TriCore™ AURIX™ 家族系列

32 位

采用 MTU(内存测试单元）的 AURIX™ 内存检测

AP32197

应用笔记

V1.0 2014-05

微控器

免责条款

为方便客户浏览，英飞凌以下所提供的将是有关英飞凌产品及服务资料的中文翻译版本。该中文翻译版本仅供参考，并不可作为任何论点之依据。虽然我们尽力提供与英文版本含义一样清楚的中文翻译版本，但因语言翻译和转换过程中的差异，可能存在不尽相同之处。因此，我们同时提供该中文翻译版本的英文版本供您阅读，请参见 www.infineon-ecosystem.org。并且，我们在此提醒客户，针对同样的英飞凌产品及服务，我们提供更加丰富和详细的英文资料可供客户参考使用。请详见 www.infineon.com

客户理解并且同意，英飞凌毋须为任何人士由于其在翻译原来的英文版本成为该等中文翻译版本的过程中可能存在的任何不完整或者不准确而产生的全部或者部分、任何直接或者间接损失或损害负责。英飞凌对于中文翻译版本之完整与正确性不担负任何责任。英文版本与中文翻译版本之间若有任何歧异，以英文版本为准，且仅认可英文版本为正式文件。

您如果使用以下提供的资料，则说明您同意并将遵循上述说明。如果您不同意上述说明，请不要使用本资料。

版本 2014/05

出版发行：
英飞凌科技公司
上海，中国

© 2014 Infineon Technologies
版权所有

免责条款

本应用笔记中给出的信息仅作为实现英飞凌器件的建议，不得被视为英飞凌器件的任何特定功能、条件或质量作出的任何说明或保证。此应用笔记的接受者必须在实际应用中判定此种描述的任何功能。英飞凌科技在此否认承担此应用笔记中任何和所有信息相关的任何形式的保证和责任（包括但不限于不侵犯第三方知识产权）。

信息

有关技术、交货条款及条件和价格，请与您最近的 Infineon Technologies 办事处联系。

警告

由于技术要求，组件可能含有危险物质。如需相关型号的信息，请与您最近的 Infineon Technologies 办事处联系。如果可能合理地预期此类组件的故障会导致生命支持器件或系统发生故障或影响该器件或系统的安全性或有效性，则 Infineon Technologies 提供的组件仅可用于获得 Infineon Technologies 明确书面批准的生命支持器件或系统。生命支持器件或系统的目的是植入人体或支持和/或保持并维持和/或保护生命。如果出现故障，则可能危及使用者或他人的人身安全。

文献修订史

日期	版次	修订人	修订内容
2012/10/17	V1.0	Tomislav Garaca	初版

商标：

Infineon®, TriCore™ and AURIX™ are registered trademarks of Infineon Technologies AG.

请留下您的宝贵建议

您是否认为本文档中的任何信息存在错误，含糊不清或遗漏？您的宝贵意见和建议将帮助我们持续不断地改进文档质量。请将您的建议（请注明文档的索引号）发送电子邮件至：

ctdd@infineon.com

目录

1 导言 5
1.1 对读者说 5
1.2 范围和目的 5
1.3 缩略词 5
1.4 参考文献 5

2 内存测试单元 (MTU) 6
2.1 概况 6
2.2 MBIST(存储器单元内部建自测) 6
2.3 存储器控制框图 7
2.4 直接 CPU 接口 8
2.5 存储器控制寄存器 8
2.6 寄存器映射 9

3 内存错误 10
3.1 概况 10
3.2 内存错误分类 10
3.2.1 静态和动态错误 11
3.2.2 简单错误和连锁错误 11
3.2.3 单个单元错误和多个单元错误 11

4 内存测试算法 12
4.1 概览和术语 12
4.2 测试算法的选择 12
4.3 算法测试参数 14
4.3.1 寻址方向 14
4.3.1.1 快速的列寻址 14
4.3.1.2 快速的行寻址 15
4.3.2 寻址方式 16
4.3.3 后台数据 16
4.4 测试算法的有效性 17

5 执行和用途 18
5.1.1 启动内存测试时序 18
5.2 MTU 执行的算法 20
5.2.1 非破坏的倒置测试 20
5.2.2 Checkerboard 测试 22
5.2.3 March-U 测试 24

6 附录：使能/停用存储控制器 26
6.1 使能存储控制器 26
6.2 停用存储控制器 27

1 导言

该应用笔记描述了如何使用内存测试单元(MTU)去测试英飞凌 32 位微控器 AURIX™ 家族的内存。

1.1 对读者说

此文件为应用软件工程师撰写英飞凌微控器 AURIX™家族系列而编制。

1.2 范围和目的

该文件涵盖了在微控器 AURIX™家族内进行内存测试所需的 MTU 硬件和软件架构的特征。

1.3 缩略词

本文档使用了以下缩略词：

BIST	内部自检
BPI	总线外设接口
ECC	错误纠正码
MBIST	存储器内部自检
MTU	存储器测试单元
SCU	系统控制单元
SPB	系统外围总线

1.4 参考文献

额外的参考文档:

[1] AURIX™ TC27x User Manual V1.1 2012-07, Infineon Technologies (or later)

2 内存测试单元(MTU)

该章节简要概述了 MTU,存储控制器的框图,MBIST/ECC 总线接口,存储控制器寄存器,以及 MBIST/ECC 寄存器映射方案。

2.1 概况

AURIX™家族微控器的内存有一个统一的接口用于控制 BIST（内部自检），ECC(错误纠正码）和冗余特征。该接口也可被用于将预定义的数据值存储到存储器以便初始化。MTU 控制并监控这些内存测试，初始化，和数据完整性检查功能。

MTU 包含了以下两部分：

- MTU 配置寄存器
  - 用于使能并停用内存测试功能
- 与存储单元控制器相关的寄存器
  - 这些寄存器被器件里的每个存储模块重复使用，并控制着特定存储模块各自的 MBIST,ECC 和冗余设置。

有关 MTU 配置寄存器和存储控制器寄存器更多详情，请参考 AURIX™ TC27x 用户手册[1]。

2.2 MBIST(存储器单元内部自检）

基于硬件的 MBIST 被实现为连接到每个内存块的额外逻辑，它控制着存储单元上的算法测试序列。MBIST 包含了一组寄存器，一个或多个功能状态机（FSM)。寄存器被用于配置 MBIST,而测试序列则由 FSM 执行。使用 MBIST 的优势就是无需 CPU 就可以直接访问存储器。即使那些通过 CPU 都无法直接访问的很小的储存器都可以通过 MBIST 进行测试。

以下是连接了一些 SRAM 和 MBIST 的被测器件（DUT)图：

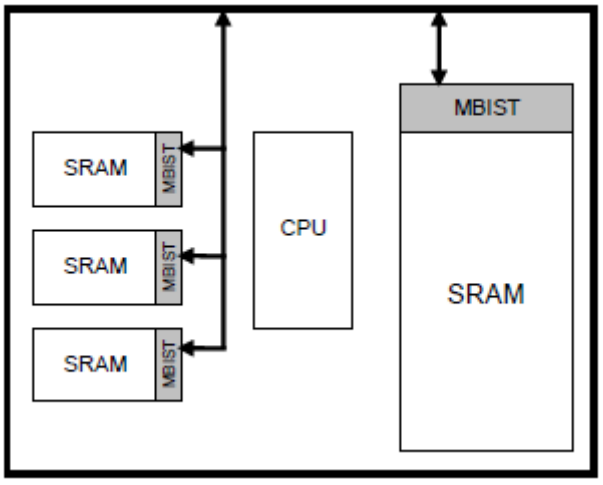

图 1 使用 MBIST 的被测器件 DUT

每一个 SRAM 都连接了一个 MBIST。而每一个 MBIST 都被连接到了 DUT 的外围，以使能测试参数的配置并读取测试结果。MBIST 测试未使用 CPU。

