when classification is disabled, is shorter. In these cases, the error is assumed to be essential because no information is available to suggest otherwise, and there is little to report:

```
SC 08
FC 60          Bit 6, essential is set
```

System Level

Solution Reliability Estimates

The system-level design example is analyzed in the following section to provide an estimate of the FIT of the solution itself, as implemented in the FPGA. This analysis method is also appropriate for generating quick estimates of other circuits implemented in the FPGA.

In this analysis, all features are considered enabled, with all signals brought to I/O pins. Chipscope is specifically excluded from analysis, as it is unlikely a production design will include this interactive debug and experimentation capability. As a result, the estimate represents an upper bound.

Estimation Data

Xilinx device FIT data is reported in UG116, *Device Reliability Report*. Table 8-4 provides example data for a sample reliability estimation.

Note: The data in Table 8-4 is an example. This example data is for illustrative purposes only and must not be used in critical design decisions. See UG116 for current device FIT data.

Table 8-4: **Example Device FIT Data**

Device	Memory Cell Type	Real Time Soft Error Rate FIT/Mbit
7 Series	Configuration Memory	100
	Block Memory	100
	Distributed Memory (same as Configuration Memory)	100
	Flip Flops	Unspecified
Virtex-6	Configuration Memory	94
	Block Memory	245
	Distributed Memory (same as Configuration Memory)	94
	Flip Flops	Unspecified
Spartan-6	Configuration Memory	182
	Block Memory	381
	Distributed Memory (same as Configuration Memory)	182
	Flip Flops	Unspecified

Table 8-5 provides an approximate relationship between resources and the number of configuration memory cells associated with each resource.

Table 8-5: **Configuration Bits Per Device Feature**

Device	FPGA Device Feature (Includes Routing)	Approximate Number of Configuration Bits
7 Series	Logic Slice	1,166
	Block RAM (36Kb)	9,396
	Block RAM (18Kb)	4,698
	I/O Block	2,850
Virtex-6	Logic Slice	1,166
	Block RAM (36Kb)	9,396
	Block RAM (18Kb)	4,698
	I/O Block	2,850
Spartan-6	Logic Slice	1,166
	Block RAM (18Kb)	4,698
	Block RAM (9Kb)	2,349
	I/O Block	2,850

Typically less than 10% of configuration cells would directly impact the active design if a soft error occurred. Therefore, the sample reliability estimation uses a de-rating factor of 10%.

Sample 7 Series Reliability Estimation

The controller and shims use approximately 250 logic slices, 56 I/O blocks, 3 block RAM (18 Kb), and 9 block RAM (36 Kb) in a mid-size XC7K325T device, with all optional features enabled. Consider the configuration bit contribution:

Configuration Bit FIT = 10% * (250 * 1,166 + 56 * 2,850 + 3 * 4,698 + 9 * 9,396) * 100 FIT/Mbit

Configuration Bit FIT = 5.2 FIT

The controller and shims use several hundred flip flops for data, their contribution is ignored due to the small number of bits.

The controller and shims use 65 LUT RAM. The usage breakdown is as follows:

- The MON shim uses 31 LUT RAM for data buffering, but the buffers are generally empty and data corruption not observable. These memory bits are therefore ignored.
- The controller uses 34 LUT RAM for data storage. Errors in used locations are highly likely halt the controller. Approximately 256 memory bits are used.

  LUT RAM Bit FIT = 100% * 256 * 100 FIT/Mbit

  LUT RAM Bit FIT = 0.02 FIT

The controller uses three block RAM (18 Kb) and nine block RAM (36 Kb). The usage breakdown is:

- An internal buffer uses one block RAM. In the data array, 9600 bits are allocated to data buffers used in correction and classification; a soft error here would only cause potential issue if it occurred *during* mitigation activity. No permanent data resides here; these are therefore ignored. Another 7480 bits are allocated to constant storage; errors in these locations are highly likely to break the controller and must be considered in the analysis. The remaining 1352 bits are unused.
- The controller firmware resides in two block RAMs. The word count is approximately 1932 out of 2048, with at least 336 of the used words only executed one time at system start and therefore ignored. The number of bits considered for the analysis is 28728.

  Block RAM Bit FIT = 100% * 36208 * 100 FIT/Mbit

  Block RAM Bit FIT = 3.5 FIT
- The enhanced repair algorithm within the 7 series Controller stores frame-level CRC checksums in the remaining block RAM (36 Kb). These block RAM contents are protected by built-in ECC of the block RAM, and do contribute to the Block RAM Bit FIT. As computed above, these block RAMs do contribute to the Configuration Bit FIT

The total controller fit is then:

5.2 FIT + 0.02 FIT + 3.5 FIT ≈ 8.7 FIT

Sample Virtex-6 Reliability Estimation

The controller and shims use approximately 148 logic slices, 52 I/O blocks, and 3 block RAM (18 Kb). Consider the configuration bit contribution:

Configuration Bit FIT = 10% * (148 * 1,166 + 52 * 2,850 + 3 * 4,698) * 94 FIT/Mbit

Configuration Bit FIT = 2.9 FIT

The controller and shims use several hundred flip flops for data, their contribution is ignored due to the small number of bits.

The controller and shims use 65 LUT RAM. The usage breakdown is as follows:

- The MON shim uses 31 LUT RAM for data buffering, but the buffers are generally empty and data corruption not observable. These memory bits are therefore ignored.
- The controller uses 34 LUT RAM for data storage. Errors in used locations are highly likely halt the controller. Approximately 256 memory bits are used.

  LUT RAM Bit FIT = 100% * 256 * 94 FIT/Mbit

  LUT RAM Bit FIT = 0.02 FIT

The controller uses three block RAM (18 Kb). The usage breakdown is:

- An internal buffer uses one block RAM. In the data array, 10368 bits are allocated to data buffers used in correction and classification; a soft error here would only cause potential issue if it occurred *during* mitigation activity. No permanent data resides here; these are therefore ignored. Another 7552 bits are allocated to constant storage; errors in these locations are highly likely to break the controller and must be considered in the analysis. The remaining 512 bits are unused.
- The controller firmware resides in two block RAMs. The word count is approximately 1550 out of 2048, with at least 150 of the used words only executed one time at system start and therefore ignored. The number of bits considered for the analysis is 25200.

