Arm® PMC-100

Revision: r0p0

Technical Reference Manual

101528_0000_01_en

Arm® PMC-100

Technical Reference Manual

Release Information

Document History

Issue	Date	Confidentiality	Change
0000-01	21 April 2021	Non-Confidential	First early access release for r0p0

Contents

Arm® PMC-100 Technical Reference Manual

Preface

This preface introduces the *Arm® PMC-100 Technical Reference Manual*.

It contains the following:

- *About this book* on page 8.
- *Feedback* on page 10.

About this book

This manual is for the PMC-100. This document describes the behavior of the PMC-100, including the programmer's model and signals.

Product revision status

The r*x*p*y* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

r*x* Identifies the major revision of the product, for example, r1.

p*y* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This manual is written to help system designers, system integrators, verification engineers, and software programmers who are implementing a *System on Chip* (SoC) device based on the PMC-100 processor.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter provides an overview of the PMC-100 and its features.

Chapter 2 MBIST usage models
IP cores that support on-line *Memory Built-In Self-Test* (MBIST) also support on-line MBIST on-line memory and on-line MBIST off-line memory in-field test usage models.

Chapter 3 PMC-100 functional description
This chapter describes the PMC-100 functionality.

Chapter 4 PMC-100 programmers model
This chapter describes the PMC-100 registers and provides more information on programming the processor.

Appendix A Short-burst software-transparent algorithm
This chapter describes the short-burst software-transparent algorithm.

Appendix B Production test March Algorithm
Ths chapter describes the March *Memory Buile-In Self-Test* (MBIST) algorithm. It also provides information on how to program PMC-100 to perform an example March MBIST algorithm called March C-, and this information can be used as the basis to implement other production test MBIST algorithms.

Appendix C On-line MBIST Memory Protection Logic Test Algorithms
The on-line *Memory Built-In Self Test* (MBIST) test algorithms described in this section show how *Error Correcting Code* (ECC) generation, checking, correction logic, parity generation, and checking logic can be tested.

Appendix D Miscellaneous Algorithms
This section describes miscellaneous algorithms.

Appendix E Signal descriptions
This appendix describes the PMC-100 signals.

Appendix F PMC-100 software library
This section describes the PMC-100 software library.

Appendix G Revisions
This appendix describes the technical changes between released issues of this book.

Glossary

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

See the *Arm® Glossary* for more information.

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` (underlined: mono)
: 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.