2.3 存储器控制框图

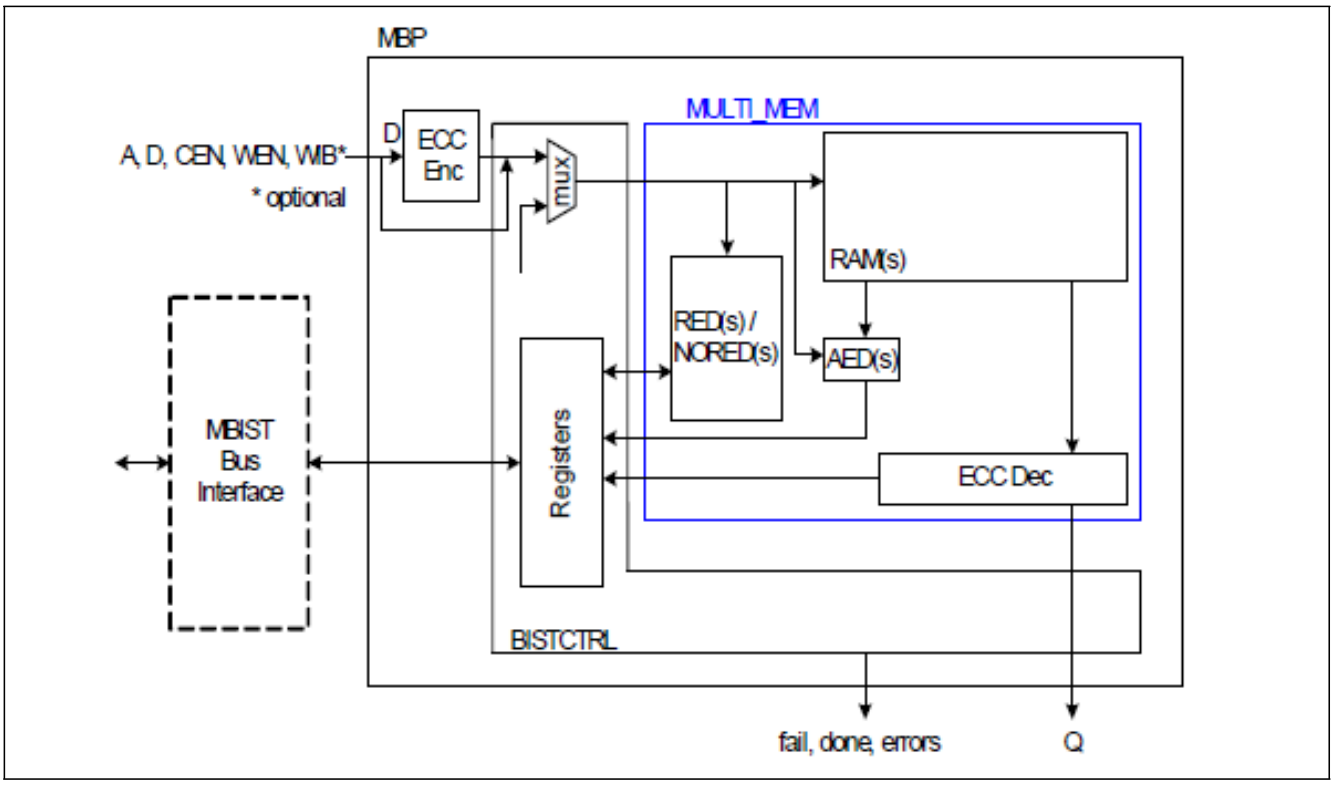

图 2 所有 MBIST/ECC 功能块(单口 SRAM)框图

关键单元:

- MBP:MBIST/ECC 控制器,RAM 和冗余存储单元的封装
- RAM(s):被测试的单个/多个内存单元
- RED(s):包含冗余存储单元的单个/多个冗余封装,用于替换缺陷的 RAM 单元。
- NORED(S):如果冗余单元数为 0，单个/多个空的冗余封装。
- BISTCTRL:MBIST/ECC 控制器
- MULTI_MEM:在组件配置里，用于示例多个 RAM 和 RED/NORED 的包装。
- ECC:根据代码执行错误纠正和解码的 ECC 编码器和解码器
- AED:地址错误检测逻辑

2.4 直接 CPU 接口

到 MBIST/ECC 控制器寄存器有一个专用的接口。可以经由一条双向总线（逻辑上为 16 位宽）访问所有的寄存器。

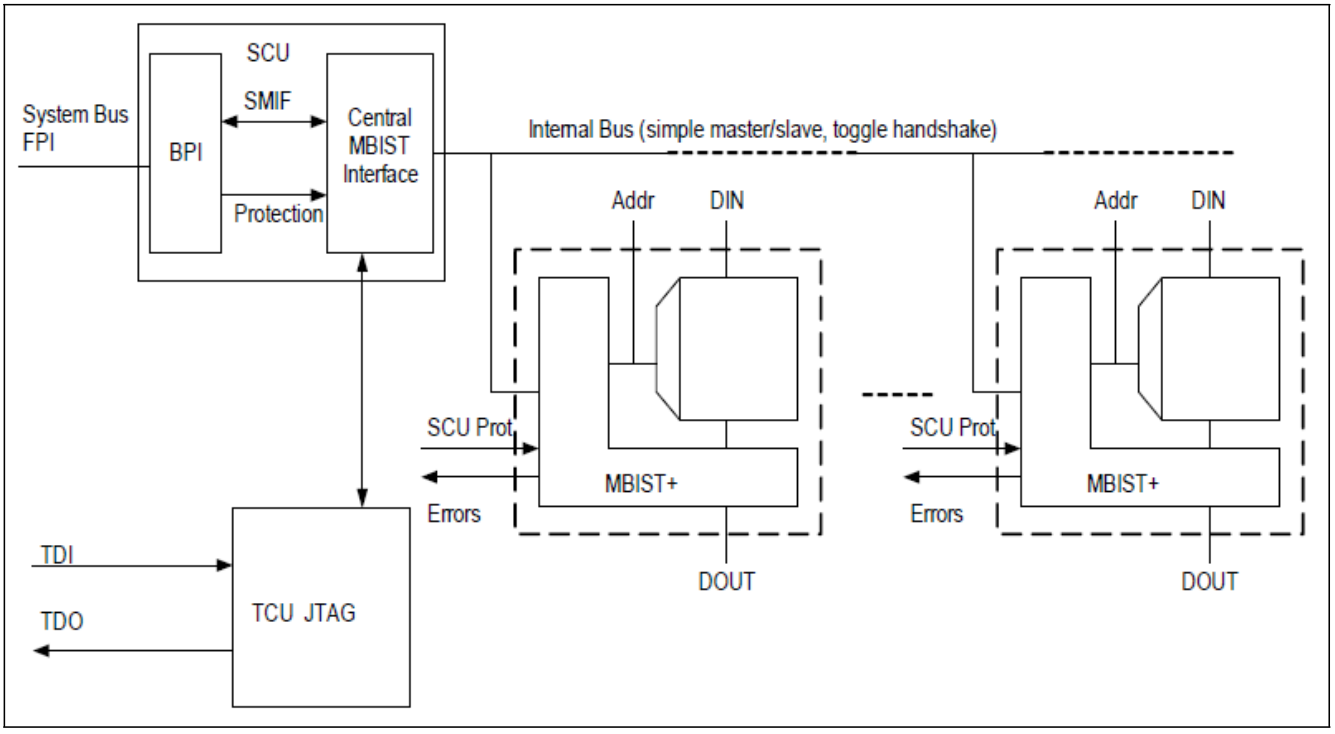

图 3 MBIST/ECC 总线接口

2.5 存储器控制寄存器

本章节简要描述了 MTU 存储器控制寄存器。

注释：有关寄存器的详细描述，请参考 AURIX™ TC27x 用户手册[1].

