Ncore 3.6 Functional Safety Specification

Revision: 0.55, October 9, 2023

ARTERIS® NCORE 3.6 FUNCTIONAL SAFETY SPECIFICATION

Release Information

Version	*Editor*	*Change*	*Date*
0.5	MF	Initial Document created from Ncore 3.0 Functional Safety Specification, Rev. 0.84 Recreating most drawings to have editable version	03/10/2023
0.51	MF	Added description of partial duplication (3.6) and optimization for ASIL-B (3.8)	03/21/2023
0.52	MF	Added changes to fault register mapping	03/22/2023
0.53	MF	Modification to BIST to include timeout test state	03/24/2023
0.54	MF	Updated drawings and some text	03/24/2023
0.55	MF	Replaced drawings 3.1/3.2	08/29/2023
0.56	ET	3.6 Implimentation Updates	10/9/2023
Legend:	CC	Cheng Chung Wang	
	SD	Said Derradji	
	ET	Eric Taylor	
	MF	Michael Frank	
	Xx	Whoever else edited this document	

Confidentiality Status

Product Status

The information in this document is ***Preliminary***.

Web Address

http://www.arteris.com

Table of Contents

Table of Figures

Table of Tables

Preface

This preface introduces the Arteris® Network-on-Chip Hierarchical Coherency Engine Architecture Specification.

About this document

This technical document is for the Arteris Network-on-Chip Hierarchical Coherency Engine Architecture. It describes the subsystems and their function along with the system's interactions with the external subsystems. It also provides reference documentation and contains programming details for registers.

Product revision status

TBD

Intended audience

This manual is for system designers, system integrators, and programmers who are designing or programming a System-on-Chip (SoC) that uses or intend to use the Arteris Network-on-Chip Hierarchical Coherency System (ANoC-HCS).

Using this document

TBD

Glossary

The Arteris© Glossary is a list of terms used in Arteris© documentation, together with definitions for those terms. The Arteris© Glossary does not contain terms that are industry standard unless the Arteris© meaning differs from the generally accepted meaning.

Typographic conventions

italic

Introduces special terminology, denotes cross-references, and citations.

bold

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

monospace italic

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name. monospace italic Denotes arguments to monospace text where the argument is to be replaced by a specific value. monospace bold Denotes language keywords when used outside example code.

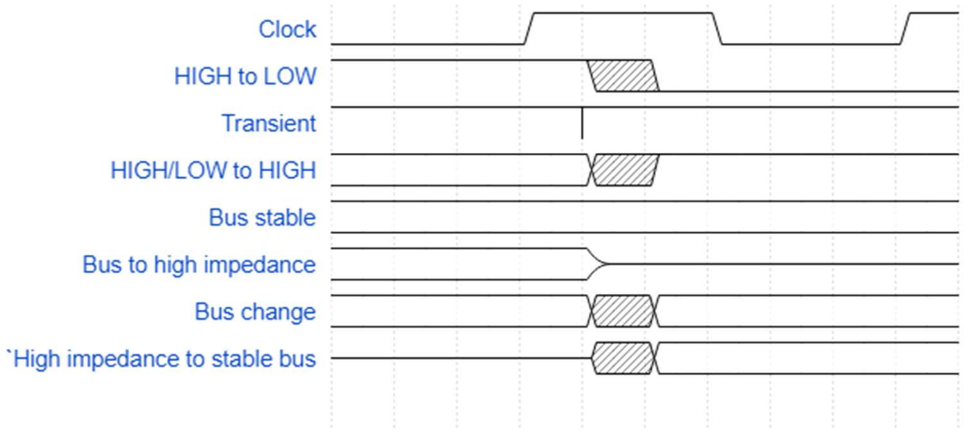
SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the Arteris® Glossary. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.



Signals

The signal conventions are:

Signal level

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW.
Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n

At the start or end of a signal name denotes an active-LOW signal.

Additional reading

This book contains information that is specific to this product. See the following documents for other relevant information.

History of the World II, Mel Brooks.

1 Introduction

This document specifies the Functional Safety features for Ncore 3.0.

A complete Ncore 3.0 system level view with various components including the Functional Safety controller is shown in Figure 1-1. Note that, for the sake of simplicity, not all features are shown in this figure.