`<and>`
: Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
ADD Rd, SP, #<imm>
```

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

Additional reading

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

Arm publications

- *AMBA® APB Protocol Version 2.0 Specification* (IHI 0033).
- *Arm® CoreSight™ Architecture Specification v3.0* (IHI 0029).

Other publications

None

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm PMC-100 Technical Reference Manual*.
- The number 101528_0000_01_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter provides an overview of the PMC-100 and its features.

It contains the following sections:

1.1 PMC-100 overview

PMC-100 allows transparent, in-field testing of SRAMs and memory protection logic within a core concurrently with functional operation, without corrupting memory or logic state.

PMC-100 is typically used in functional safety applications and can be used to carry out testing on a periodic basis, at power on/off, or when a parity or *Error Correcting Code* (ECC) error is detected. It is programmable and highly parameterized, and therefore, it can be used with any IP core that supports on-line *Memory Built-In Self Test* (MBIST) and contains embedded memory. PMC-100 can also be used to dump embedded memory content to a debugger using a software read triggered execution mode, making it useful for silicon bring-up and investigating software faults such as cache coherency bugs.

It allows memory and memory protection logic to be tested at-speed. Therefore, MBIST transactions are performed back-to-back using the IP core clock, and as a result, delay faults can be detected by toggling signals at full functional speed.

It is also used in top-level testbenches and FPGAs to validate the on-line MBIST features of cores. Software running on a processor or software running on another processor within an SoC might use PMC-100 to test itself. An example is a safety agent, which monitors and errors and controls testing. *Software Test Libraries* (STLs), use PMC-100 to perform in-field memory testing of parity or ECC logic and other logic within the memory data path.

Example software is also available to show how PMC-100 can be used to perform various types of testing. PMC-100 is also used in top-level testbenches and FPGAs to the validate the on-line MBIST features of IP cores.

Interfaces

PMC-100 has the following standard interfaces:

- On-line MBIST master interface.
- AMBA 4 APB slave interface.

The MBIST interface is fully parameterized, providing compatibility with all IP that supports on-line MBIST. The APB programming interface has privilege and security indication signals to ensure that PMC-100 can only be accessed by software running the highest Security state.

Programming requirements

PMC-100 uses a programmable microcode-based architecture to allow a high degree of flexibility to accommodate different use models and memory test algorithms. PMC-100 has several software-accessible read/write registers containing configuration, control, memory data, memory address, and microcode information.

PMC-100 is software driven. Therefore, it will not function without being programmed. It contains data, address, program, configuration, control and ID registers that software can program. This is different to production MBIST controllers, that do not require programming. Therefore, prior to testing a memory array, PMC-100 must be programmed with attribute information for the memory array to be tested. For more information on the PMC-100 programmers model, see *Chapter 4 PMC-100 programmers model* on page 4-30.

Microcode execution can be initiated by software reading a data register. Wait states are inserted on the APB interface until execution is completed and read data is returned, allowing more efficient software operation.

Using PMC-100 for testing:

Example software is available for PMC-100 to show how it can be used to test memory protection logic, SRAMs using March algorithm (destructive algorithm) and on-line MBIST Short burst algorithms (non-destructive algorithm). March algorithms can be fully implemented in microcode. Therefore, programming is not required after execution of each March element.