  Block RAM Bit FIT = 100% * 32752 * 245 FIT/Mbit

  Block RAM Bit FIT = 7.7 FIT

The total controller fit is then:

2.9 FIT + 0.03 FIT + 7.7 FIT ≈ 10.6 FIT

Sample Spartan-6 Reliability Estimation

The controller and shims use approximately 249 logic slices, 52 I/O blocks, and 10 block RAM (18 Kb) in a mid-size XC6SLX45T device. Consider the configuration bit contribution:

Configuration Bit FIT = 10% * (249 * 1,166 + 52 * 2,850 + 10 * 4,698) * 182 FIT/Mbit

Configuration Bit FIT = 8.4 FIT

The controller and shims use several hundred flip flops for data, their contribution is ignored due to the small number of bits.

The controller and shims use 70 LUT RAM. The usage breakdown is as follows:

- The MON shim uses 36 LUT RAM for data buffering, but the buffers are generally empty and data corruption not observable. These memory bits are therefore ignored.
- The controller uses 34 LUT RAM for data storage. Errors in used locations are highly likely halt the controller. Approximately 256 memory bits are used.

  LUT RAM Bit FIT = 100% * 256 * 182 FIT/Mbit

  LUT RAM Bit FIT = 0.04 FIT

The controller uses ten block RAM (18 Kb). The usage breakdown is:

- An internal buffer uses one block RAM. In the data array, 3268 bits are allocated to data buffers used in correction and classification; a soft error here would only cause potential issue if it occurred during mitigation activity. No permanent data resides here; these are therefore ignored. Another 6812 bits are allocated to constant storage; errors in these locations are highly likely to break the controller and must be considered in the analysis. The remaining 8352 bits are unused.
- The controller firmware resides in two block RAMs. The word count is approximately 1732 out of 2048, with at least 444 of the used words only executed one time at system start or as debug and therefore ignored. The number of bits considered for the analysis is 23184.
- The soft logic FRAME ECC module within the Spartan-6 Controller stores the ECC checksums in the remaining block RAM. These block RAM contents are protected by the FRAME ECC, and do not contribute to the Block RAM Bit FIT. As computed above, these block RAMs do contribute to the Configuration Bit FIT.

  Block RAM Bit FIT = 100% * (6812+23184) * 381 FIT/Mbit

  Block RAM Bit FIT = 10.9 FIT

The total controller fit is then:

8.4 FIT + 0.04 FIT + 10.9 FIT ≈ 19 FIT

Solution Latency Estimates

The error mitigation latency of the solution is defined as the total time that elapses between the creation of an error condition and the conclusion of the mitigation process. The mitigation process consists of detection, correction, and classification. Depending on the application, error mitigation latency may be a key metric to consider.

Estimation Data

The solution behaviors are based on processing of FPGA configuration frames. Single-bit errors always reside in a single frame. Generally, an *N*-bit error may present in a variety of ways, ranging from one frame containing all bit errors, to *N* frames each containing one bit error. When multiple frames are affected by an error, the sequence of detection, correction, and classification is repeated for each affected frame.

The solution will properly mitigate an arbitrary workload of error events. The error mitigation latency estimation of an arbitrary workload is complex. This section focuses on the common case involving a single frame, but provides insight into the controller behavior to aid in understanding other scenarios.

Start-Up Latency

Start-up latency is the delay between the end of FPGA configuration and the completion of the controller initialization, as marked by entry into the observation state. This latency is a function of the FPGA size (frame count) and the solution clock frequency. It is also a function of the selected correction mode.

The start-up latency is incurred only once. It is not part of the mitigation process. Table 8-6 illustrates start-up latency, decomposed into sub-steps of boot and initialization. The boot time is independent of the selected correction mode, while the initialization time is dependent on the selected correction mode.

Table 8-6: **Start-Up Latency**

Device		Boot Time at ICAP Fmax	Initialization Time at ICAP Fmax (Repair / Replace)	Initialization Time at ICAP Fmax (Enhanced Repair)
7 Series	XC7K70T	160 ms	25 ms	3050 ms
	XC7K160T	160 ms	55 ms	6650 ms
	XC7K325T	160 ms	100 ms	12150 ms
	XC7K355T	160 ms	115 ms	13650 ms
	XC7K410T	160 ms	130 ms	15650 ms
	XC7K420T	160 ms	135 ms	15950 ms
	XC7K480T	160 ms	155 ms	18200 ms
	XC7VX330T	160 ms	110 ms	13200 ms
	XC7VX415T	160 ms	140 ms	16750 ms
	XC7VX485T	160 ms	165 ms	19750 ms
	XC7VX550T	160 ms	190 ms	22350 ms
	XC7V585T	160 ms	180 ms	21300 ms
	XC7VX690T	160 ms	235 ms	27950 ms
	XC7VX980T	160 ms	305 ms	36450 ms

The start-up latency is the sum of the boot and initialization latency, using the correct column of initialization latency data for the selected correction mode. The start-up latency at the actual frequency of operation may be estimated using data from Table 8-6 and Equation 8-3.

$$StartUpLatency_{ACTUAL} = StartUpLatency_{\mathrm{ICAP\_F}_{MAX}} \cdot \left[\frac{\mathrm{ICAP\_F}_{MAX}}{Frequency_{ACTUAL}}\right] \quad \textit{Equation 8-3}$$

Error Detection Latency

Error detection latency is the major component of the total error mitigation latency. Error detection latency is a function of the FPGA size (frame count) and the solution clock frequency. It is also a function of the type of error and the relative position of the error with respect to the position of the silicon readback process. Table 8-7 illustrates full device scan times.