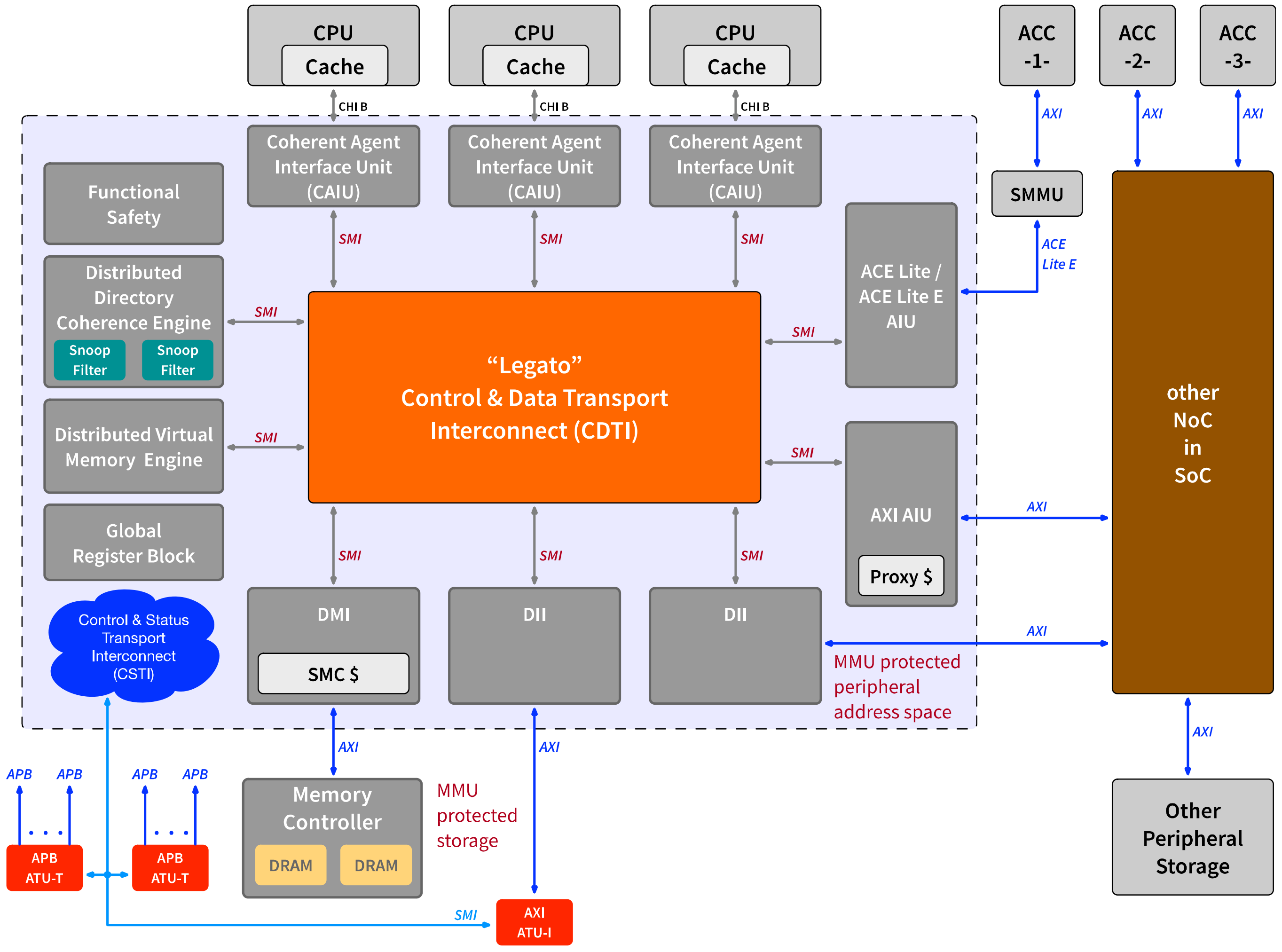

CPU
Cache
CPU
Cache
CPU
Cache
ACC -1-
ACC -2-
ACC -3-
CHI B
AXI
SMMU
ACE Lite E
Functional Safety
Coherent Agent Interface Unit (CAIU)
Distributed Directory Coherence Engine
Snoop Filter
Snoop Filter
SMI
"Legato" Control & Data Transport Interconnect (CDTI)
ACE Lite / ACE Lite E AIU
Distributed Virtual Memory Engine
AXI AIU
Proxy $
other NoC in SoC
Global Register Block
Control & Status Transport Interconnect (CSTI)
DMI
SMC $
DII
DII
MMU protected peripheral address space
APB
APB ATU-T
Memory Controller
DRAM
DRAM
MMU protected storage
AXI ATU-I
Other Peripheral Storage

FIGURE 1-1: NCORE 3.0 SYSTEM LEVEL VIEW

2 Functional Safety Features

The Ncore 3 Functional Safety Architecture supports ASIL-B and ASIL-D by implementing:

Table 2-1: Functional Safety Support

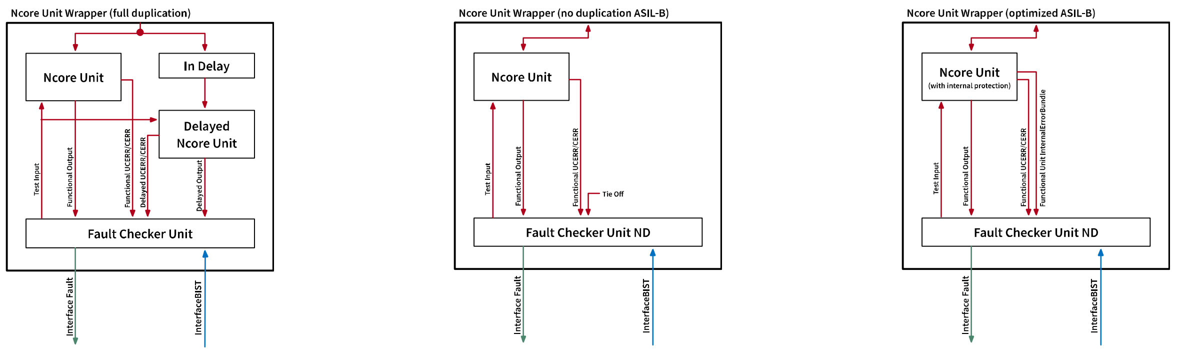
Protection Level	Ncore 3.4	Ncore 3.6	Ncore 3.8
none	• No protection wrappers instantiated • No Fault Checkers • No Functional Safety Controller		
ASIL-B	• Full unit duplication	• Partial unit duplication • Full unit duplication	• Optimized internal protection, no duplication • Partial unit duplication • Full unit duplication
ASIL-D		• Full unit duplication	• Full unit duplication

Protection is provided through

- Unit duplication with 1 to 4 cycle delay (3.4, 3.6, 3.8)
- Native interface protection via place holder (3.4, 3.6, 3.8)
- Transport protection on SMI interface, this can be either ***parity*** or ***ECC*** (SECDEC) (3.4, 3.6, 3.8)
- SRAM structures will use ***parity*** or ***ECC*** (SECDEC) for address and data (3.4, 3.6, 3.8)
- Internal protection within functional units for all data paths, pipelines and state machines (3.8)
- Memory like structures (large register files & Flop arrays), for example the ***OTT/ATT*** or skid buffers shall use parity (3.6, 3.8)
    - Pipelines, Queues and address Fifos shall use parity
    - State-machines shall detect illegal (non-reachable) states and preferably encode state information in a way to have all state encodings separated by a Hamming distance ≥ 2. One-hot encodings shall verify that only one bit is set
    - All CSR registers shall use parity protection

Any system configuration with ***ASIL*** level functional safety must instantiate a wrapper for each functional module. This wrapper shall include at least one active functional unit and a fault checker unit (with BIST support).

Up to Ncore 3.4, ***ASIL*** protection always required a duplicated functional unit, the fault checker performs a cycle-by-cycle comparison of the active and the checker unit. Full duplication (see Figure 2-2) may not always be required to achieve lower levels of protection (***ASIL-B/C***), with Ncore 3.6 area optimization is possible by performing FMEDA analysis on the full configuration and optionally remove duplicate units from the system configuration. In an optimized, partially duplicated configuration, non-duplicated units will also be encapsulated by a wrapper and use a modified fault checker – this fault checker does not require the cycle-by-cycle comparator unit (see Figure 2-3). Eventually, functional units in Ncore 3.8 will be equipped with an optional internal fault checking mechanism. Every unit must be designed to independently meet ASIL-B safety requirements – fault reporting will use an additional port on the fault checker module – starting with Ncore 3.6, the alternate fault checker module used to observe non-duplicated FUs shall provide a bundle of fault reporting inputs.