有了这个存储器控制寄存器，用户就可以执行表 1 中所给出的测试算法，或者是创造他们自己的测试算法。同时，还可以控制测试行为并读出测试结果。

配置寄存器：CONFIG0 和 CONFIG1

MBIST 配置寄存器 CONFIG0 和 CONFIG1 可被用于控制并设置任意定向测试算法。所有有关测试序列，读写操作的位都集中在此。在试运行期间，这两个寄存器里的位都不会改变。

MBIST 控制寄存器：MCONTROL

控制寄存器， MCONTROL 里的位被用于控制 MBSIT 行为并定义需被执行的测试。当 MCONTROL.START 位置位时，开始并执行任意测试。所有的测试都是不可中断的。

MBIST 状态寄存器：MSTATUS

状态寄存器里的位分别显示了当前正在运行的测试状态,以及被执行的最后一次测试。当前运行的测试的结束在 MSTATUS.DONE 位里以信号标志出。

范围寄存器：RANGE

范围寄存器可被用于在 RAM 的专用区进行测试。范围可被设置为 64 字步长。

注意：　由于实现 RANGE 定义所用的地址加扰算法复杂，因此不打算将 RANGE 作为一个主要特征。对英飞凌 AURIX™ 器件内存的架构和配置都需达到专家级的水平，以便合理地定义部分内存测试或部分内存初始化。因此强烈建议不要试图配置 RANGE 寄存器，并使用默认值，因为他们使能了由某个特定储存控制器控制的整个内存块的测试。

2.6 寄存器映射

物理上来讲，所有的寄存器都位于 MBIST / ECC 功能块。逻辑上来讲，他们都具备一个 SCU(系统控制单元）地址，并可通过正常的 16 位 SPB 总线访问进行访问。

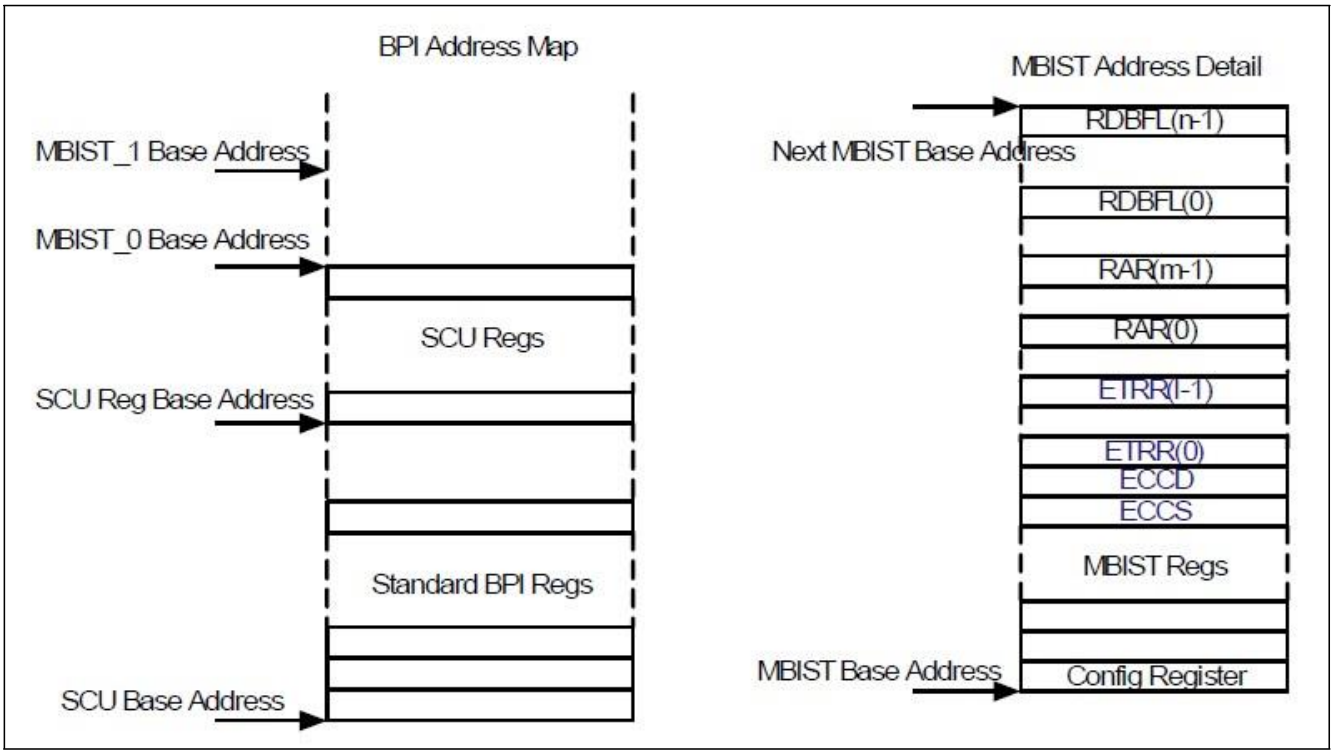

图 4　MBIST / ECC 寄存器映射方案

3 内存错误

本章节简要描述了内存错误以及他们的分类。

3.1 概况

内存错误往往是电子器件故障行为的起因。存储到内存或从内存读取的不正确的信息可以造成整个系统的故障。物理缺陷是内存错误的原因之一，但该缺陷在正常情况下无法目测发现。仅仅只有错误行为被视为功能故障。