Table 8-7: **Device Scan Times at ICAP Maximum Frequency**

Device		ICAP F_{MAX}	Scan Time at ICAP F_{MAX}
7 Series	XC7K70T	70 MHz	8.4 ms
	XC7K160T	70 MHz	18.4 ms
	XC7K325T	70 MHz	33.6 ms
	XC7K355T	70 MHz	37.8 ms
	XC7K410T	70 MHz	43.3 ms
	XC7K420T	70 MHz	44.1 ms
	XC7K480T	70 MHz	50.4 ms
	XC7VX330T	70 MHz	36.6 ms
	XC7VX415T	70 MHz	46.4 ms
	XC7VX485T	70 MHz	54.7 ms
	XC7VX550T	70 MHz	61.9 ms
	XC7V585T	70 MHz	58.9 ms
	XC7VX690T	70 MHz	77.4 ms
	XC7VX980T	70 MHz	100.9 ms

Table 8-7: **Device Scan Times at ICAP Maximum Frequency *(Cont'd)***

Device		ICAP F_{MAX}	Scan Time at ICAP F_{MAX}
Virtex-6	XC6VCX75T	100 MHz	6.2 ms
	XC6VCX130T	100 MHz	10.4 ms
	XC6VCX195T	100 MHz	15.0 ms
	XC6VCX240T	100 MHz	18.0 ms
	XC6VHX250T	100 MHz	18.7 ms
	XC6VHX255T	100 MHz	18.7 ms
	XC6VHX380T	100 MHz	28.0 ms
	XC6VHX565T	100 MHz	39.3 ms
	XC6VLX75T	100 MHz	6.2 ms
	XC6VLX130T	100 MHz	10.4 ms
	XC6VLX195T	100 MHz	15.0 ms
	XC6VLX240T	100 MHz	18.0 ms
	XC6VLX365T	100 MHz	25.2 ms
	XC6VLX550T	100 MHz	37.7 ms
	XC6VLX760	100 MHz	49.9 ms
	XC6VSX315T	100 MHz	24.1 ms
	XC6VSX475T	100 MHz	36.1 ms
Spartan-6	XC6SLX4	50 MHz	2.7 ms
	XC6SLX9	50 MHz	2.7 ms
	XC6SLX16	50 MHz	3.9 ms
	XC6SLX25(T)	50 MHz	6.6 ms
	XC6SLX45(T)	50 MHz	11.9 ms
	XC6SLX75(T)	50 MHz	20.0 ms
	XC6SLX100(T)	35 MHz	37.7 ms
	XC6SLX150(T)	35 MHz	50.6 ms

The device scan time for the target device, at the actual frequency of operation, can be estimated using data from Table 8-7 and Equation 8-4.

$$ScanTime_{ACTUAL} = ScanTime_{\mathrm{ICAP\_F}_{MAX}} \cdot \left[\frac{\mathrm{ICAP\_F}_{MAX}}{Frequency_{ACTUAL}}\right] \qquad \textit{Equation 8-4}$$

The error detection latency can be bounded as follows:

- Absolute minimum error detection latency is effectively zero.
- Typical error detection latency for detection by ECC is 0.5 * Scan $Time_{ACTUAL}$
- Maximum error detection latency for detection by ECC is Scan $Time_{ACTUAL}$

- Absolute maximum error detection latency for detection by CRC alone is 2.0 * Scan $Time_{ACTUAL}$

The frame-based ECC method used will always detect single, double, triple, and all odd-count bit errors in a frame. The remaining error types are usually detected by the frame-based ECC method as well. It is rare to encounter an error that defeats the ECC and is detected by CRC alone.

Error Correction Latency

After detecting an error, the solution attempts correction. Errors may be correctable depending on the selected correction mode and error type. Table 8-8 provides typical error correction latency with no throttling on the Monitor Interface.

Table 8-8: **Typical Error Correction Latency, No Throttling on Monitor Interface**

Device Family	Correction Mode	Errors in Frame (Correctability)	Typical Error Correction State at ICAP_F_{MAX}
7 Series FPGA	Repair	1-bit (Correctable)	865 µs
		2-bit (Uncorrectable)	30 µs
	Enhanced Repair	1-bit (Correctable)	865 µs
		2-bit (Correctable)	26780 µs
		2-bit (Uncorrectable)	13015 µs
		BFR-only (Uncorrectable)	15 µs
	Replace	Any (Correctable)	1175 µs
	Any	CRC-only (Uncorrectable)	15 µs
Virtex-6 FPGA	Repair	1-bit (Correctable)	490 µs
		2-bit (Uncorrectable)	20 µs
	Replace	Any (Correctable)	660 µs
	Any	CRC-only (Uncorrectable)	10 µs
Spartan-6 FPGA LX4 through LX75(T)	Repair	1-bit (Correctable)	370 µs
		2-bit (Uncorrectable)	35 µs
	Any	CRC-only (Uncorrectable)	15 µs
Spartan-6 FPGA LX100(T) through LX150(T)	Repair	1-bit (Correctable)	525 µs
		2-bit (Uncorrectable)	50 µs
	Any	CRC-only (Uncorrectable)	25 µs

The error correction latency at the actual frequency of operation may be estimated using data from Table 8-8 and Equation 8-5.

$$CorrectionLatency_{ACTUAL} = CorrectionLatency_{\text{ICAP\_F}_{MAX}} \cdot \left[\frac{\text{ICAP\_F}_{MAX}}{Frequency_{ACTUAL}}\right] \quad \textit{Equation 8-5}$$

Error Classification Latency

After attempting correction of an error, the solution classifies the error. The classification result depends on the correction mode, error type, error location, and selected classification mode. Table 8-9 provides typical error classification latency with no throttling on the Monitor Interface.