Faulty address and faulty data bitmap information can be read by software. XOR and XOR mask register allow injection of errors in address and data values. XOR mask register left and right shifting function.

PMC-100 is used in ECC/parity logic testing, allowing program loops to be constructed to test every address or data bit. Looping operations, loop start register, loop suspend register and loop counter register allow efficient program loops to be constructed. Fixed data patterns can also be used. It is possible to select repeating patterns of `0b10101010`, `0b01010101`, `0b10100101`, `0b01011010`, `0b11111111` or `0b00000000`. This can be used to improve the performance of the short burst algorithm. The core can be interrupted when a test has completed or if a memory fault is detected. Individual test bursts can be triggered by hardware or software.

PMC-100 allows the processor to perform arbitrary read and write accesses to SRAM data and ECC fields. Multiple SRAM test bursts can be executed back-to-back using a power of 2 software configured loop counter. This improves the test efficiency for cores that have a larger penalty for on-line MBIST entry. Two port SRAM testing is supported using separate read and write address signals on the MBIST interface. The test continue counter allows the gap between test bursts to be programmed when using the on-line memory use model. See *2.1 On-line MBIST on-line memory* on page 2-17

Example integration

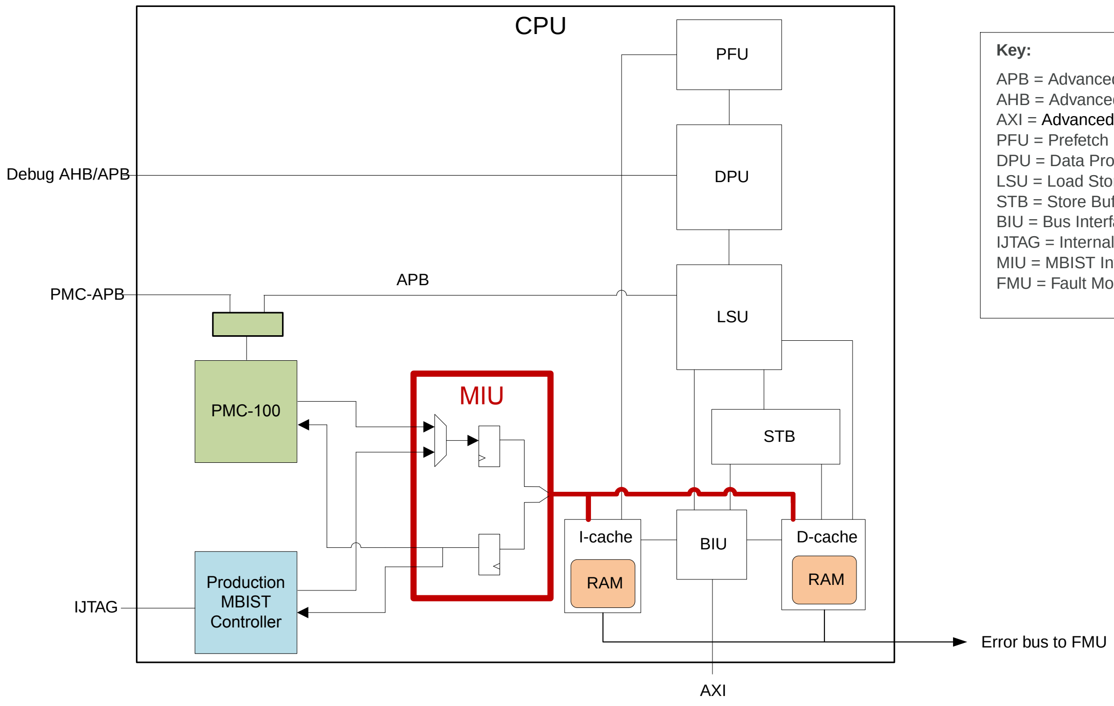
Cores that support on-line MBIST have two internal MBIST slave interfaces that are provided by the *MBIST Interface Unit* (MIU).

- One internal interface for a production MBIST controller.
- One internal interface for PMC-100.

Both the production MBIST and the PMC-100 have the same MBIST data path to and from the embedded SRAMs. The following figure shows the MBIST controller integrated into a simplified representation of a processor core with L1 caches.

The following figure shows:
- An example of a direct MIU connection to the L1 caches and the *Load Store Unit* (LSU), which provides an internal APB interface to allow self-testing software to program PMC-100.
- The internal APB can also be accessed from the debug APB or AHB interface, allowing an external processor to program PMC-100 to test the core.
- It is also possible for PMC-100 to inject an error into a memory array and read it back through the ECC checking logic. This causes an ECC error to be generated and if enabled by PMC-100, it is propagated out of the core through its error bus to a *Fault Monitoring Unit* (FMU). This allows testing of the connections between the core and FMU.

Normally ECC errors detected during MBIST accesses are only visible to PMC-100.



Key:

APB = Advanced Peripheral Bus
AHB = Advanced High-performance Bus
AXI = Advanced Extensible Interface
PFU = Prefetch Unit
DPU = Data Processing Unit
LSU = Load Store Unit
STB = Store Buffer
BIU = Bus Interface Unit
IJTAG = Internal JTAG
MIU = MBIST Interface Unit
FMU = Fault Monitoring Unit

Figure 1-1 Example PMC-100 integration

1.2 PMC-100 advantages

The main advantages of PMC-100 include fault detection and memory testing.

The following table summarizes the benefits of PMC-100.

Table 1-1 PMC-100 benefits

Benefit	**Details**
Fault detection	Detecting faults in SRAMs and single-point and latent faults in memory parity or *Error Correcting Code* (ECC) logic.
Testing	Testing can be carried out transparently to software running on the processor core, without corrupting the memory or logic state.
	Error reporting output signals can be tested from an IP core to a SoC-level *Fault Monitoring Unit* (FMU).
ECC error analysis	Testing an SRAM entry when an ECC error is detected to determine if the error is a soft error or a hard error. • For soft errors, there is no need to use memory error cache registers in the core to replace the entry. • For hard errors, the fault is repaired using the memory error cache register to disable entry or SRAM repair features.
Memory error injection	This is for error monitoring software and system hardware verification.