3.2 内存错误分类

在内存里面会发生很多类型的错误，如存储单元阵列错误，地址译码错误，和外围电路错误等。

下图是内存错误分类一览：

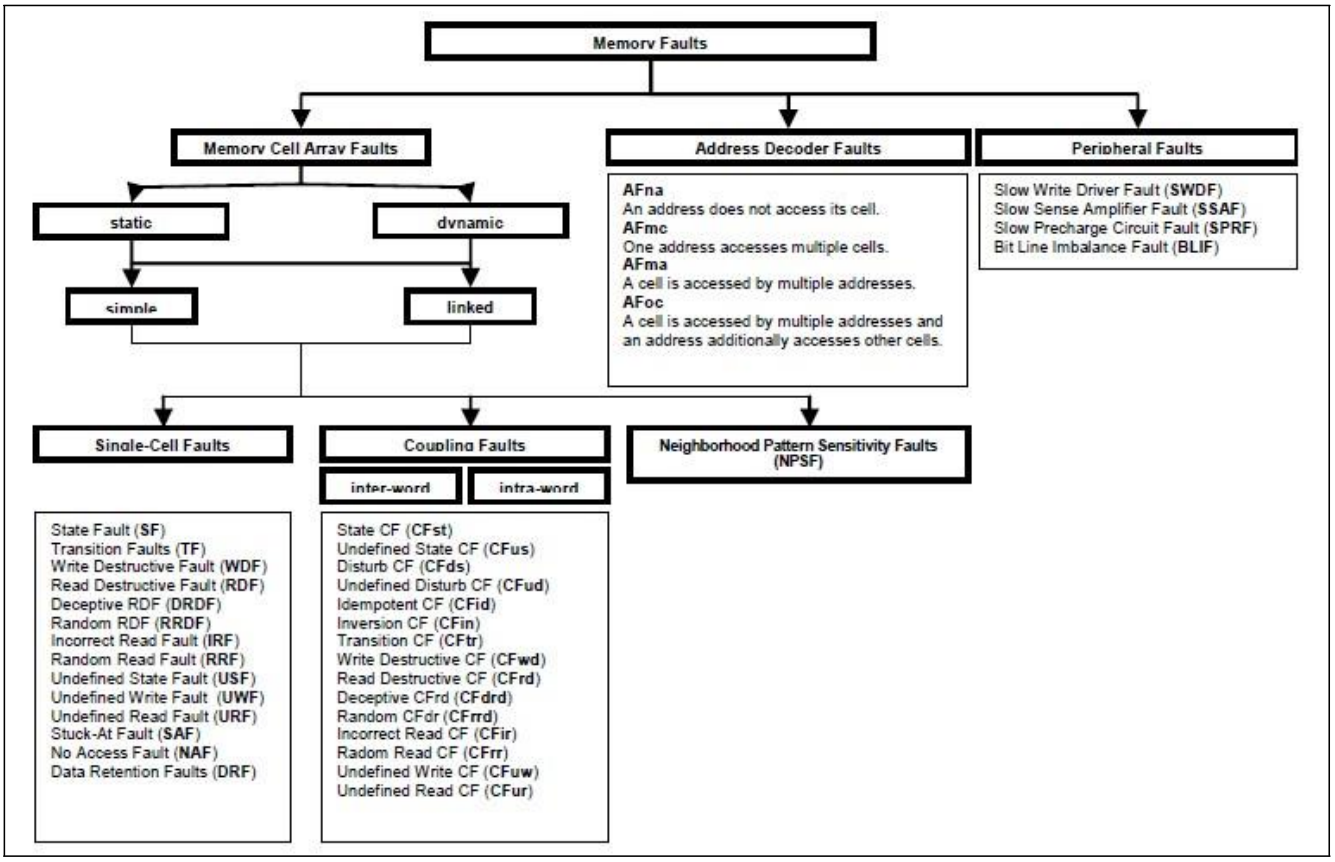

图 5 内存错误分类

3.2.1 静态和动态错误

有一些错误仅仅发生在一定的应力条件下,因此根据所执行的顺序操作的数量,可将错误分为静态错误和动态错误。

•静态错误是指那些最多一次操作就可被激活的操作
•动态错误需要多次操作才可被激活

3.2.2 简单错误和连锁错误

根据一个错误的表现形式，可将它分为“简单”或“连锁”错误。

•简单错误不会影响其它错误的表现
•连锁错误会影响其它错误的表现并可能发生掩蔽

3.2.3 单个单元错误和多个单元错误

有一些错误只有当相邻的两个单元在某种状态时才可被检测出来。

根据所涉及的不同单元的数量，可将错误分为单个单元错误和多个单元错误（耦合错误和邻里方式敏感性错误）。

•如果在应用的顺序操作里仅仅涉及了单个单元，则发生单个单元错误
•如果在顺序操作里涉及了两个或多个单元，则发生耦合错误

因此，如果错误发生在应用顺序操作的同一个单元，则发生了单个单元错误。如果检测到错误的单元与错误出现的单元不一致，则发生了耦合错误。

4 内存测试算法

本章节解释了内存测试算法的定义和结构,以及与这些算法结合使用的一些算法测试参数。

4.1 概览和术语

必须定义激活和检测故障的操作时序（如读写操作时序）以用于算法的内存测试。因此开发了一个开放的符号，以便在测试算法里描述操作时序。

测试算法包含了一系列前进单元。在进入下一个地址之前，一个前进单元的读写操作被按照顺序应用到一个存储单元，并且在进入下一个前进单元之前，一个前进单元被应用到所有的内存地址。

测试算法由波形括号{ }分隔，且每一个前进单元由括号（）分隔。前进单元由分号分隔，且在每个前进单元内的单个读写操作由逗号分隔。

跨步单元之前的符号⇧,⇩或⇳分别指出了向上寻址方向（最低地址到最高地址），向下寻址方向（最高地址到最低地址）或任意寻址方向。

应用到一个单元的操作可以是“w0”(write 0), “r0” (read 0), “w1” (write 1) 或 “r1” (read 1)。

例如：SCAN 测试：{⇧(w0);⇧(r0);⇩(w1);⇩(r1)}
•第一个前进单元：指增加的地址序列：非反相写模式
•第二个前进单元：指增加的地址序列：非反相读模式
•第三个前进单元：指递减的地址序列：非反相写模式
•第四个前进单元：指递减的地址序列：非反相读模式

该算法访问每个内存地址 4 次，因此测试长度是 TL=4n。

4.2 测试算法的选择

在 20 世纪 70,80 年代，测试算法还处于初始阶段，仅仅定义了少数用于单个单元和静态错误的简单错误模型，因此只开发使用了少量简单的测试算法。随着更多不同类型及更复杂错误的出现，如连锁错误和动态错误，开发了一些新的测试算法。今天，已经发布了很多的内存测试算法，并且能从广泛的算法清单中调阅最流行的算法子集。

对于寻址序列，可分为两种类型的测试算法：

•一维算法
  -一个单元接一个单元访问（行进算法）
•多维算法
  -单元阵列的“阶跃”

英飞凌 MBIST 设计，被称之为 MBISTPLUS，被用于汽车微控器器件。MBISTPLUS 是一个可配置的 BIST，容许用户编写一些广泛使用而又广为人知的算法，但同时也允许用户编写他们各自的算法跨步测试并选择算法测试参数。可用寄存器配置测试设置并储存测试结果。

从可获得的大量测试算法中，选取了 MBISTPLUS 所支持的 30 个一维跨步算法，并显示在表 1 中。

表 1 内存测试算法

#	Algorithm	Sequence
1	SCAN	{⇕(w0);⇕(r0);⇕(w1);⇕(r1)}
2	SCAN+	{⇑(w0);⇑(r0);⇑(w1);⇑(r1);⇓(w0);⇓(r0);⇓(w1);⇓(r1)}
3	MATS	{⇕(w0);⇕(r0,w1);⇕(r1)}
4	MATS+	{⇕(w0);⇑(r0,w1);⇓(r1,w0)}
5	MATS++	{⇕(w0);⇑(r0,w1);⇓(r1,w0,r0)}
6	March C-	{⇕(w0);⇑(r0,w1);⇑(r1,w0);⇓(r0,w1);⇓(r1,w0);⇕(r0)}
7	March A	{⇕(w0);⇑(r0,w1,w0,w1);⇑(r1,w0,w1);⇓(r1,w0,w1,w0);⇓(r0,w1,w0)}
8	March B	{⇕(w0);⇑(r0,w1,r1,w0,r0,w1);⇑(r1,w0,w1);⇓(r1,w0,w1,w0);⇓(r0,w1,w0)}
9	Algorithm B	{⇕(w0);⇑(r0,w1,w0,w1);⇑(r1,w0,r0,w1);⇓(r1,w0,w1,w0);⇓(r0,w1,r1,w0)}
10	March C+	{⇑(w0);⇑(r0,w1,r1);⇑(r1,w0,r0);⇓(r0,w1,r1);⇓(r1,w0,r0); ⇓(r0)}
11	PMOVI	{⇓(w0);⇑(r0,w1,r1);⇑(r1,w0,r0);⇓(r0,w1,r1);⇓(r1,w0,r0)}
12	March 1/0	{⇑(w0);⇑(r0,w1,r1);⇓(r1,w0,r0);⇑(w1);⇑(r1,w0,r0); ⇓(r0,w1,r1)}
13	March TP	{⇓(w0);⇑(r0,w1);⇑(r1,w0);⇓(r0,w1,r1);⇓(r1,w0,r0)}
14	March U	{⇕(w0);⇑(r0,w1,r1,w0);⇑(r0,w1);⇓(r1,w0,r0,w1);⇓(r1,w0);⇓(r0)}
15	March X	{⇕(w0);⇑(r0,w1);⇓(r1,w0);⇕(r0)}
16	March Y	{⇕(w0);⇑(r0,w1,r1);⇓(r1,w0,r0);⇓(r0)}
17	March LR	{⇕(w0);⇓(r0,w1);⇑(r1,w0,r0,w1);⇑(r1,w0);⇑(r0,w1,r1,w0);⇑(r0)}
18	March LA	{⇕(w0);⇑(r0,w1,w0,w1,r1);⇑(r1,w0,w1,w0,r0);⇓(r0,w1,w0,w1,r1); ⇓(r1,w0,w1,w0,r0);⇓(r0)}
19	March RAW	{⇕(w0);⇑(r0,w0,r0,r0,w1,r1);⇑(r1,w1,r1,r1,w0,r0);⇓(r0,w0,r0,r0,w1,r1); ⇓(r1,w1,r1,r1,w0,r0);⇕(r0)}
20	March RAW1	{⇕(w0);⇕(w0,r0);⇕(r0);⇕(w1,r1);⇕(r1);⇕(w1,r1);⇕(r1);⇕(w0,r0);⇕(r0)}
21	March AB	{⇕(w1);⇓(r1,w0,r0,w0,r0);⇓(r0,w1,r1,w1,r1);⇑(r1,w0,r0,w0,r0); ⇑(r0,w1,r1,w1,r1);⇕(r1)}
22	March AB1	{⇕(w0);⇕(w1,r1,w1,r1,r1);⇕(w0,r0,w0,r0,r0)}
23	March BDN	{⇑(w0);⇓(r0,w1,r1,w1,r1);⇓(r1,w0,r0,w0,r0);⇑(r0,w1,r1,w1,r1); ⇑(r1,w0,r0,w0,r0);⇑(r0)}
24	March SR	{⇓(w0);⇑(r0,w1,r1,w0);⇑(r0,r0);⇑(w1);⇓(r1,w0,r0,w1);⇓(r1,r1)}
25	March SS	{⇕(w0);⇑(r0,r0,w0,r0,w1);⇑(r1,r1,w1,r1,w0);⇓(r0,r0,w0,r0,w1); ⇓(r1,r1,w1,r1,w0);⇕(r0)}
26	BLIF	{⇕(w0);$_x$⇕(w1,r1,w0);⇕(w1);$_x$⇕(w0,r0,w1)}
27	Ham5R	{⇑(w0);⇑(w1,r1^5);⇑(w0,r0^5);⇓(w1,r1^5);⇓(w0,r0^5)}
28	Ham5W	{⇑(w0);⇑(w0^5,r0);⇑(w1^5,r1);⇓(w0^5,r0);⇓(w1^5,r1)}
29	March G	{⇕(w0);⇑(r0,w1,r1,w0,r0,w1);⇑(r1,w0,w1);⇓(r1,w0,w1,w0);⇓(r0,w1,w0); ***D***;⇑(r0,w1,r1);***D***;⇑(r1,w0,r0)}
30	Ham_Walk	{⇑(w1);⇑(w0);⇑(r0,w1,r1,w0,r0);⇑(r0,w1);⇑(r1,w0,r0,w1,r1);⇑(r1)}