Table 8-9: **Typical Error Classification Latency, No Throttling on Monitor Interface**

Device Family	Correction Mode	Errors in Frame (Correctability)	Classification Mode	Typical Error Classification State at ICAP_F_{MAX}
7 Series FPGA	Any	Correctable	Enabled	1070 µs
	Any	Uncorrectable	Disabled	10 µs
	Any	Uncorrectable	Any	10 µs
Virtex-6 FPGA	Any	Correctable	Enabled	610 µs
	Any	Correctable	Disabled	10 µs
	Any	Uncorrectable	Any	10 µs
Spartan-6 FPGA LX4 through LX75(T)	Repair	Correctable	Enabled	435 µs
		Correctable	Disabled	10 µs
		Uncorrectable	Any	10 µs
Spartan-6 FPGA LX100(T) through LX150(T)	Repair	Correctable	Enabled	620 µs
		Correctable	Disabled	15 µs
	Any	Uncorrectable	Any	15 µs

The error classification latency at the actual frequency of operation may be estimated using data from Table 8-9 and Equation 8-6.

$$ClassificationLatency_{ACTUAL} = ClassificationLatency_{\text{ICAP\_F}_{MAX}} \cdot \left[\frac{\text{ICAP\_F}_{MAX}}{Frequency_{ACTUAL}}\right] \quad \textit{Equation 8-6}$$

Sources of Additional Latency

It is highly desirable to avoid throttling on the Monitor Interface, as it will increase the total error mitigation latency:

- After an attempted error correction, but before exiting the error correction state (at which time the correctable status flag is updated), the controller issues a detection and correction report through the Monitor Interface. If the MON Shim transmit FIFO becomes full during this report generation, the controller dwells in this state until it has written the entire report into the MON Shim transmit FIFO. When this happens, the error correction latency increases.
- After classifying an error, but before exiting the error classification state (at which time the essential status flag is updated), the controller issues a classification report through the Monitor Interface. If the MON Shim transmit FIFO becomes full during

this report generation, the controller dwells in this state until it has written the entire report into the MON Shim transmit FIFO. When this happens, the error classification latency increases.

The approaches to completely eliminate the potential bottleneck are to remove the MON Shim and leave the Monitor Interface unused, or use the Monitor Interface with a peripheral that never signals a buffer full condition. In the event the Monitor Interface is unused, the Status Interface remains available for monitoring activity.

For peripherals where the potential bottleneck is a concern, it can be mitigated. This is accomplished by adjusting the transmit FIFO size to accommodate the longest burst of status messages that are anticipated so that the transmit FIFO never goes full during error mitigation.

In the event that a transmit FIFO full condition does occur, the increase in the total error mitigation latency is roughly estimated as shown in Equation 8-7.

$$AdditionalLatency = \frac{MessageLength - BufferDepth}{TransmissionRate} \quad \text{Equation 8-7}$$

In Equation 8-7, MessageLength-BufferDepth is in message bytes, and the Transmission Rate is in bytes per unit of time.

Sample Latency Estimation

The first sample estimation illustrates the calculation of typical error mitigation latency for a single-bit error by the solution implemented in an XC6VLX240T device with a 66 MHz clock. The solution is configured for error correction by repair, with error classification disabled. The initial assumption is that no throttling occurs on the Monitor Interface.

$$DetectionLatency = 0.5 \cdot ScanTime_{ACTUAL} = 0.5 \cdot 18.0ms \cdot \left\lceil \frac{100MHz}{66MHz} \right\rceil = 13.636ms \quad \text{Equation 8-8}$$

$$CorrectionLatency = 490\mu s \cdot \left\lceil \frac{100MHz}{66MHz} \right\rceil = 0.742ms \quad \text{Equation 8-9}$$

$$ClassificationLatency = 10\mu s \cdot \left\lceil \frac{100MHz}{66MHz} \right\rceil = 0.015ms \quad \text{Equation 8-10}$$

$$MitigationLatency = 13.636ms + 0.742ms + 0.015ms = 14.393ms \quad \text{Equation 8-11}$$

The second sample estimation illustrates the calculation of typical error mitigation latency for a two-bit error by the solution implemented in an XC6VLX240T device with a 66 MHz clock. The solution is configured for error correction by replace, with error classification enabled. Again, it is assumed that no throttling occurs on the Monitor Interface.

$$DetectionLatency = 0.5 \cdot ScanTime_{ACTUAL} = 0.5 \cdot 18.0ms \cdot \left\lceil \frac{100MHz}{66MHz} \right\rceil = 13.636ms \quad \text{Equation 8-12}$$

$$CorrectionLatency = 660\mu s \cdot \left\lceil \frac{100MHz}{66MHz} \right\rceil = 1.000ms \quad \text{Equation 8-13}$$

$$ClassificationLatency = 610\mu s \cdot \left\lceil \frac{100MHz}{66MHz} \right\rceil = 0.924ms \quad \text{Equation 8-14}$$

$$MitigationLatency = 13.636ms + 1.000ms + 0.924ms = 15.560ms$$ *Equation 8-15*

The final sample estimation illustrates an assessment of the additional latency that would result from throttling on the Monitor Interface. Assume the message length in both the first and second samples is approximately 80 bytes, but the buffer depth of the MON Shim is 32 bytes. Further, the MON Shim has been modified to raise the bit rate from 9600 baud to 460800 baud. Note that the standard 8-N-1 protocol used requires 10 bit times on the serial link to transmit a one byte payload:

$$AdditionalLatency = \frac{80bytes - 32bytes}{\left[\frac{460800bittimes}{s} \cdot \frac{byte}{10bittimes} \cdot \frac{s}{1000ms}\right]} = 1.042ms$$ *Equation 8-16*

This result illustrates that the additional latency resulting from throttling on the Monitor Interface can become significant, especially when the data transmission is serialized and the data rate is low.

Supervisory Considerations

Although the soft error mitigation solution can operate autonomously, most applications will use the solution in conjunction with an application-level supervisory function. This supervisory function monitors the event reporting from the controller and determines if additional actions are necessary (for example, reconfigure the device or reboot the application). The nature of application-level supervisory functions varies from design to design.