Wrapper configuration to support full duplication (3.4)

Wrapper configuration to support non-duplicated module. The inputs for a unit's internal fault reporting are tied off

Wrapper configuraton to support functional units with internal fault detection

FIGURE 2-1A TO 2-1C: NEW WRAPPER CONFIGURATIONS SUPPORTED

Figure 2-2 to Figure 2-4 show example configurations of Ncore systems, taking advantage of the new optimization.

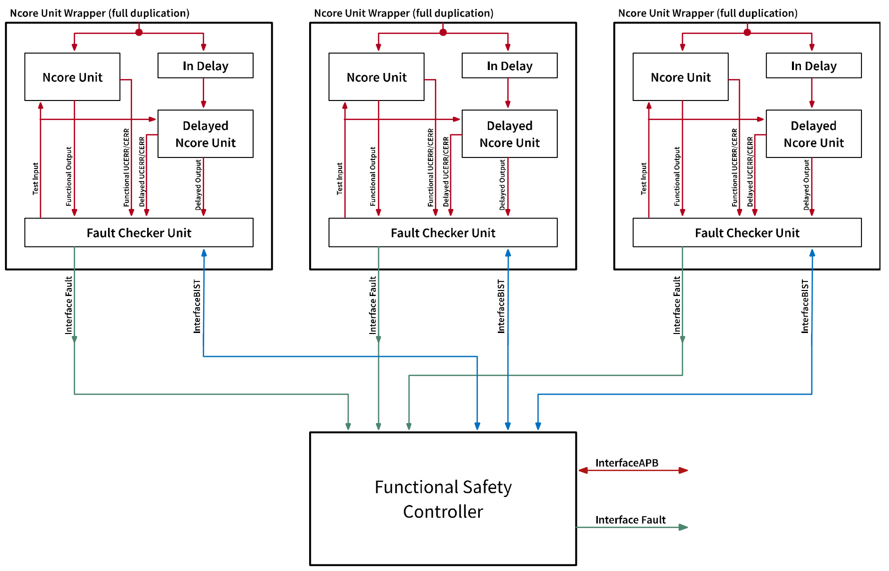


FIGURE 2-2: FULL DUPLICATION APPLIED

Figure 2-2 shows the current approach (Ncore 3.4), any system with ASIL protection at any level requires full duplication in all modules.

A new parameter ***protMode*** shall be defined for each unit to control the choice between: no-protection, unit-protection, full-duplication. If any ASIL level has been configured, any non duplicated unit shall protect internal registers by parity or ECC and implement BIST access ports to support testing the reporting infrastructure, also refer to Section 3.3 in this document.

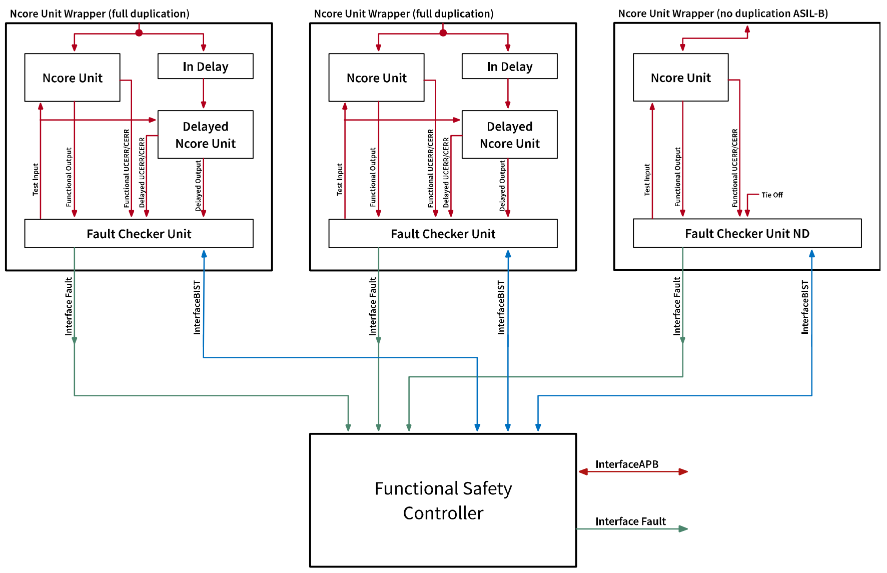


FIGURE 2-3: PARTIAL DUPLICATION APPLIED

Figure 2-3 shows an example where the rightmost Ncore unit is not duplicated because the system will comply with ASIL-B through full duplication of the remaining units. The fault checker used in this configuration will be significantly simplified and provide an optional bundle of additional fault detection inputs – these inputs may be tied off in Ncore 3.6 because the Ncore unit will only have limited fault reporting capability.

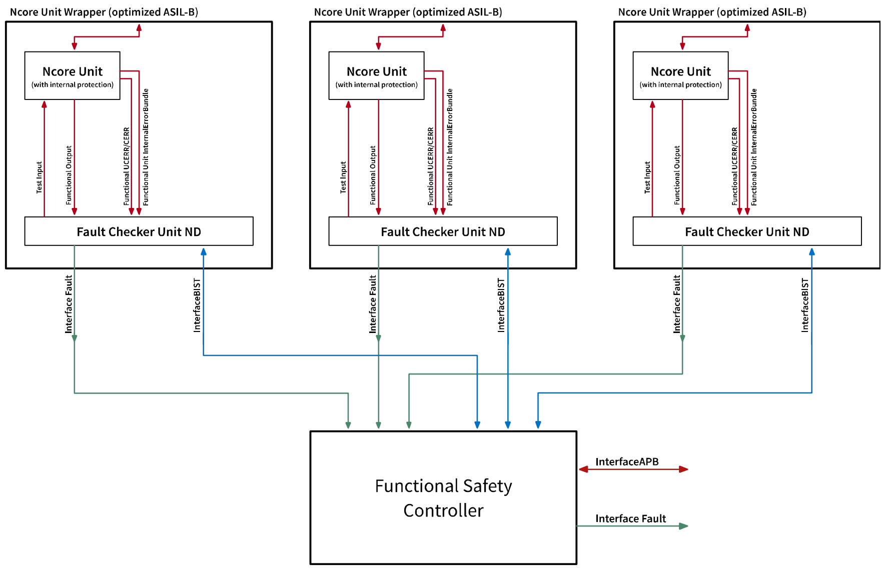

Ncore Unit Wrapper (optimized ASIL-B)
Ncore Unit (with internal protection)
Test Input
Functional Output
Functional UCERR/CERR
Functional Unit InternalErrorBundle
Fault Checker Unit ND
Interface Fault
InterfaceBIST
Functional Safety Controller
InterfaceAPB
Interface Fault