Memory scrubbing	This is to correct soft ECC errors in the SRAM. This prevents error accumulation, therefore, ensuring that correctable single-bit errors do not degenerate into multi-bit errors that cannot be corrected.
Memory dumping and monitoring by an external debugger	This is useful for: • Software and hardware debug. • Silicon bring-up. • Diagnosing SRAM power issues. • Core lockup issues. A debugger can access all memories, even those that are not directly accessibly by software. These include caches, *Translation Lookup Buffers* (TLBs), *Branch Target Buffers* (BTBs), *Long-Term Data Buffers* (LTDBs), and other data buffers.
Cache preloading using an external debugger	This is useful for software and hardware debug.
Initialization of ECC and parity fields in memory.	-

Chapter 2
MBIST usage models

IP cores that support on-line *Memory Built-In Self-Test* (MBIST) also support on-line MBIST on-line memory and on-line MBIST off-line memory in-field test usage models.

It contains the following sections:

2.1 On-line MBIST on-line memory

In this usage model, periodic autonomous or software-initiated short-burst testing can be performed.

For more information on the algorithm that is used, see *Appendix A Short-burst software-transparent algorithm* on page Appx-A-104.

In this case, a series of short transaction sequences or bursts, which are applied separately, test memory. Typically, each burst tests two locations. Each burst lasts fewer than 20 cycles and targets different locations, allowing all locations in a memory to be tested. During each burst, the *Memory Built-In Self Test* (MBIST) controller saves and restores the memory locations under test. After each burst, the MBIST controller automatically increments or decrements the memory address location. This allows all entries within an SRAM to be tested without software intervention.

2.1.1 Test memory with on-line MBIST on-line memory use case

The *Memory Built-In Self Test* (MBIST) controller performs a series of short bursts is applied separately to test memory.

Only one memory type, for example the L1 instruction cache, is locked for processor access during a burst. Therefore, the processor is free to access other memories. If the processor tries to access a locked memory, it stalls until the burst is complete. If the processor tries to access a memory under test, this usage model only has a small impact on performance because bursts are short and occur infrequently. The MBIST controller and on-line MBIST lock in an IP core perform the following steps automatically.

Procedure

1. The MBIST controller requests access to the target memory.
2. When the IP core is ready for memory testing, it acknowledges the request and automatically locks the memory for normal accesses. This guarantees full speed MBIST access and no changes to target memory by software during testing.
3. The MBIST controller performs a short burst of transactions.
4. The MBIST controller releases the request.
5. The lock on targeted memory is automatically removed.

2.2 On-line MBIST off-line memory

In this usage model, standard production *Memory Built-In Self Test* (MBIST) March algorithms are allowed, which requires access to all entries within the memory under test.

For more information on the algorithm that is used, see *Appendix B Production test March Algorithm* on page Appx-B-111.

Memory contents are destroyed during testing, and software might need to save and restore memory contents. The processor cannot access a memory under test because it has been disabled. Therefore, an alternate memory is accessed instead. For example, if an L1 cache is being tested, then software accesses the L2 cache instead. The memory must be taken off-line for a relatively extended period to allow the test algorithm to be carried out. Software must not rely on data that is normally stored in the memory under test.

2.2.1 Test memory with on-line MBIST off-line memory use case

The memory under test is taken off-line, preventing software from accessing it.

In this context, the memory referred to is logical memory. For example, an L1 data cache. The data cache might contain several SRAMs for storing tag and data values, and they are all inaccessible to software during testing. This allows access to all SRAM entries in the memory during test. There might be a degradation in performance because the software disables the memory under test. The following steps are carried out for each test:

Procedure

1. Software disables the memory under test so that it can be used for testing.
2. If the memory contents are required to be preserved, then software saves the memory contents under test in a memory that is not being tested.
3. Software instructs the MBIST controller to test the memory.
4. The MBIST controller tests the memory using Production Test March algorithm.
5. After the test is complete, software checks the MBIST controller for errors.
6. Depending on application requirements and type of memory being tested, software must either restore the contents of the memory or if a cache is being tested, then the cache must be invalidated.
7. Software enables the memory for functional use.

Chapter 3
PMC-100 functional description

This chapter describes the PMC-100 functionality.

It contains the following sections:

3.1 PMC-100 functionality

PMC-100 contains an APB slave interface, register store, and execution unit.

PMC-100 occupies 4 KB in the processor memory map. The APB slave interface provides read and write access to the register store. The register store contains configuration, address data, program, and CoreSight ID registers. The execution unit uses the information stored in the registers to generate transactions on the *Memory Built-In Self Test* (MBIST) interface and check the read data that is returned from the selected memory array.

As shown in the following diagram, the processor can access the registers to configure and initialize PMC-100, but it cannot access the execution unit. Therefore, the processor is decoupled from the MBIST interface and has to set up memory transactions in the program registers in the register store and wait for the execution unit to carry them out. PMC-100 can be configured to interrupt the processor or set a flag when it has carried out the transactions configured by the processor or when an error is detected.

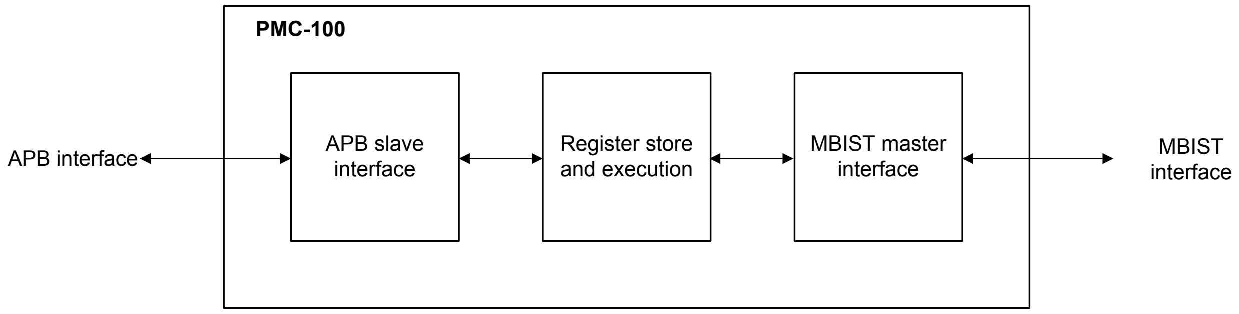


Figure 3-1 PMC-100 functionality

3.2 RTL parameters

RTL parameters that configure the PMC-100 are fixed for the IP core but the `PROGSIZE` parameter is normally a top-level parameter on the IP core and can be modified at implementation time. Software can read the parameters using the DEVID registers.

For more information on DEVID registers, see *4.36 Device Configuration Register, PMC100_DEVID* on page 4-99 and *4.35 Device Configuration Register 1, PMC100_DEVID1* on page 4-98 registers.

The following table describes the RTL parameters.

Table 3-1 RTL parameters

Parameter	Description
`RAR`	*Reset all registers* (RAR). Specifies whether all synchronous state or only required state is reset. • 0 - Only reset state required by the design • 1 - Reset all synchronous state **Note** When the RAR parameters are set to 1, all flops in the design will include an explicit reset.
`FLOPPARITY`	Specifies whether the PMC-100 is configured with parity generation and checks on all flip-flops: • 0 - No parity on flip-flops • 1 - Include parity on flip-flops **Note** If FLOPPARITY is set to 1, then RAR must also be set to 1.
`MAWIDTH[5:0]`	*Memory Built-In Self Test* (MBIST) address width. For more information, see: • **MBISTOLADDR** signal in *E.3 MBIST master interface signals* on page Appx-E-138 • PMC100_RADDR register in *4.15 Row address register, PMC100_RADDR* on page 4-67 • PMC100_HIGHADDR in *4.13 High address register, PMC100_HIGHADDR* on page 4-64
`MDWIDTH[8:0]`	MBIST data width. For more information, see: • **MBISTOLINDATA** and **MBISTOLOUTDATA** signals in *E.3 MBIST master interface signals* on page Appx-E-138 • PMC100_X/PMC100_Y registers in *4.16 Data registers, PMC100_X0-PMC100_X7 and PMC100_Y0-PMC100_Y7* on page 4-70 • PMC100_DM register in *4.20 Data mask, fault bitmap, and data registers, PMC100_DM0-PMC100_DM7* on page 4-74 • PMC100_XM register in *4.21 XOR mask registers, PMC100_XM0-PMC100_XM7* on page 4-76 `MDWIDTH[8:0]` must be greater than or equal to `MAWIDTH[5:0]`.