As noted previously, there is a small, yet finite possibility that the soft error mitigation solution is disrupted by a soft error. The solution has indicators of general health that the application-level supervisory function may elect to monitor:

- The controller heartbeat, `status_heartbeat`: This signal is a direct output from the soft error mitigation solution. This signal exhibits pulses which indicate the FPGA configuration system readback process is active. If, during the observation state, these pulses cease for no apparent reason, the application-level supervisory function should conclude that the FPGA configuration system readback process has experienced a fault. This is an uncorrectable, essential error.
- The hardware CRC failure indicator, `INIT_B` (7 Series and Virtex-6 devices only): This signal is a direct output from the hardware readback process. If the readback process detects a hardware CRC failure, it will assert `INIT_B`. A transient assertion of `INIT_B` is expected during FPGA configuration and the controller initialization process, after error injections, and in some cases when soft error events occur. If, during observation state, `INIT_B` indicates an error and the controller does not transition into correction state after the expected duration of a complete readback cycle, the application-level supervisory function should conclude the controller has experienced a fault. This is an uncorrectable, essential error.

The recommended interface between the soft error mitigation solution and the application-level supervisory function for normal communication is the serial interface supported by the MON shim. Using the MON shim introduces a small amount of logic, but drastically reduces the number of I/O required. As a result, the MON shim offers higher reliability than using the direct logic signal based interfaces.

XILINX®

Chapter 9

Customizing the Solution

The system-level design example encapsulates the controller and various shims that serve to interface the controller to other devices. These shims may include I/O Pins, ChipScope, I/O Interfaces, Memory Controllers, or application-specific system management interfaces.

As delivered, the system-level design example is not a reference design, but an integral part of the total solution and fully verified by Xilinx. While designers do have the flexibility to modify the system-level design example, the recommended approach is to use it as delivered. However, if modifications are desired, this chapter provides additional detail required for success.

This chapter does not describe customization of the user application that exists in the system-level design example. This portion of the system-level design example is for demonstration purposes only and not functionally involved in soft error mitigation. The only anticipated customization of the user application is to simply remove it.

HID Shim Customizations

The HID shim is a bridge between the controller and an interface device. The resulting interface may be used to exchange commands and status with the controller. The HID shim is only present in certain controller configurations. When present, it exports access to the Status Interface and Error Injection Interface through the use of ChipScope. When absent, the Status Interface and Error Injection Interface are only accessible through I/O pins.

If desired, the Status Interface and Error Injection Interface may be connected in other ways. These interfaces are easy to disconnect from the HID shim or I/O Pins and reconnect to other logic, such as register files or finite state machines. See the Status Interface in Chapter 8 and Error Injection Interface in Chapter 8.

MON Shim Customizations

The MON shim is a bridge between the controller and a standard RS-232 port. The resulting interface can be used to exchange commands and status with the controller. This interface is designed for connection to processors.

Increase Bit Rate

If the RS-232 interface is needed, changing to the highest feasible bit rate is strongly encouraged. This reduces the potential for throttling of the controller due to status report transmission. See Monitor Interface in Chapter 8 for more information.

Increase Buffer Depth

Increasing the MON shim bit rate is the easiest method to reduce the potential for throttling of the controller due to status report transmission. Another method is to increase the buffer depth. The MON shim contains two FIFOs, a transmit buffer and a receive buffer.

There is no need for the buffer depths to be symmetric and little advantage is gained from increasing the depth of the receive buffer. However, increasing the depth of the transmit buffer reduces the potential for throttling of the controller due to status report transmission.

The Xilinx LogiCORE IP FIFO Generator may be used to create replacements for the MON shim FIFOs. See the product page for the FIFO Generator for more details. The FIFO configuration must be for a common clock (that is, fully synchronous to a single clock) with first word fall through enabled. The data width must be eight, with the depth as great as desired. When making a FIFO replacement, note that an "empty" flag is the logical inverse of a "data present" flag.

Replace with Alternate Function

If an interface other than RS-232 is desired, the MON shim may be replaced with an alternate function. For example, the MON shim could be replaced with a custom processor interface or other scheme for inter-process communication. When replacing the MON shim with an alternate function, it becomes critical to understand the behavior of the controller monitor interface.

For an overview of the signals, see Monitor Interface in Chapter 8. The data exchanged over this interface is ASCII. For a summary of the status and command formats, see Monitor Interface Commands in Chapter 8 and Monitor Interface Messages in Chapter 8.

Figure 9-1 illustrates the protocol used on the transmit portion of the Monitor Interface. When the controller wants to transmit a byte of data, it first samples the `monitor_txfull` signal to determine if the transmit buffer is capable of accepting a byte of data. Provided that `monitor_txfull` is low, the controller transmits the byte of data by applying the data to `monitor_txdata[7:0]` and asserting `monitor_txwrite` for a single clock cycle.

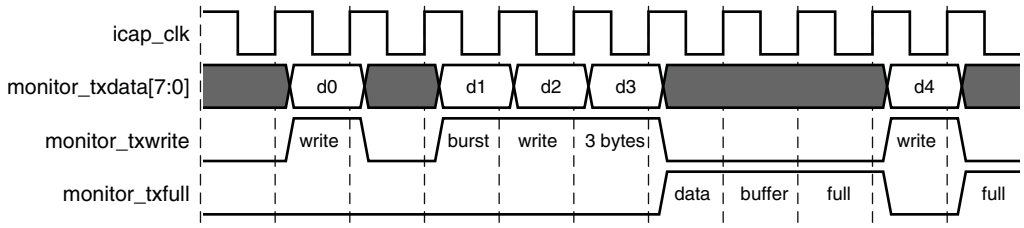

Figure 9-1: **Monitor Interface Transmit Protocol**

The controller may perform burst writes by applying a data sequence to `monitor_txdata[7:0]` and asserting `monitor_txwrite` for multiple cycles. However, the controller will observe `monitor_txfull` so that it never over-runs the transmit buffer.