FIGURE 2-4: OPTIMIZED CONFIGURATON FOR ASIL-B

Figure 2-4 shows an example where each functional unit conforms to ***ASIL-B*** and therefore the higher system level only needs to provide the proper infrastructure to report detected faults and a ***BIST*** infrastructure to verify the integrity of said infrastructure. Each Ncore unit is accompanied by a fault checker to detect, aggregate and record the fault signals arriving from the local unit. A ***BIST*** controller will sequentially activate all implemented fault signals to verify the reporting infrastructure – local ***BIST*** controllers within the fault checkers will be triggered one-by-one from the centralized ***BIST*** manager within ***FSC***. Every time a local ***BIST*** sequence has been completed, a handshake in the ***interfaceBIST*** tells the central manager to proceed with the next module.

A detailed view of a complex Ncore unit with all Functional Safety features (for a duplicated module) is shown in Figure 2-5. Note that the SRAMs are not duplicated and must be protected with either ***ECC*** or ***parity***. This protection can be configured in Maestro, before RTL generation.

Functional units with a native interface port (e.g. ***CAIU***, ***NCAIU***) provide a place holder on the native interface where a customer may add his own protection logic to detect faults on the native protocol . This custom logic shall implement appropriate error detection for incoming transactions or add correction checksum (***ECC***, ***parity***) for the protected signals output by the Ncore unit. In addition it shall also implement mechanisms to verify and report faults based on the correctness of the applied checksum on the protected signals that are input to the Ncore unit. The inplemented logic must track the native interface protocol in signalling and timing.

The network protection logic adds either ***ECC*** or ***parity*** protection on communication with the ***CDTI***. The type of protection can be chosen during configuration before RTL generation. It adds the protection on outgoing ***SMI*** packets and checks incoming ***SMI*** packets for erros and report appropriate faults.

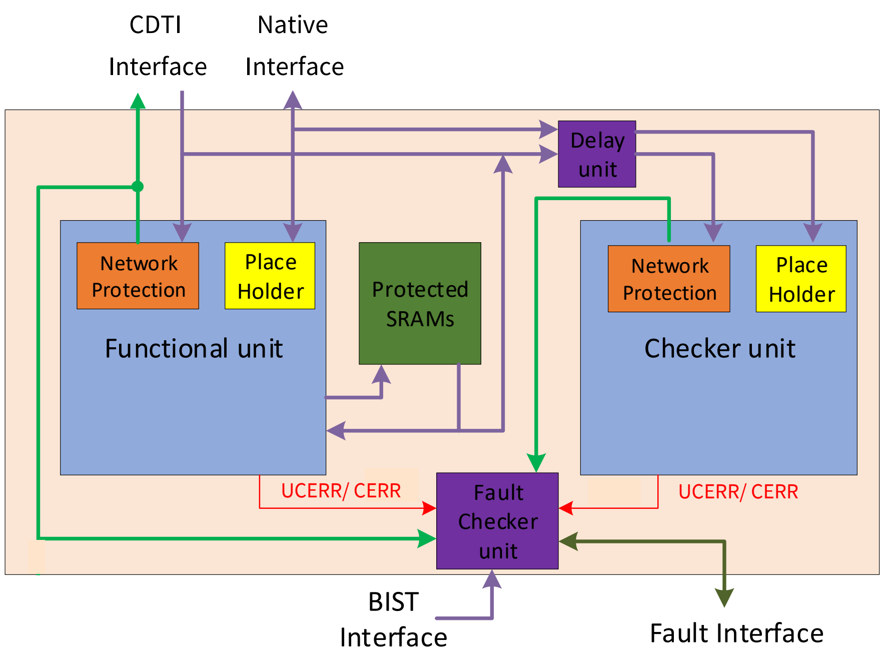


FIGURE 2-5: NCORE UNIT VIEW

3 Fault Checker

3.1 Architecture of fault checker used for duplicated units

Figure 3-1 shows the detailed structure of the current (3.2, 3.4) Fault Checker Unit used in systems with unit duplication. The main components include the (duplicated) comparator trees and the BIST FSM. The BIST FSM has been augmented with a timeout trigger output.