4.3 算法测试参数

除了用于激活和检测错误的内存测试算法操作序列外，也使用了一些算法测试参数以影响测试表现并提高检测错误的能力。算法测试参数直接关系到测试算法的性能。这些参数影响寻址方式和测试算法的数据类型。

算法测试的参数有：

- 寻址方向
- 寻址模式
- 后台数据

所有的算法参数都可被相互独立地组合在一起。

4.3.1 寻址方向

寻址方向指出了行或列地址以何种顺序渐增或渐减。无论是首先增加/减少行地址还是首先增加/减少列地址都是可行的。因此，就有了两种类型的地址方向。

该特性使能了更多被施加到位或字线上的应力，以便在这些线上检测到动态错误。

4.3.1.1 快速的列寻址

一步一步地，行地址被首先增加或减少；如行进方向沿着位线，当达到上限/下限范围时，MBIST / ECC 自动对下一个有效的地址进行环绕式处理（根据方向）。

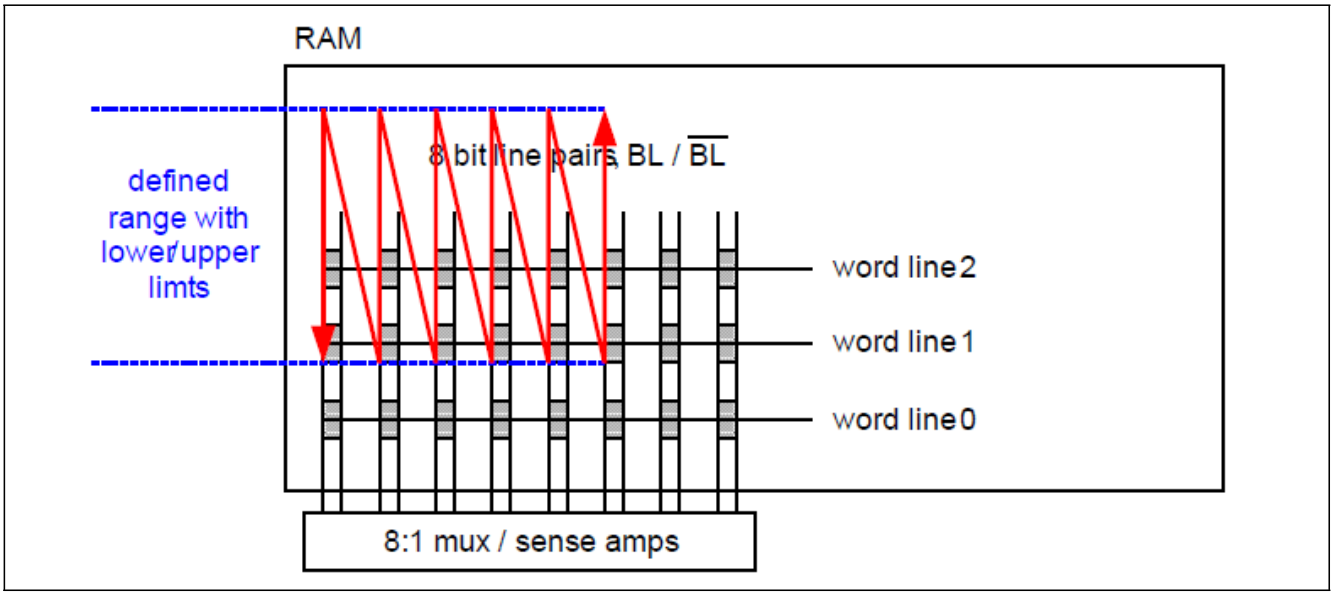

图 6 快速的列寻址

4.3.1.2 快速的行寻址

一步一步地，列地址被首先增加或减少；如行进方向沿着字线，当达到上限/下限范围时，MBIST / ECC 自动对下一个有效的地址进行环绕式处理（根据方向）。

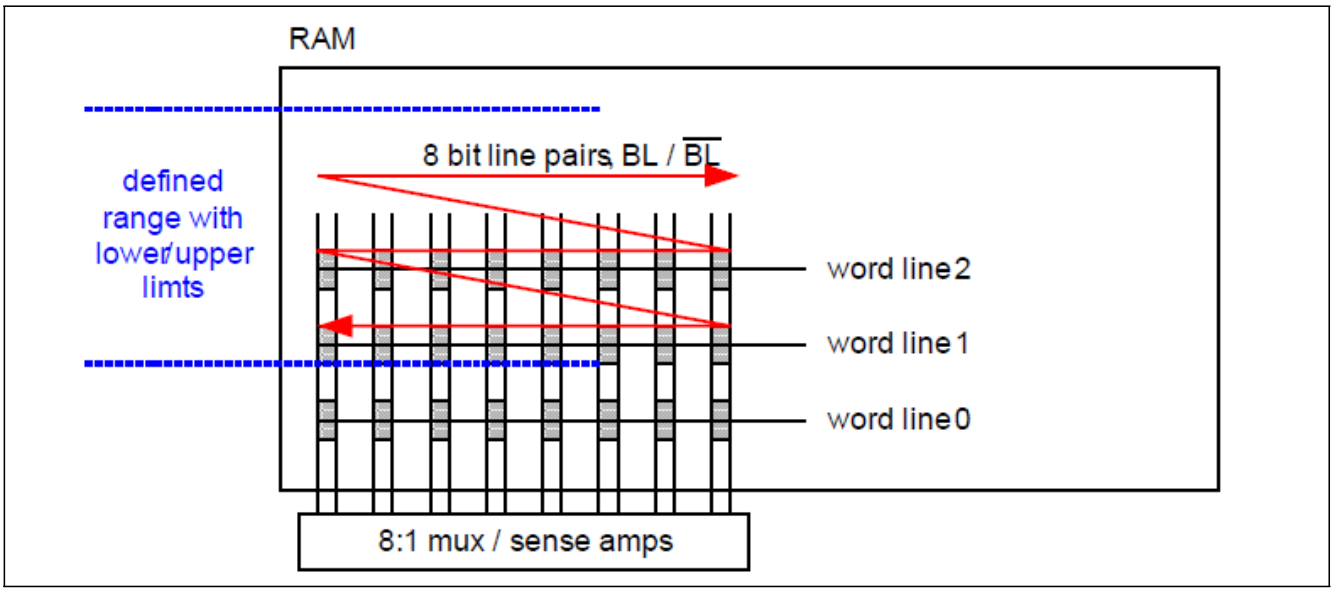

图 7 快速的行寻址

4.3.2 寻址方式

寻址方式定义了定址存储单元的计数时序。寻址方式有：

•线性
•2^i (2 的 i 次幂)
•格雷码
•地址补码

图 8 阐述了不同的寻址方式的寻址时序。每个例子中的物理地址布置都相当于线性寻址方式。单元的编号显示了顺序。

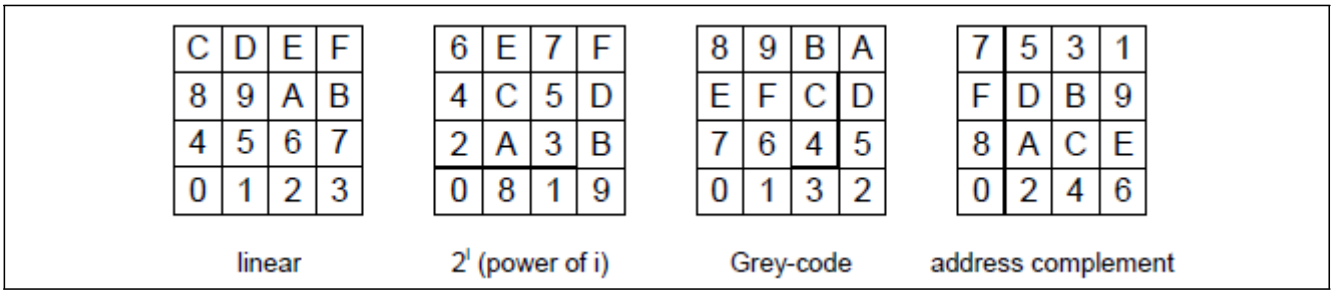

图 8 寻址方式

线性寻址方式以线性的方式沿着列或行，逐一访问存储单元。该寻址方式不仅极易实现，而且还被广泛使用。

如同在上图中说明的一样，更复杂的寻址方式更改顺序时，就要以特定的方式访问单元。

注释：建议采用线性寻址方式，且英飞凌 MBIST 支持该寻址方式。

4.3.3 后台数据

后台数据描述了写入到内存的数据类型（非倒置的“0”和倒置的“1”）。
可将它描述为覆盖在存储器阵列上的掩模。

数据背景的类型有：

- Solid
- Row-stripe
- Column-stripe
- Checkerboard

图 9 阐明了这四个后台数据非倒置的(“0”) 和倒置的(“1”) 数据的类型。

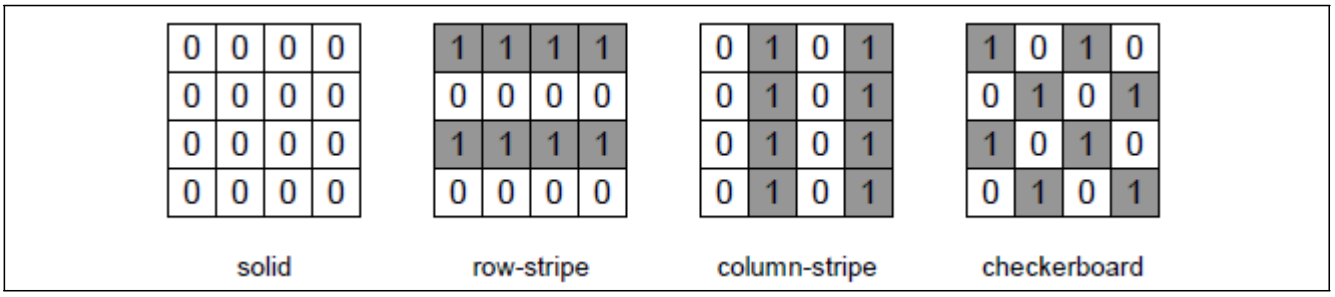

图 9 后台数据类型

这四种后台数据是最常使用的,但也有其它的类型.

column-stripe 和 Checkerboard 已被证明是内存测试最有效的数据背景。

4.4 测试算法的有效性

在内存测试里，测试算法的有效性必不可少。测试时间必须简短，但同时错误覆盖率却又要尽可能地高。测试算法的错误覆盖率是检测到的错误数量与总错误数量之间的比。

可以以不同的方式确定内存测试算法的有效性。对单个测试算法错误覆盖率的分析表明了一个算法独立于其它算法的有效性如何。对于复杂的测试，在一个内存测试设置中，结合了两个或多个测试算法。因此可以评估两种算法组合的有效性。

使用不同的算法测试参数会影响每次测试算法的错误覆盖率。分析表明，一般来说，对于所有的算法而言，最有效的数据背景是 Checkerboard 和 column-stripe。分析也表明了，选择“快速行”或“快速列”寻址没有不同的后台数据重要。

测试设置最优化意味着选择内存测试的测试算法和算法测试参数应同时满足两个要求：

- 最大错误覆盖率
- 最小测试时间

从可获得的测试算法和算法测试参数中，内存测试时建议使用以下测试算法：

- 非破坏性的倒置测试
- Checkerboard 测试(采用 Checkerboard 后台数据的 SCAN 测试)
- March-U 测试