It is the responsibility of the peripheral receiving the data to correctly track its buffer availability and report it via `monitor_txfull`. In the cycle after the peripheral samples

monitor_txwrite high, it must assert monitor_txfull if the data written completely fills the buffer.

Further, the peripheral must only assert monitor_txfull in response to monitor_txwrite from the controller. Under no circumstances may the peripheral assert monitor_txfull based on events internal to the peripheral. This requirement exists because the controller may sample monitor_txfull several cycles in advance of performing a write.

While monitor_txfull is asserted by the peripheral, the controller will stall if it is waiting to transmit additional data. This can have negative side effects on the error mitigation performance of the controller. For example, if a correction takes place, the controller will successfully correct (or handle) the error and then send a status report. If the entire message can not be accepted by the peripheral, the controller will stall, preventing it from returning to the observation state. Therefore, a custom peripheral must have an adequate balance of buffer depth and data consumption rate.

Figure 9-2 illustrates the protocol used on the receive portion of the monitor interface. When the controller wants to receive a byte of data, it first samples the monitor_rxempty signal to determine if the receive buffer is providing a byte of data. Provided that monitor_rxempty is low, the controller receives the byte of data from monitor_rxdata[7:0] and acknowledges reception by asserting monitor_rxread for a single clock cycle.

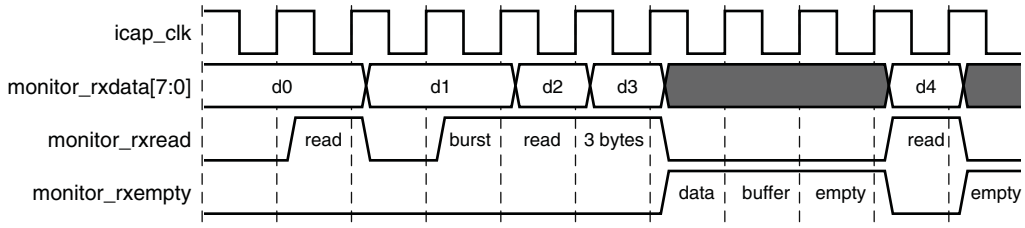

Figure 9-2: **Monitor Interface Receive Protocol**

The controller may perform burst reads by obtaining a data sequence from monitor_rxdata[7:0] and asserting monitor_rxread for multiple cycles. However, the controller will observe monitor_rxempty so that it never under-runs the receive buffer.

It is the responsibility of the peripheral providing the data to correctly track its buffer status and report it via monitor_rxempty. In the cycle after the peripheral samples monitor_rxread high, it must assert monitor_rxempty if the data written completely fills the buffer.

Further, the peripheral must only assert monitor_rxempty in response to monitor_rxread from the controller. Under no circumstances may the peripheral assert monitor_rxempty based on events internal to the peripheral. This requirement exists because the controller may sample monitor_rxempty several cycles in advance of performing a read.

During the initialization state, the controller purges the receive buffer by reading and discarding data until it is empty. The controller assumes the transmit buffer is ready immediately and does not wait for the transmit buffer to empty, as it has no way to observe this condition.

Removal

One possible customization of the MON shim is to entirely eliminate it. If removing the MON shim, Xilinx strongly recommends preserving the two I/O pins used by the MON shim and making those accessible for probing at test points. Even though the MON shim may not be necessary in certain applications, it offers a critical debugging capability that is required when obtaining assistance from Xilinx technical support teams.

To eliminate the MON shim, disconnect and remove it, including any related signals and ports used to bring the RS-232 signals to I/O pins at the design top level. Then, address the exposed monitor interface on the controller:

- Leave all controller monitor interface outputs "open" or "unconnected."
- Connect the controller's `monitor_txfull` input to logic zero.
- Connect the controller's `monitor_rxempty` input to logic one.
- Connect the controller's `monitor_rxdata[7:0]` input to logic zero.

Elimination of the MON shim reduces the size of the solution and also prevents throttling of the controller due to status report transmission.

EXT Shim Customizations

The EXT shim is a bridge between the controller and a standard SPI bus. The resulting interface may be used to fetch data by the controller. This shim is only present in certain controller configurations and is designed for connection to standard SPI flash.

Replace with Alternate Function

If an interface other than SPI bus is needed, the EXT shim may be replaced with an alternate function. For example, the EXT shim could be replaced with a parallel flash memory controller or other scheme for inter-process communication. When replacing the EXT shim with an alternate function, it becomes critical to understand the behavior of the Controller Fetch Interface.

For an overview of the signals, Fetch Interface in Chapter 3. The byte transfer protocols for the Fetch Interface are identical to those of the Monitor Interface.

Figure 9-3 illustrates the protocol used on the transmit portion of the Fetch Interface. When the controller wants to transmit a byte of data, it first samples the `fetch_txfull` signal to determine if the transmit buffer is capable of accepting a byte of data. Provided that `fetch_txfull` is low, the controller transmits the byte of data by applying the data to `fetch_txdata[7:0]` and asserting `fetch_txwrite` for a single clock cycle.

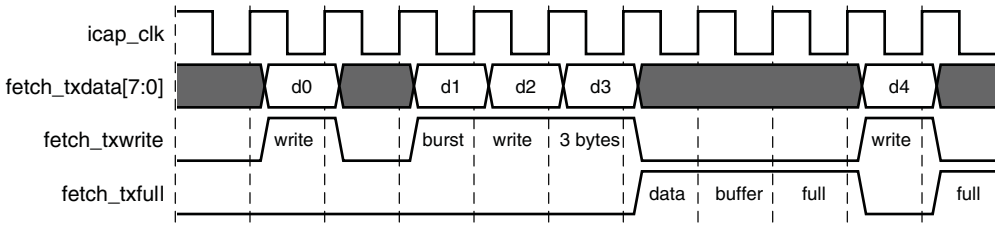

Figure 9-3: **Fetch Interface Transmit Protocol**

The controller may perform burst writes by applying a data sequence to `fetch_txdata[7:0]` and asserting `fetch_txwrite` for multiple cycles. However, the controller will observe `fetch_txfull` so that it never over-runs the transmit buffer.