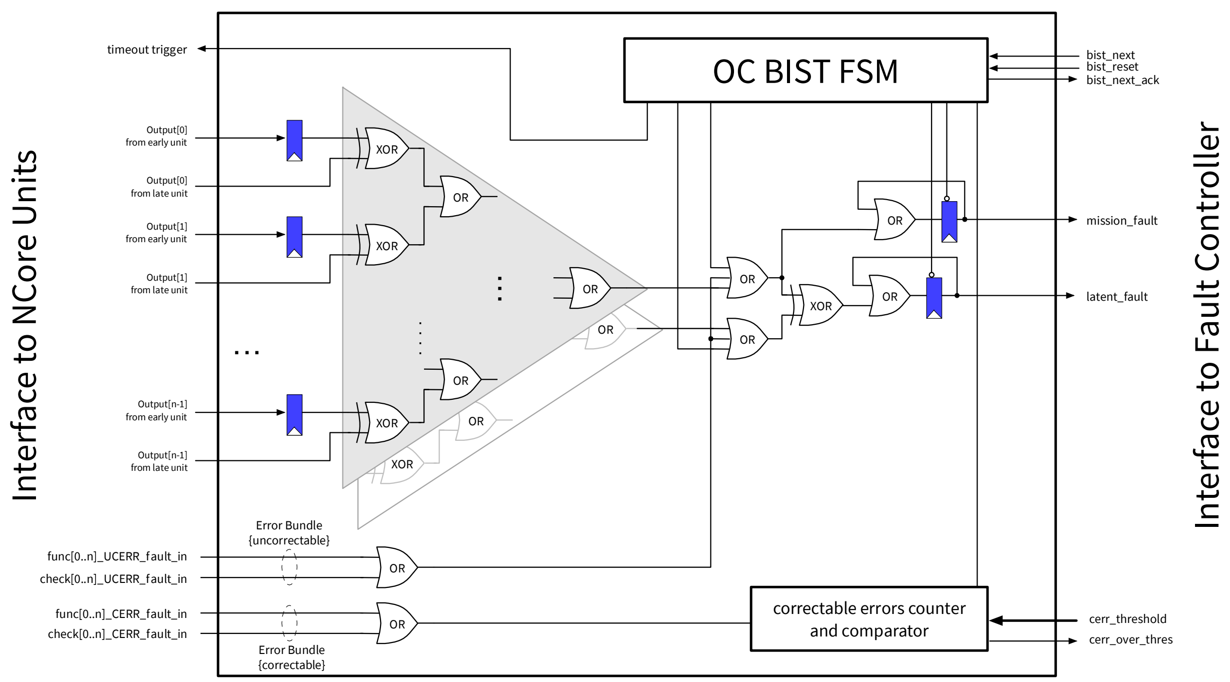

Interface to NCore Units
timeout trigger
Output[0] from early unit
Output[0] from late unit
Output[1] from early unit
Output[1] from late unit
Output[n-1] from early unit
Output[n-1] from late unit
XOR
OR
OC BIST FSM
bist_next
bist_reset
bist_next_ack
mission_fault
latent_fault
Interface to Fault Controller
Error Bundle {uncorrectable}
func[0..n]_UCERR_fault_in
check[0..n]_UCERR_fault_in
func[0..n]_CERR_fault_in
check[0..n]_CERR_fault_in
Error Bundle {correctable}
correctable errors counter and comparator
cerr_threshold
cerr_over_thres

FIGURE 3-1: FAULT CHECKER UNIT

The fault checker unit has 3 main functions.

1. Compare a "***late***" unit's delayed output to the functional block's output. If a discrepancy is detected, a fault will be signaled and then propagate through the fault path upstream. This feature is active when unit duplication is enabled.
    - Timeout event outputs from duplicate units will be part of the comparator input bundle and therefore the Fault Checker will detect it as a mismatch between functional and checker unit when only one unit times out.
    - The BIST controller shall implement a timeout trigger as a step within the self-testing sequence. This output signal shall be an input to both, the functional and the checker unit and, when activated, trigger a timeout signal within the timeout detection unit. The preferred implementation would preset the timeout counter to a value of (threshold – 1), followed by running the counter for a small number of clock cycles to exercise the counter, the comparator logic and the communication path to the Fault Checker.

2. This block also accumulates faults reported outside the check in function 1. Using ***numFaultInputs*** the checker creates ports to take in outside uncorrectable (***ucerr***) and correctable error (***cerr***) faults. The checker accumulates these faults in a sticky register bit and reports them upstream. The checker also implements a counter to count the number of ***cerr*** faults, the counter is compared to a programmable threshold (***cerr_threshold***) and a fault will be reporte upstream (to the fault controller) as soon as the threshold is reached.
3. The checker also has BIST functionality. A BIST finite state machine is implemented within the fault checker unit, its functionality is:
    - Reset the sticky bits set by detected faults or as part of the selftest sequence
    - Reset the error counter keeping track of correctable errors
    - Perform self-test sequences
    - The communication between the safety controller and the BIST FSM is asynchronous, it uses 2 wires, ***bist_next*** and ***bist_next_ack***. These 2 wires implement a full handshake. The BIST FSM has 5 states, these are sequentially executed, under control of the safety controller (after each handshake):
        - Reset fault D-FF and correctable error counter. The fault controller expects to see ***cerr_over_thres***, ***mission_fault*** and ***latent_fault*** return to 0.
        - Force the output of the functional comparator tree to 1. The fault controller expects to see ***mission_fault*** = 1 ***latent_fault*** = 1.
        - Reset the sticky fault D-FF, and force the output of the duplicated comparator tree to 1. The fault controller expects to see ***mission_fault*** = 0 ***latent_fault*** = 1.
        - Reset faults D-FF, and force both outputs of the functional and duplicated comparator trees to 1. The fault controller expects to see ***mission_fault*** = 1 ***latent_fault*** = 0.
        - Reset faults D-FF, and back to mission mode. The fault controller expects to see ***mission_fault*** = 0 ***latent_fault* =** 0.

This Fault Checker Unit architecture will continue to be used in Ncore 3.6 and 3.8 for any system that requires full duplication, independent if a single, a subset or all blocks will be duplicated.

3.2 Architecture of fault checker used for non-duplicated units

Figure 3-2 shows a proposed structure for a new (3.6 →) Fault Checker Unit used in systems without unit duplication.

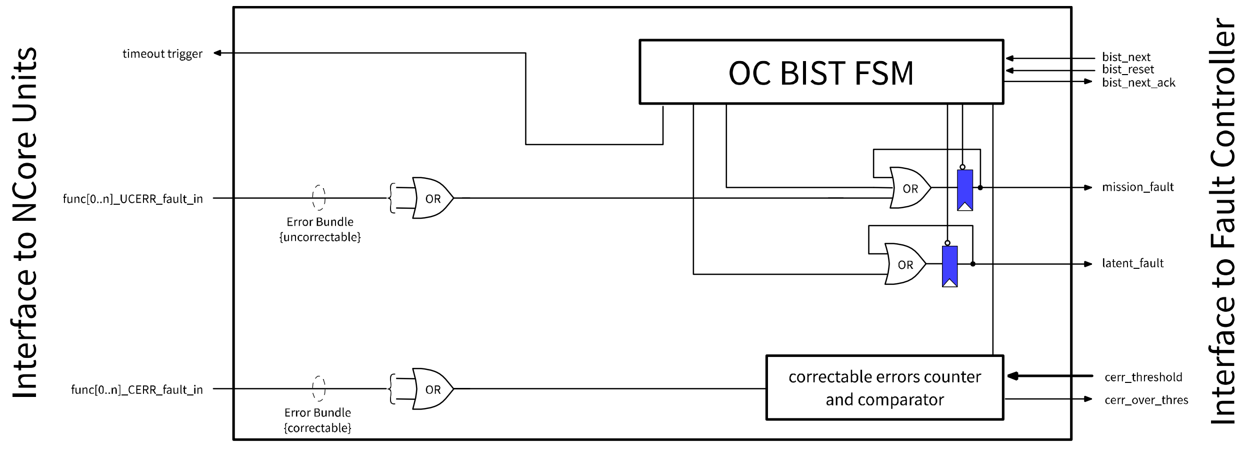


FIGURE 3-2: FAULT CHECKER ND UNIT

This checker is significantly simpler as it does no longer need to implement the comparator structures to check duplicated units against each other. The lack of a large comparator removes the requirement for duplication and therefore the ***latent_fault*** logic may be deleted and the output tied inactive. The BIST unit is the same as in the current Fault Checker units.