`MARWIDTH[3:0]`	MBIST array width. For more information, see: • **MBISTOLARRAY** signal in *E.3 MBIST master interface signals* on page Appx-E-138. The **MBISTOLARRAY** signal is divided into two parts, the lower part containing the memory controller and sub-array fields and the upper part containing the protection logic unit field. PMC-100 has a fixed protection logic unit field width of two, which occupies the two **MBISTOLARRAY** MSBs. Therefore, the `MARWIDTH[3:0]` value must include these two bits whether they are used by an IP core. • PMC100_AR register in *4.8 Array register, PMC100_AR* on page 4-56.
`MERWIDTH[5:0]`	PMC100_MER register and **MBISTOLERR** signal width. *4.19 MBISTOLERR input register, PMC100_MER* on page 4-73 and *E.3 MBIST master interface signals* on page Appx-E-138.

Table 3-1 RTL parameters (continued)

Parameter	Description
`MBWIDTH[5:0]`	MBIST byte enable width. For more information, see: • **MBISTOLBE** signal. *E.3 MBIST master interface signals* on page Appx-E-138 • PMC100_BER.BE field in *4.9 Byte enable register, PMC100_BER* on page 4-59
`MCWIDTH[4:0]`	MBIST configuration width. For more information, see: • **MBISTOLCFG** signal in *E.3 MBIST master interface signals* on page Appx-E-138 • PMC100_CFGR register in *4.6 MBISTOLCFG output register, PMC100_CFGR* on page 4-52
`PROGSIZE[5:0]`	Program size. For more information, see: • Program registers in *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77 • PMC100_PCR register in *4.10 Program counter register, PMC100_PCR* on page 4-60
`PDWIDTH[2:0]`	Pipeline depth field width, including memory protection logic pipeline depth, PMC100_MCR.PD and PMC100_MCR.PDP. For more information, see *4.7 Memory control register, PMC100_MCR* on page 4-53
`RCOWIDTH[2:0]`	RAM cycles of operation field width, PMC100_MCR.RCOR and PMC_MCR.RCOW. For more information, see *4.7 Memory control register, PMC100_MCR* on page 4-53
`AIWIDTH[5:0]`	PMC100_AIR register and **AUXIN** signal width. For more information, see *4.17 Auxiliary input register, PMC100_AIR* on page 4-71 and *E.5 Miscellaneous signals* on page Appx-E-140
`AOWIDTH[5:0]`	PMC100_AOR register and **AUXOUT** signal width. For more information, see *4.18 Auxiliary input register, PMC100_AOR* on page 4-72 and *E.5 Miscellaneous signals* on page Appx-E-140.

3.3 Two-port SRAM support

PMC-100 supports two-port SRAMs, that is, SRAMs that have one read port and one write port, allowing a read and a write to be performed simultaneously, at different addresses. Therefore, these SRAMs have separate read and write address input signals. PMC-100 has signals for the write address (**MBISTOLWADDR**) and the read address (**MBISTOLADDR**).

For more information on these signals, see *E.3 MBIST master interface signals* on page Appx-E-138.

When PMC100_CTRL.BAMEN is 0, **MBISTOLWADDR** and **MBISTOLADDR** have different values. When **MBISTOLADDR** is the current address, then **MBISTOLWADDR** is the next address and so on. When PMC100_CTRL.BAMEN is 1, **MBISTOLWADDR** and MBISTOLADDR will have the same value.

For more information on PMC100_CTRL, see *4.5 Main control register, PMC100_CTRL* on page 4-40.

The instruction AO field controls whether the current or next address is output on the **MBISTOLADDR** and so the address output on the **MBISTOLWADDR** signal is the inverse of the AO field, even when there is a single write transaction. This is a consequence of using **MBISTOLADDR** for the write address for single port SRAMs.

The instruction TRANS field has two bits, allowing simultaneous read and write MBIST transactions to be indicated in the microcode. For more information, see *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77.

When there are simultaneous MBIST read and write transactions, the read and write MBIST data are the inverse of each other and this is controlled by the instruction DPOL field. For more information, see *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77. . In this case, DPOL indicates the polarity of the read data signal and the write data signal is controlled by the inverse DPOL value. When there is a single MBIST read or write transaction the data is controlled by the unmodified DPOL value. See *A.4.1 Microcode* on page Appx-A-109 for an example of how PMC-100 can be programmed to test two port SRAMs.