It is the responsibility of the peripheral receiving the data to correctly track its buffer availability and report it via `fetch_txfull`. In the cycle after the peripheral samples `fetch_txwrite` high, it must assert `fetch_txfull` if the data written completely fills the buffer.

Further, the peripheral must only assert `fetch_txfull` in response to `fetch_txwrite` from the controller. Under no circumstances may the peripheral assert `fetch_txfull` based on events internal to the peripheral. This requirement exists because the controller may sample `fetch_txfull` several cycles in advance of performing a write.

While `fetch_txfull` is asserted by the peripheral, the controller will stall if it is waiting to transmit additional data. This can have negative side effects on the error mitigation performance of the controller.

Figure 9-4 illustrates the protocol used on the receive portion of the fetch interface. When the controller wants to receive a byte of data, it first samples the `fetch_rxempty` signal to determine if the receive buffer is providing a byte of data. Provided that `fetch_rxempty` is low, the controller receives the byte of data from `fetch_rxdata[7:0]` and acknowledges reception by asserting `fetch_rxread` for a single clock cycle.

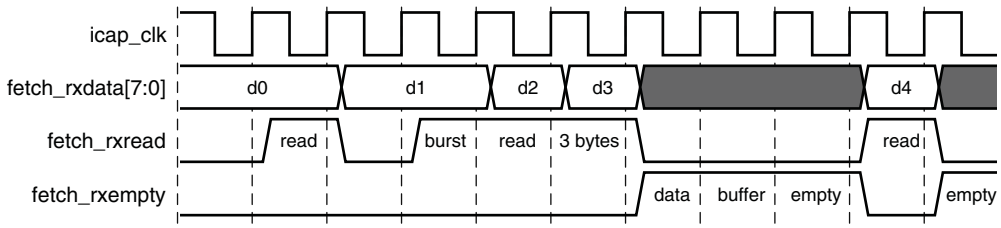

Figure 9-4: **Fetch Interface Receive Protocol**

The controller may perform burst reads by obtaining a data sequence from `fetch_rxdata[7:0]` and asserting `fetch_rxread` for multiple cycles. However, the controller will observe `fetch_rxempty` so that it never under-runs the receive buffer.

It is the responsibility of the peripheral providing the data to correctly track its buffer status and report it via `fetch_rxempty`. In the cycle after the peripheral samples `fetch_rxread` high, it must assert `fetch_rxempty` if the data written completely fills the buffer.

Further, the peripheral must only assert `fetch_rxempty` in response to `fetch_rxread` from the controller. Under no circumstances may the peripheral assert `fetch_rxempty` based on events internal to the peripheral. This requirement exists because the controller may sample `fetch_rxempty` several cycles in advance of performing a read.

During the initialization state, the controller purges the receive buffer by reading and discarding data until it is empty. It then performs two more reads. Reads with an empty buffer condition are decoded as a reset signal. The controller assumes the reset signal initializes the transmit buffer and therefore does not wait for the transmit buffer to empty, as it has no way to observe this condition.

The data exchanged is binary and represents a command-response pair. The controller transmits a 6-byte command sequence to initiate a data fetch. The command sequence is:

- Byte 1: ADD[31:24]

- Byte 2: ADD[23:16]
- Byte 3: ADD[15:8]
- Byte 4: ADD[7:0]
- Byte 5: LEN[15:8]
- Byte 6: LEN[7:0]

In response, the peripheral must fetch LEN[15:0] bytes of data starting from address ADD[31:0], and return this data to the controller. Once the controller has issued a command, the peripheral must fulfill the command with the exact number of requested bytes. For additional information about the organization of the data in the address space, see Creating the External Memory Programming File in Chapter 6.

During the initialization state, the controller purges the receive buffer by reading and discarding data until it is empty. The controller then performs two more reads "beyond empty." The condition of a receive buffer read while the receive buffer is empty is a signal for the peripheral to reset itself. This provides a mechanism to synchronize the command-response protocol. The controller assumes the transmit buffer is ready immediately and does not wait for the transmit buffer to empty, as it has no way to observe this condition.

XILINX®

Chapter 10

Additional Considerations

This chapter provides detailed information on additional considerations. The information here is critical to successful application of the SEM Controller in a complete design.

Unsupported Features and Limitations

Unsupported features and limitations include functional, implementation, and use considerations.

- EasyPath devices are not compatible with the error correction by replace method.
- The SEM Controller initializes and manages the FPGA integrated silicon features for soft error mitigation. When the controller is included in a design, do not include any design constraints related to these features. Similarly, do not use any related bitgen options other than those for generating essential bit data files.
- Software computed ECC and CRC values are not supported.
- Simulation of designs that instantiate the controller is supported. However, it is not possible to observe the controller behaviors in simulation. Simulation of a design including the controller will compile, but the controller will not exit the initialization state. Hardware-based evaluation of the controller behaviors is required. Alternatively, customers can use ISim Hardware Co-simulation to simulate their design. Contact Xilinx for more information.
- Use of bitstream security (encryption and authentication) is not supported by the controller.
- Use of SelectMAP persistence is not supported by the controller.
- When the controller requires storage of configuration data for correction by replace, this data must be available to the controller through the Fetch Interface, typically through the EXT shim. This decouples the controller from the FPGA configuration method and allows customers flexibility in selection of configuration method, configuration data storage, and soft error mitigation solution data storage.
- The EXT shim implementation supports only one SPI flash read command (fast read) in SPI Mode 0 (CPOL = 0, CPHA = 0) to a single SPI flash device.
- Due to potential I/O voltage incompatibility between the FPGA device and standard SPI flash devices, level translation may be required in the design of the SPI memory system.
- ICAP Arbitration, ICAP Switchover, and Partial Reconfiguration are not supported. Only a single ICAP instance is supported, and it must reside at the primary/top physical location.
- Implementation of ICAP Sharing or MultiBoot may be possible in certain circumstances. Contact Xilinx Support for more details.