3.3 Detectedable internal fault conditions

Starting with Ncore 3.8 each functional unit shall include internal fault detection to make the unit fault resiliend. The goal shall be that each unit reaches the reliability level required by ASIL-B, i. e. ≥ 90 % of faults shall be detected.

The following type of faults shall be detected and signaled as part of the error bundle:

TABLE 3-1: UNIT ERRORS

Location	Error Type	Classification	Commment
Address path	Parity	Uncorrectable	Address paths, including internal queues, pipelines and FiFo structures shall use parity protection
Data path	Parity/ECC	Uncorrectable	Data paths, including internal queues, pipelines, FiFo structures and storage arrays (register files or SRAMs) shall use protection
OTT/ATT	Address Field	(Un-)correctable	Protected by parity
	Control Field		Protected by parity
	Linked List		Protected by parity
	Tracking State		Use resilient state encoding
	Timeout		Detect transactions that have seen starvation for a very long time; use a starvation counter with a very slow clock, add two extra bits to the entry, make this a maskable fault, optionally (CSR control) report as correctable, non-correctable, interrupt generation, ignore

Location	Error Type	Classification	Commment
Skid buffer	Overrun	Correctable	Skid buffer reached a threshold - optionally report as correctable, non-correctable, interrupt generation, ignore Threshold configurable to arbitrated range, FiFo half or fully occupied
	Backup	Uncorrectable	Skid buffer overrun with FiFo structure full and this state exists for ≥ 127 clocks
Protocol	Illegal command	Uncorrectable	Unrecognizable or unexpected command arrived
	Timeout	Uncorrectable	Protocol timeout – e.g. a response does not come back within a certain time
	Credit error	Uncorrectable	Native transition arrives at a unit that is not configured properly, e.g. has no credits allocated to a decoded target unit
Control	Illegal/Unreachable state in logic	Uncorrectable	Depends on the individual agent and logic block
Coherence	SysCoReq	Uncorrectable – maybe recoverable	coherent transaction arrives while unit is not participating in coherence protocol – check against SysCoReq FSM
CSR error	Parity	Uncorrectable	CSR parity check is thrown
Concerto Mux	TBD	uncorrectable	Parity error on internal queue or FiFo, any logic error

This list may not be exhaustive – depending on the individual units and their internal µarchitecture, detection of additional error may be possible and desirable. FMEDA and future updates to this document will be required during the µarchitecture and implementation phase. All of the above errors shall be setting a sticky bit in a status register – this status register shall be readable in a units CSR space.

4 Mission and Latent Fault Causes

The table 1 displays the Fault Checker Unit ***Mission_Fault*** and ***Latent_Fault*** outputs values based on the values of the Functional unit and Checker unit (delayed Ncore Unit) uncorrectable error signals values.
Any mismatch in output signals of the functional unit and checker unit excluding uncorrectable faults causes a mission fault.

TABLE 4-1: MISSION _FAULT AND LATENT_FAULT SIGNALS OUTPUTS VALUES

UCERR Functional	UCERR Checker	Mission_Fault	Latent_Fault
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

5 Functional Safety Controller

The FSC accumulates fault interfaces of multiple blocks which can be in multiple clock and power domains. It then reports any faults seen on these interfaces up stream and through registers. The FSC in addition can perform BIST on the Resiliency reporting logic of each of the blocks. Figure 5-1 shows the detailed block diagram of FSC.