5 执行和用途

英飞凌 MBIST/ECC 支持"行进"算法,其中位的值“走遍”内存。使用 MBIST 寄存器，用户就可以配置从表 1 给定的算法中所选取的任意行进测试算法，或者是创造它们自己的测试算法。同时，还可以控制测试行为并读出测试结果。

本章节描述了在 MTU 驱动程序中实现的内存测试，以及为 B-step 硅片实现算法的代码。

注意： *当 MBIST 模块被 MTU_MEMTEST.MEMxEN 寄存器使能或停用时，为了防止 RAM 内容的自动初始化，必须按照如下方式在 PROCOND 和 PROCOND 备份寄存器里配置 DFlash 保护：*

FLASH0.PROCOND.RAMIN = 11

FLASH0.PROCOND.RAMINSEL = 1111

这样应用使能了使用 MBIST 控制器对数据的修改或运行时的内存的测试。

5.1.1 启动内存测试时序

如同以下例子的描述，为了启动内存测试，必须将特定的值写入到 MBIST 配置和控制寄存器。

1. 进入内存测试模式
   - 使能 MBIST 存储控制器，将 MTU_MEMTESTx 寄存器里相应的位置位
   - 为需被测试的 CPU 内存选择基于内存映像的测试模式，将 MTU_MEMMAP 寄存器里相应的位置位（只有 CPU 缓存映射位生效）

2. 初始化存储控制器寄存器
   - 将行进测试序列写入到 CONFIG 寄存器
   -MUNACCS 规定了对当前行进单元里单个地址的访问总次数
   -ACCSTYPE 规定了对当前跨步单元里单个地址所执行的访问类型（读还是写）
   -AG_MOD 规定了寻址方式
   -ACCSPAT 规定了对当前跨步单元里每单个地址进行访问时所使用的位类型。
   - 定义要执行的测试，在 MCONTROL 寄存器里规定所需的行为。
   -BITTOG 和 ROWTOG 位被用于生成在行进测试期间所使用的后台数据类型。表 2 显示了用 BITTOG 和 ROWTOG 位生成的数据类型的定义。

表 2 数据类型定义

Pattern	BITTOG [7]	ROWTOG [6]
Solid	0	0
Row stripe	0	1
Column stripe	1	0
Checkerboard	1	1

- 定义需在 RANGE 寄存器里测试的内存范围

- RANGE :=<memory range to be tested>

注释：用在 MBIST / ECC 里的地址是逻辑地址。因此在写 RANGE 寄存器之前，当将物理地址转变为逻辑地址时，必须实现并考虑地址加扰算法。

注意：由于实现 RANGE 定义所用的地址加扰算法复杂，因此不打算将 RANGE 作为一个特别特征。对英飞凌 AURIX™ 器件内存的架构和配置都需达到专家级的水平，以便合理地定义部分内存测试或部分内存初始化。因此强烈建议不要试图配置 RANGE 寄存器，并使用默认值，因为他们使能了由某个特定储存控制器控制的整个内存条的测试。

3. 开启内存测试
   - 通过将“1”写入到 MCONTROL.START 寄存器位开启内存测试

4. 等待测试的结束
   - 轮询 STATUS 寄存器里的 DONE 位

5. 检查测试结果
   - 验证 STATUS 寄存器里的 FAIL 位

注释：复杂的算法可能需要多种跨步启动，以获取充分的测试。

5.2 MTU 执行的算法

建议在 MTU 驱动程序里实现如下 3 种广为人知的测试算法:

- 非破坏性的倒置测试
- Checkerboard 测试(采用 Checkerboard 数据背景的 SCAN 测试)
- March-U test

5.2.1 非破坏的倒置测试

该测试是一种简单的线性测试,无需损坏任何用户数据便可找出所有的固定错误(对于任何操作该单元都固定在一个值)。测试访问 RANGE 寄存器定义的地址范围里的每个字 4 次。