- Use of design capture, including the use of the capture primitive and related functionality, is not supported by the controller.
- In implementations for Spartan-6 FPGAs, neither Suspend mode nor PLL DRP may be used.
- Controller implementations for Spartan-6 FPGAs operate only on soft errors in Type 0 configuration frames. See UG380, *Xilinx Spartan-6 FPGA Configuration User Guide* for information on Spartan-6 configuration frame types.
- Controller implementations for Virtex-6 FPGAs operate on soft errors in Type 0, Type 2, and Type 3 configuration frames. See UG360, *Xilinx Virtex-6 FPGA Configuration User Guide* for information on Virtex-6 configuration frame types.
- Controller implementations for 7 series FPGAs operate on soft errors in Type 0, Type 2, and Type 3 configuration frames. See UG470, *Xilinx 7 Series FPGA Configuration User Guide*, for information on 7 series configuration frame types.

No Controller Reset

There is deliberately no reset for the controller because the entire configuration of the device cannot be reset. The controller is a monitor of the device configuration from the point when the device is configured until the power is removed (or it is reconfigured). The task of the SEM Controller is to monitor and maintain the original configuration state and not restart from some interim (potentially erroneous) state.

Master Clock Source

The master clock is absolutely critical to the controller and therefore needs to be provided from the most reliable source possible. To achieve the very highest reliability, the clock must be connected as directly as possible to the controller. This means the use of an external oscillator of the desired frequency, connected directly to a pin associated directly with a clock buffer.

The inclusion of any additional logic (for example, clock management blocks) and interconnect into the clock path will result in additional configuration memory being used to control the connection of the clock to the controller. This additional memory has a negative effect on the estimated controller FIT. Although the impact is small, it is best to strive for high reliability unless it poses a significant burden.

The system-level design example, the controller, and the configuration system are all static. This means, any clock frequency can be used up to the specified maximum allowed by the FPGA configuration system or the maximum clock frequency of the system-level design example and controller (whichever is lower). However, higher clock rates result in faster mitigation of errors, which is desirable.

Data Consistency

When using optional features such as error correction by replacement and error classification, the controller requires access to externally stored data. This data is created by bitgen at the same time the programming file for the FPGA is created. The files are related.

Any time the FPGA design is changed and a new programming file is created, the additional data files used by the controller must also be updated. When the hardware

design is updated with the new programming file, the externally stored data must also be updated.

Failure to maintain data consistency may result in unpredictable controller behavior. Xilinx recommends use of an update methodology which ensures that the programming file and the additional data files are always synchronized.

Configuration Memory Masking

By design, certain configuration memory bits can change value during design operation. This is frequently the case where logic slice resources are configured to implement LUTRAM functions such as Distributed RAM or Shift Registers. It also occurs when other resource types with Dynamic Reconfiguration Ports are updated during design operation.

The memory bits associated with these resources must be masked so that they are excluded from CRC and ECC calculations to prevent false error detections. Xilinx FPGA devices implement configuration memory masking to prevent these false error detections. A global control signal, GLUTMASK, selects if masking is enabled or disabled. The controller always enables masking.

7 Series FPGAs

7 Series FPGAs implement fine grain masking at a resource level. This means individual resources, when configured for dynamic operation, have their configuration memory bits masked. Only the required memory bits are masked, without impacting unrelated memory bits. The masked bits are no longer monitored by the controller.

Configuration memory reads of bits associated with masked resources return constant values (either logic one or logic zero). This prevents false error detections. Configuration memory writes to bits associated with masked resources are discarded. This prevents over-writing the contents of dynamic state elements with stale data. A side effect is that error injections into masked resources do not result in error detections.

In many cases (for example, LUTRAM functions) it is possible for the user design to implement data protection on these bits for purposes of soft error mitigation. Another approach is to modify the user design to eliminate the use of features that introduce configuration memory masking.

Virtex-6 FPGAs

Virtex-6 FPGAs implement fine grain masking at a resource level. This means individual resources, when configured for dynamic operation, have their configuration memory bits masked. Only the required memory bits are masked, without impacting unrelated memory bits. The masked bits are no longer monitored by the controller.

Configuration memory reads of bits associated with masked resources return constant values (either logic one or logic zero). This prevents false error detections. Configuration memory writes to bits associated with masked resources are discarded. This prevents over-writing the contents of dynamic state elements with stale data. A side effect is that error injections into masked resources do not result in error detections.

In many cases (for example, LUTRAM functions) it is possible for the user design to implement data protection on these bits for purposes of soft error mitigation. Another approach is to modify the user design to eliminate the use of features that introduce configuration memory masking.

Spartan-6 FPGAs

Spartan-6 FPGAs implement coarse grain masking at a frame level. This means individual resources, when configured for dynamic operation, result in one or more frames of configuration memory being masked. The masked bits are no longer monitored by the controller.

This over-masking can compromise error detection and error injection capability of the controller. The easiest way to avoid over-masking is to eliminate the use of functions that involve dynamic memory bits. However, this is undesirable because it prohibits use of powerful Spartan-6 FPGA features. Generally, a better approach is to understand the masking rules and then apply placement constraints to cluster resources that involve dynamic memory bits into the same configuration memory frames. Consult UG380, *Xilinx Spartan-6 FPGA Configuration User Guide* for additional information.

Configuration memory reads of bits associated with masked resources return constant values (either logic one or logic zero). This prevents false error detections. Configuration memory writes to bits associated with masked resources will overwrite the contents of dynamic state elements. A side effect is that error injections into masked resources do not result in error detections, although the memory contents have changed.

In many cases (for example, LUTRAM functions) it is possible for the user design to implement data protection on these bits for purposes of soft error mitigation. Another approach is to modify the user design to eliminate the use of features that introduce configuration memory masking.