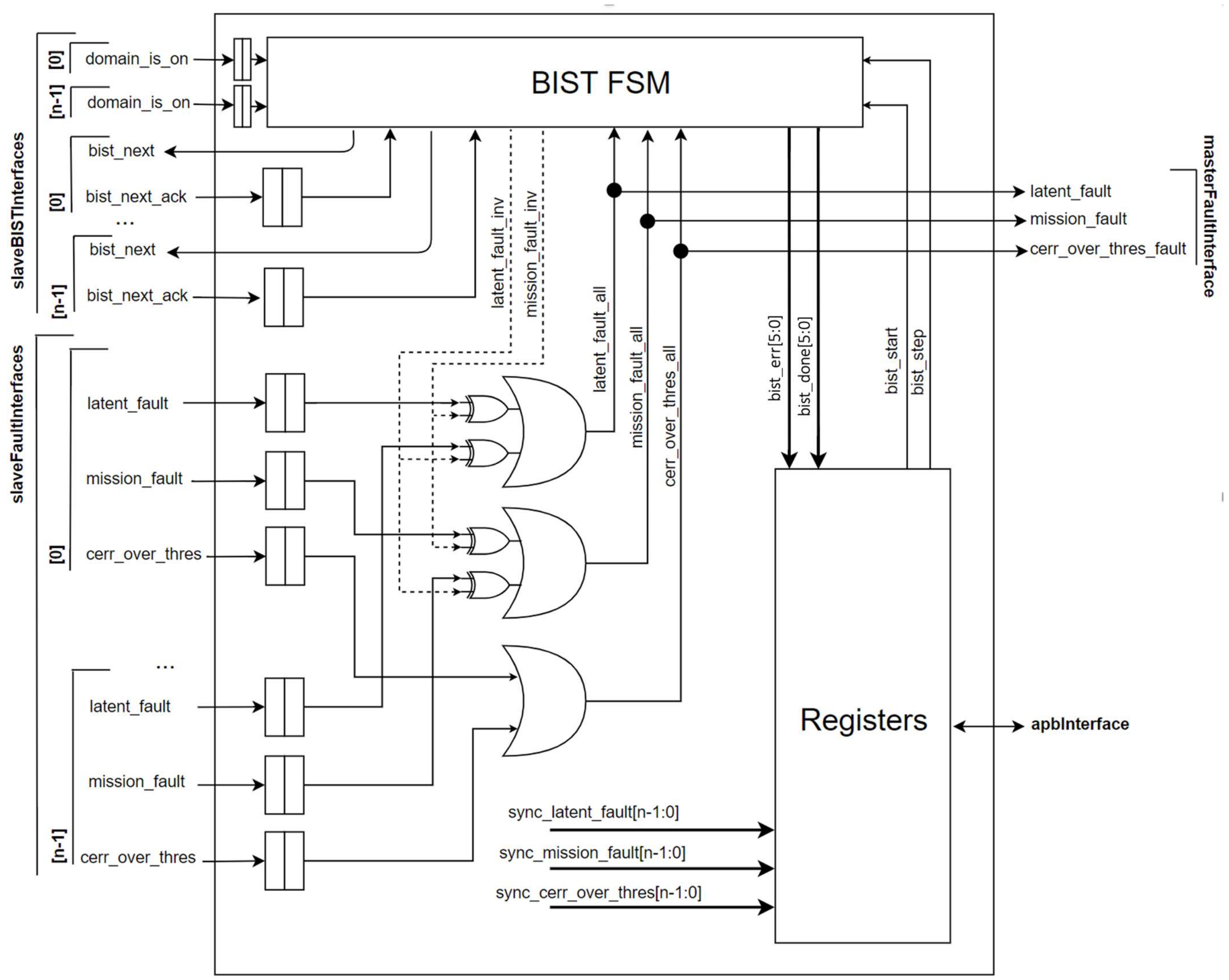


Figure 5-1: FSC top level block diagram

5.1 Feature List

FSC supports the following features

- Accumulates faults and reports them upstream and in registers.
- Performs BIST of Resiliency reporting logic of connected blocks.
- Configurable through an APB Interface.
- Can connect to blocks in multiple power and clock domains.
- All internal registers shall be protected by parity

5.2 BIST FSM

A BIST FSM can control BIST operations in each output checker connected to the safety controller, under software control, through access to the safety controller registers.

Communication with the BIST FSM in each output checker is done through a full handshake mechanism: the safety controller asserts bist_next to indicate that the checkers FSM need to execute the next step of the BIST sequence. The checkers FSM assert in turn bist_next_ack. When all bist_next_ack from all checkers are asserted, the safety controller de-asserts bist_next. This deassertion is the event that will make the checkers FSM execute the step of the sequence. After the checkers are done executing that step, they de-assert bist_next_ack. When the safety controller sees all bist_next_ack de-asserted, it checks for the BIST step result.

Two modes of operation are supported: automatic and step-by-step. In automatic mode, the software writes a bit in a register to start the BIST sequence, and all five steps listed below are sequenced automatically in every output checkers in parallel. At the end of the sequence of five steps, a register reports when the sequence is complete, and another register reports a success/failure bit per step. The sequence will always complete whether any specific step has an error or not.

In step-by-step mode, the software writes into a register for each BIST step, one by one. This mode allows for accessing more detailed information in case a step is reporting an error in automatic mode. Every time the software request execution of the next step, the BIST FSM send a request to execute the next BIST step to all output checkers. A register reports when the sequence is complete, and through access to the details of the mission_fault and latent_fault inputs it is possible to identify the source of a failure.

During BIST, the BIST FSM can invert all the faults inputs, to detect that all inputs are 0, or all inputs are 1, using the same OR tree, and per the step in the BIST sequence.

The BIST State Machine states and associated detailed information are described in the Table 5-1

State	Description	State Outputs	Conditions for Next State
0 (Idle)	Idle State.	bist_done[5:0] = {1,1,1,1,1,1}	(bist_start | bist_step) **Set internal flag:** bist_run <= bist_mode & bist_start

1 (Check_1)	The fault controller expects to see all cerr_over_thres, mission_fault and latent_fault go to 0.	bist_done[5:0] = {0,0,0,0,0,0}
		latent_fault_inv = 0
		mission_fault_inv = 0
			~bist_next &bist_all_de_ack & (bist_start | bist_run)

2 (Check_2)	The fault controller expects to see all mission_fault = 1 and all latent_fault = 1.	bist_done[5:0] = {0,0,0,0,0,1}
		latent_fault_inv = 1
		mission_fault_inv = 1
			~bist_next &bist_all_de_ack & (bist_start | bist_run)

3 (Check_3)	The fault controller expects to see all mission_fault = 0 and all latent_fault = 1	bist_done[5:0] = {0,0,0,0,1,1}
		latent_fault_inv = 1
		mission_fault_inv = 0
			~bist_next &bist_all_de_ack & (bist_start | bist_run)

4 (Check_4)	The fault controller expects to see all mission_fault = 1 and all latent_fault = 0	bist_done[5:0] = {0,0,0,1,1,1}
		latent_fault_inv = 0
		mission_fault_inv = 1
			~bist_next &bist_all_de_ack & (bist_start | bist_run)

5 (Check_5)	The fault controller expects to see all mission_fault =1 and all latent_fault = 0	bist_done[5:0] = {0,0,1,1,1,1}
		latent_fault_inv = 0
		mission_fault_inv = 1
			~bist_next &bist_all_de_ack & (bist_start | bist_run)

6 (Check_6)	The fault controller expects to see all mission_fault = 0 and all latent_fault = 0	bist_done[5:0] = {0,1,1,1,1,1}
		latent_fault_inv = 0
		mission_fault_inv = 0
			~bist_next & bist_all_de_ack & (bist_start | bist_run)
			
			Clear internal flag:
			bist_run <= 0

Table 5-1: BIST FSM

Note: During BIST in automatic mode the behavior of interrupt signals is undefined. However, in manual mode the interrupt signals are valid after the respective steps bist_done bit is set.