测试步骤:

- 读取包含了校验位的数据字
- 写回所有的取反位
- 读取包含了校验位的数据字
- 写回所有的取反位

在这个程序结束时，用户数据保持不变且每一个位都被设置为“0”和“1”。
任何单次固定错误都被视为一个可更正的错误。

MBIST/ECC 不产生错误位或错误信息丢弃请求。测试将仅仅贯穿指定的存储范围，且通过设置 MSTATUS.DONE 位，发出“完成”信号。固件必须读取固化的不能修正的错误检测位 ECCD.UERR。ETRR 寄存器包含第一个错误的地址，并且如果检测到多个错误，ECCD.EOV 位置位。

测试算法：{⇧ (r0, w1, r1, w0)}

测试建立了一个行进测试，它只包含了一个行进单元，但可以对 RANGE 寄存器指定的存储单元里的每个存储位置进行 4 次访问；如读非反转类型，写反转类型，读反转类型和写非反转类型。

因为该算法访问每个内存地址 4 次，因此测试长度是 TL=4n。

代码 1 阐述了如何实现 AURIX™ B-step 硅的非破坏的倒置测试算法。

注释：传送到安全看门狗功能元件用于清除/设置 ENDINIT 保护位的参数密码代表用户定义的 ENDINIT 密码。可将该参数密码宣称为一个全局变量，或者是作为一个预编译的宏定义来提供。有关 CPU 和安全看门狗的实现和用途，在“CPU 和安全看门狗”应用笔记 AP32221 里进行了描述。

代码 1　非破坏的倒置测试

```
uint8 IfxMtu_Non_Destructive_test(
        uint8 mbistSel, uint8 numberRedundancyLines, uint8 RangeSel,
        uint8 RangeAddrUp, uint8 RangeAddrLow, uint16 *ErrorAddr)
{
        uint8 retVal = 0U;

        /* Select MBIST Memory Controller:
         * Ifx_MC is a type describing structure of MBIST Memory Controller
         * registers defined in IfxMc_regdef.h file - MC object */
        Ifx_MC *mc = (Ifx_MC *)(IFXMTU_MC0_ADDR + 0x100 * mbistSel);

        /* Enable MBIST Memory Controller */
        IfxScuWdt_clearSafetyEndinit(password);
        IfxMtu_lenableMbistShell(mbistSel);
        IfxScuWdt_setSafetyEndinit(password);

        /* Configure Non-destructive Inversion test */
        mc->CONFIG0.U = 0x4005;             //NUMACCS=4, ACCSTYPE=5
        mc->CONFIG1.U = 0x5000;             //AG_MOD=5

        /* Set the range register */
        mc->RANGE.U = (RangeSel << 15) | (RangeAddrUp << 7) | (RangeAddrLow << 0);

        /* Run the tests */
        mc->MCONTROL.U = (0xF000 | IFXMTU_FAILDMP_EN | IFXMTU_START_TEST);            //0xF201
        mc->MCONTROL.B.START = 0;

        /* Wait for the end of the test */
        while (!(mc->MSTATUS.U & IFXMTU_DONE));

        /* Check the Fail Status */
        if ((mc->MSTATUS.U & IFXMTU_FAIL) == IFXMTU_FAIL) {
                /* Test has failed, check if any un-correctable error */
                 if (mc->ECCD.B.UERR) {
                        /* Read the Error tracking register and return saying test failed */
                        *ErrorAddr     = mc->ETRR[0].U;
                        retVal         = 1U;
                }
        }

        /* Disable Memory Controller */
        IfxScuWdt_clearSafetyEndinit(password);
        IfxMtu_ldisableMbistShell(mbistSel);
        IfxScuWdt_setSafetyEndinit(password);
        return (retVal);
}
```

5.2.2 Checkerboard 测试

该测试映射广为人知的 SCAN 测试,其中测试算法如表 1 所示, Checkerboard 后台数据如图 10 所示。测试包括了将物理 Checkerboard 模型写入到内存，并回读以验证。

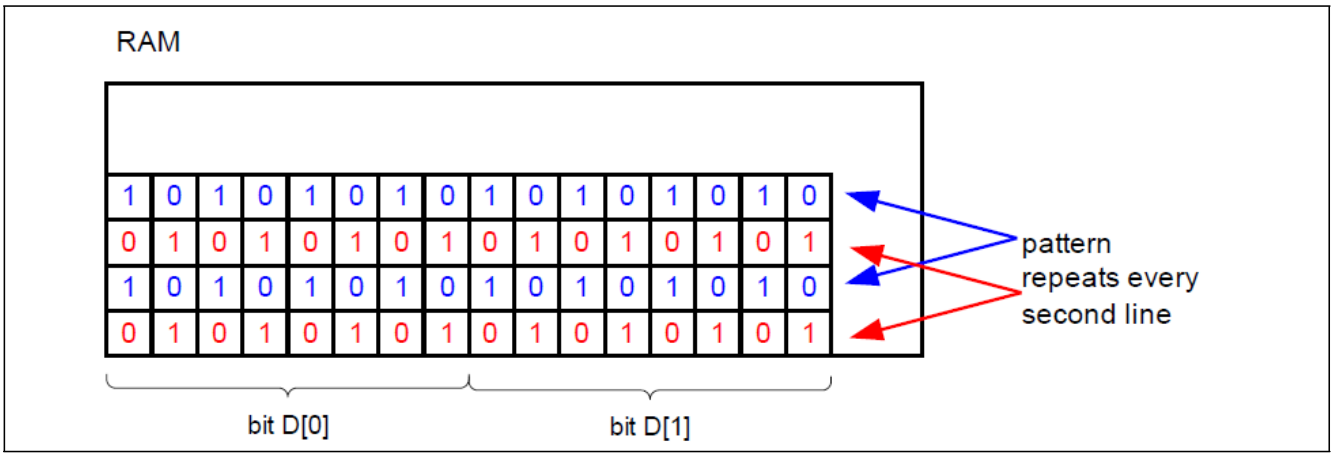

图 10 Checkerboard 数据模型

测试算法：{ ⇧(w0) ; ⇧ (r0) ; ⇩(w1) ; ⇩(r1) }

该测试建立了一个跨步测试，它包含了 4 个行进单元。每个行进单元对 RANGE 寄存器指定的存储单元里的每个内存位置进行一次访问。因为该算法访问每个内存地址 4 次，因此测试长度是 TL=4n。

表 3 描述了 Checkerboard 测试时序。

表 3 Checkerboard 测试时序

	方向	跨步单元	描述
1	Up	W0	Write checkerboard data pattern
2	Up	R0	Read checkerboard data pattern
3	Down	W1	Write inverted checkerboard data pattern
4	Down	R1	Read inverted checkerboard data pattern

代码 2 给出了用于实现 AURIX™ B-step 硅片的 Checkerboard 算法。

注释： 传送到安全看门狗功能元件用于清除/设置 ENDINIT 保护位的参数密码代表用户定义的 ENDINIT 密码。可将该参数密码宣称为一个全局变量，或者是作为一个预编译的宏定义来提供。有关 CPU 和安全看门狗的实现和用途，在“CPU 和安全看门狗”应用笔记 AP32221 里进行了描述。

代码 2　Checkerboard 测试

```
uint8 IfxMtu_CheckerBoard_test(
        uint8 mbistSel, uint8 numberRedundancyLines, uint8 RangeSel,
        uint8 RangeAddrUp, uint8 RangeAddrLow, uint16 *ErrorAddr)
{
        uint8 retVal = 0U;
        uint8 testStep;

        /* Select MBIST Memory Controller:
         * Ifx_MC is a type describing structure of MBIST Memory Controller
         * registers defined in IfxMc_regdef.h file - MC object */
        Ifx_MC *mc = (Ifx_MC *)(IFXMTU_MC0_ADDR + 0x100 * mbistSel);

        uint32 configCheckerBoardSequence[4] = {0x08001000,            //up   /lin/w0
                                                0x08001001,            //up   /lin/r0
                                                0x00011000,            //down/lin/w1
                                                0x00011001};           //down/lin/r1

        /* Enable Memory Controller */
        IfxScuWdt_clearSafetyEndinit(password);
        IfxMtu_lenableMbistShell(mbistSel);
        IfxScuWdt_setSafetyEndinit(password);

        /* Set the range register */
        mc->RANGE.U = (RangeSel << 15) | (RangeAddrUp << 7) | (RangeAddrLow << 0);

        /* Run the test */
        For (testStep = 0; testStep < 4; ++testStep) {
                mc->CONFIG0.U = configCheckerBoardSequence[testStep] & 0x0000FFFF;
                mc->CONFIG1.U = (configCheckerBoardSequence[testStep] & 0xFFFF0000) >> 16;

                mc->MCONTROL.U = (0x00C8 | IFXMTU_FAILDMP_EN | IFXMTU_START_TEST);        //0x02C9
                mc->MCONTROL.B.START = 0;

                /* Wait for the end of the test */
                while (!(mc->MSTATUS.U & IFXMTU_DONE));

                /* Check the Fail Status */
                if ((mc->MSTATUS.U & IFXMTU_FAIL) == IFXMTU_FAIL) {
                        /* Test has failed, check if any un-correctable error */
                        if (mc->ECCD.B.UERR) {
                                /* Read Error tracking register and return saying test failed */
                                *ErrorAddr    = mc->ETRR[0].U;
                                retVal        = 1U;
                                break;
                        }
                }
        }

        /* Disable Memory Controller */
        IfxScuWdt_clearSafetyEndinit(password);
        IfxMtu_ldisableMbistShell(mbistSel);
        IfxScuWdt_setSafetyEndinit(password);
        return (retVal);
}
```

5.2.3 March-U 测试

贯穿存储器阵列的测试“行进” “0”和“1”值都是以向上和向下的方向。更准确的是，“0” 和“1”值在内存里以各个方向传播开去；如一个位单元切换到各个方向，其相邻的单元有一个给定的值，换句话说，有一个反转的给定值。

测试算法：{⇧(w0)；⇧(r0,w1,r1,w0)；⇧(r0,w1)；⇩(r1,w0,r0,w1)；⇩(r1,w0)；⇩(r0)}

该测试建立了一个行进测试，包含了 4 个行进单元。行进单元从各个不同的方向访问内存，在 RANGE 寄存器指定的内存单元里，对各个存储单元访问 14 次。

因为该算法访问每个储存单元 14 次，因此测试长度是 TL=14n。

表 4 描述了 March-U 测试时序。

表 4 March-U 测试时序

	方向	跨步单元	描述
1	Up	W0	Write “0”
2	Up	R0->W1->R1->W0	Verify “0”；write “1”；verify “1”；write “0”
3	Up	R0->W1	Verify “0”；write “1”
4	Down	R1->W0->R0->W1	Verify “1”；write “0”；verify “0”；write “1”
5	Down	R1->W0	Verify “1”；write “0”
6	Down	R0	Verify “0”

如何为 B-step 硅片实现 March-U 测试算法在代码 3 中进行了阐述。

注释： 传送到安全看门狗功能元件用于清除/设置 ENDINIT 保护位的参数密码代表用户定义的 ENDINIT 密码。可将该参数密码宣称为一个全局变量，或者是作为一个预编译的宏定义来提供。有关 CPU 和安全看门狗的实现和用途，在“CPU 和安全看门狗”应用笔记 AP32221 里进行了描述。

代码 3 March-U 测试

```
uint8 IfxMtu_MarchU_test(
        uint8 mbistSel, uint8 numberRedundancyLines, uint8 RangeSel,
        uint8 RangeAddrUp, uint8 RangeAddrLow, uint16 *ErrorAddr)
{
        uint8 retVal = 0U;
        uint8 testStep;

        /* Select MBIST Memory Controller:
         * Ifx_MC is a type describing structure of MBIST Memory Controller
         * registers defined in IfxMc_regdef.h file - MC object */
        Ifx_MC *mc = (Ifx_MC *)(IFXMTU_MC0_ADDR + 0x100 * mbistSel);

        uint32 configMarchUSequence[6] ={0x08001000,          //up   /lin/w0
                                         0x08064005,          //up   /lin/r0->w1->r1->w0
                                         0x08022001,          //up   /lin/r0->w1
                                         0x00094005,          //down/lin/r1->w0->r0->w1
                                         0x00012001,          //down/lin/r1->w0
                                         0x00001001};         //down/lin/r0
```

```
    /* Enable Memory Controller */
    IfxScuWdt_clearSafetyEndinit(password);
    IfxMtu_lenableMbistShell(mbistSel);
    IfxScuWdt_setSafetyEndinit(password);

    /* Set the range register */
    mc->RANGE.U = (RangeSel << 15) | (RangeAddrUp << 7) | (RangeAddrLow << 0);

    /* Run the test */
    For (testStep = 0 ; testStep < 6 ; ++testStep) {
        mc->CONFIG0.U = configMarchUSequence[testStep] & 0x0000FFFF;
        mc->CONFIG1.U = (configMarchUSequence[testStep] & 0xFFFF0000) >> 16;

        mc->MCONTROL.U = (0x0008 | IFXMTU_FAILDMP_EN | IFXMTU_START_TEST); //0x0209 mc-
        >MCONTROL.B.START = 0;

        /* Wait for the end of the test */
        while (!(mc->MSTATUS.U & IFXMTU_DONE));

        /* Check the Fail Status */
        if ((mc->MSTATUS.U & IFXMTU_FAIL) == IFXMTU_FAIL) {
            /* Test has failed, check if any un-correctable error */
             if (mc->ECCD.B.UERR) {
                /* Read Error tracking register and return saying test failed */
                *ErrorAddr    = mc->ETRR[0].U;
                retVal        = 1U;
                break;
            }
        }
    }

    /* Disable Memory Controller */
    IfxScuWdt_clearSafetyEndinit(password);
    IfxMtu_ldisableMbistShell(mbistSel);
    IfxScuWdt_setSafetyEndinit(password);
    return (retVal);
}
```

6 附录：使能/停用存储控制器

在本章我们介绍用 MTU 功能使能并停用存储控制器，其中控制器号作为函数的参数。

6.1 使能存储控制器

函数 IfxMtu_lenableMbistShell（代码 4）被用来使能存储控制器，其中控制器号作为函数的参数。

代码 4 使能存储控制器

```
void IfxMtu_lenableMbistShell(uint8 mbistSel)
{
        /* Enable MBIST Memory Controller with number given by mbistSel
         * and wait until MBIST is ready */
        if (mbistSel < 32)
        {
                MTU_MEMTEST0.U = (1U << mbistSel);
                while (!(MTU_MEMTEST0.U & (1U << mbistSel)));
        }
        else if (mbistSel < 64)
        {
                MTU_MEMTEST1.U = (1U << (mbistSel - 32));
                while (!(MTU_MEMTEST1.U & (1U << (mbistSel - 32))));
        }
        else if (mbistSel < 88)                                  //88 Memory Controllers
        {
                MTU_MEMTEST2.U = (1U << (mbistSel - 64));
                while (!(MTU_MEMTEST2.U & (1U << (mbistSel - 64))));
        }
}
```

6.2 停用存储控制器

函数 IfxMtu_ldisableMbistShell（代码 5）被用于停用存储控制器，其中控制器号作为函数的参数。

代码 5 停用存储控制器

```
void IfxMtu_ldisableMbistShell(uint8 mbistSel)
{
        /* Disable MBIST Memory Controller with number given by mbistSel
         * and wait until MBIST becomes disabled */
        if (mbistSel < 32)
        {
                MTU_MEMTEST0.U &= ~(1U << mbistSel);
                while (MTU_MEMTEST0.U & (1U << mbistSel));
        }
        else if (mbistSel < 64)
        {
                MTU_MEMTEST1.U &= ~(1U << (mbistSel - 32));
                while (MTU_MEMTEST1.U & (1U << (mbistSel - 32)));
        }
        else if (mbistSel < 88)                                  //88 Memory Controllers
        {
                MTU_MEMTEST2.U &= ~(1U << (mbistSel - 64));
                while (MTU_MEMTEST2.U & (1U << (mbistSel - 64)));
        }
}
```

w ww.infineon.com

Infineon Technologies AG 出版