5.3 Registers

Register Name	Register Offset	Description
SCBISTCR	0x000	Safety Controller BIST control register
SCBISTAR	0x004	Safety Controller BIST activity register
SCLF0	0x010	Latent Faults inputs values 31:0
SCLF1	0x014	Latent Faults input values 63:32
SCLF2	0x018	Latent Faults input values 95:64
SCLF3	0x01C	Latent Faults input values 127:96
SCMF0	0x020	Mission Faults inputs values 31:0
SCMF1	0x024	Mission Faults input values 63:32
SCMF2	0x028	Mission Faults input values 95:64
SCMF3	0x02C	Mission Faults input values 127:96
SCCETHF0	0x030	Correctable errors counter above threshold (CECAT) inputs values 31:0
SCCETHF1	0x034	Correctable errors counter above threshold input values 63:32
SCCETHF2	0x038	Correctable errors counter above threshold input values 95:64
SCCETHF3	0x03C	Correctable errors counter above threshold input values 127:96

Table 5-2: Register Overview

The FSC supports up to 128 error inputs for each fault category. These signals are generated by the ***Fault Checker Units*** within each wrapper, see Figure 3-1 and Figure 3-2. Ncore configurations may vary widely in the number of supported FUnits; to simplify fault analysis, Table 5-3 defines an ***architected*** mapping between **bit location** in any of the 4 fault status registers and the FUnit it is associated with.

Functional Unit Type	Error Input Index	Description	Register
DVE	127	There will be only one DVE in the system	SCCETHF3, SCFM3, SCLF3[31]
reserved	120 .. 126	No unit connected – tie off to 0	
DMI	112 .. 119	{Latent, Mission, CECAT}-input [119:112]	SCCETHF3, SCFM3, SCLF3[(num_DMI+15):16][1]
DCE	96 .. 111	{Latent, Mission, CECAT}-input [111:96]	SCCETHF3, SCFM3, SCLF3[(num_DCE-1):0][2]
DII	64 .. 95	{Latent, Mission, CECAT}-input [64:95]	SCCETHF2, SCFM2, SCLF2[(num_DII-1):0][3]
IOAIU/NCAIU	32 .. 63	{Latent, Mission, CECAT}-input [63:32]	SCCETHF1, SCFM1, SCLF1[(num_IOAIU-1):0][4]
CAIU	0 .. 31	{Latent, Mission, CECAT}-input [31:0]	SCCETHF0, SCFM0, SCLF0[(num_CAIU-1):0]

Table 5-3: Register Overview

Configuration software (Maestro) shall use the enumerated ID for each FU (within its class) to connect the fault outputs. For example, DCE0's fault signals shall be connected to the fault signal #96 within the 128-bit wide bundle

[1] Ncore supports only 8 DMI in the system – DMI are sharing registers {SCCETHF, SCFM, SCLF} [3] with DCE and DVE
[2] Supporting up to 16 DCE
[3] Up to 32 DII – it is unlikely that we will exhaust this number in Ncore 3.6/3.8
[4] Even though Ncore 3.6 and 3.8 will not support more than 64 AIUs, we reserve two registers – Coherent AIUs' status will be reflected in registers {SCCETHF, SCFM, SCLF} [0], non-coherent AIUs' will be mapped to {SCCETHF, SCFM, SCLF} [1]

representing the Latent-, Mission- and Error-Threshold-Fault. Every 32-bit register will be associated with a single unit type, except register #3 in each group which will collect DCE and DVE fault status. Any unused bits in these registers shall read back as 0.

5.3.1 SCLFX Register Bit Assignment

Bit	Name	Description	Access	Reset
N-1:0	latent_fault	Each bit indicates a latent fault on the corresponding fault interface in slaveInterfaces	RO	0x0

Table 5-4:SCLFX Register Bit Assignment

5.3.2 SCMFX Register Bit Assignment

Bit	Name	Description	Access	Reset
N-1:0	mission_fault	Each bit indicates a mission fault on the corresponding fault interface in slaveInterfaces	RO	0x0

Table 5-5: SCMFX Register Bit Assignment

5.3.3 SCCETHFX Register Bit Assignment

Bit	Name	Description	Access	Reset
N-1:0	cerr_over_thresh	Each bit indicates the amount of correctible errors went over the threshold on the corresponding fault interface in slaveInterfaces	RO	0x0

Table 5-6: SCCETHFX Register Bit Assignment

5.3.4 SCBISTCR Register Bit Assignment

SCBISTCR is a read/ write register, a read access will always return the last value written.

Bit	Name	Description	Access	Reset
31:2	Reserved	Reserved	RO	0x0
1	bist_mode	1: Sequential mode (BIST FSM runs all 6 BIST steps in sequence) 0: Single step mode	R/W	0x0
0	bist_start	Writing a 1 will start the engine, depending on bist_mode the engine will perform a single step or run to completion. If there is an error, BIST will restart the sequence when **bist_start** is written to again.	R/W	0x0

Table 5-7: SCBISTCR Bit Assignment

One write operation is required to run the BIST; all 4 combinations having a defined behavior and 6 writes are needed to complete the BIST in step mode. While the BIST is running, any write shall be disregarded.

BIST can be 1st run in step mode for 1 or more steps and then BIST run can be completed in automatic mode or be reset by writing “0” in “bist_start”. “bist_done” in the FSC BIST Activity Register shall be reset at the same time.

A parity bit shall be added to the BIST Control and Activity registers for the application to be able to discriminate between hardware and software errors.

5.3.5 SCBISTAR Register Bit Assignment

Bit	Name	Description	Access	Reset
31:12	Reserved	Reserved	RO	0x0
11:6	bist_err	Each bit in this register represents one step of the BIST sequence. When the sequence starts, it's reset. As each BIST step completes, its bit is set to 1 if the sequence has completed in error **Bit** – **Field** **6** – BIST full reset error (Step1) **7** – BIST functional comparator tree force error (Step2) **8** – BIST duplicate comparator tree force error (Step3) **9** – BIST both comparator tree force error (Step4) **10** – BIST timeout test error (Step5) **11** – BIST final full reset error (Step6)	RO	0x0
5:0	bist_done	Each bit in this register represent on step of the BIST sequence. When the sequence starts, it's reset. As each BIST step completes, its bit is set to 1 **Bit** – **Field** **0** – BIST full reset done (Step1) **1** – BIST functional comparator tree force done (Step2) **2** – BIST duplicate comparator tree force done (Step3) **3** – BIST both comparator tree force done (Step4) **4** – BIST timeout test done (Step5) **5** – BIST final full reset done (Step6)	RO	0x0

Table 5-8: SCBISTAR Bit Assignment

The expected value of the "bist_done" in FSC BIST Activity Register shall always be defined. This ensures software can take appropriate action: "0" after reset and after any write operation, report the value corresponding to the step completed.