3.4 Loop operations

For an efficient implementation of the March SRAM and memory protection test algorithms, four loop operation types are encoded in the microcode instruction OP field.

The following registers are used with the loop operations:

- The loop counter register, PMC100_LCR, can be used with LOOP-Last and LOOP-LCR operations to implement C-style loops. For more information, see *4.24 Loop counter register, PMC100_LCR* on page 4-84.
- The loop suspend counter register, PMC100_LSCR, can be used with all loop operations to allow several iterations of a loop to be executed before suspending execution. For more information, see *4.25 Loop suspend counter register, PMC100_LSCR* on page 4-86.
- The loop start program register, PMC100_LSPR, is used with all loop operations and it holds the program location of the start of the loop. For more information, see *4.23 Loop start program register, PMC100_LSPR* on page 4-83
- The program counter register, PMC100_PCR, is used to hold the location of the microcode instruction being executed. For more information, see *4.10 Program counter register, PMC100_PCR* on page 4-60.

For more information on the instruction OP field, see *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77

For more information on the March SRAM and memory protection test algorithms, see:
- *B.2 March C- algorithm* on page Appx-B-113
- *Appendix C On-line MBIST Memory Protection Logic Test Algorithms* on page Appx-C-114

3.4.1 Loop end behavior

All loop operations function in a similar way, except for their loop end behavior.

The following table describes the loop end behavior.

Table 3-2 Loop end behavior

Loop operation	Loop end behavior
LOOP-Last	When the loop end execution stops, indicates the last instruction in a microcode program and must always be present.
LOOP-LAL	When the loop ends, the address registers, PMC100_RADDR.RA and PMC100_CADDR.CA, are loaded with the PMC100_LOWADDR register value, the PMC100_LSPR register is loaded with the location of the next instruction and PMC100_CTRL.ADDRID is set to `0b1` (increment).
LOOP-LAH	When the loop ends, the address registers PMC100_RADR.RA and PMC100_CADDR.CA, are loaded with the PMC100_HIGHADDR register value, the PMC100_LSPR register is loaded with the location of the next instruction and PMC100_CTRL.ADDRID is set to `0b0` (decrement).
LOOP-LCR	When the loop ends, PMC100_LCR.LC is loaded with PMC100_LCR.LCI and the PMC100_LSPR register is loaded with the location of the next instruction.

3.4.2 Loop operation execution

Loop operations consist of LOOP-LAL, LOOP_LAH, LOOP-LCR, and LOOP_Last operations.

The loop operations execute as follows:

Procedure

1. For LOOP-LAL and LOOP-LAH operations, the memory address is compared to determine if the end condition is TRUE.

   a. If PMC100_CTRL.ADDRID is `0b1` (increment) and if the current address stored in PMC100_RADDR.RA and PMC100_CADDR.CA registers fields is equal to the PMC100_HIGHADDR register value, the loop ends.
   b. If PMC100_CTRL.ADDRID is `0b0` (decrement) and if the current address stored in PMC100_RADDR.RA and PMC100_CADDR.CA registers fields is equal to the PMC100_LOWADDR register value, the loop ends.
2. For LOOP-Last operations, when PMC100_CTRL.BAMEN is 0, the memory address is compared to determine if the end condition is TRUE:
   a. If PMC100_CTRL.ADDRID is set to `0b1` (increment), the loop ends if the current address stored in the PMC100_RADDR.RA and PMC100_CADDR.CA registers fields is equal to the PMC100_HIGHADDR register value.
   b. If PMC100_CTRL.ADDRID is set to `0b0` (decrement), the loop ends if the current address stored in the PMC100_RADDR.RA and PMC100_CADDR.CA registers fields is equal to the PMC100_LOWADDR register value.
3. For LOOP-LCR operations, if PMC100_LCR.LC is equal to 0, the loop ends. Otherwise, PMC100_LCR.LC is decremented by 1.
4. For LOOP-Last operations, when PMC100_LCR.LLEN is 1 and if PMC100_LCR.LC is equal to 0, the loop ends. Otherwise, PMC100_LCR.LC is decremented by 1.
5. For LOOP-Last operations, when PMC100_CTRL.BAMEN is 1 and PMC100_P.UA is 1 and if PMC100_CADDR.CA is equal to PMC100 _ CADDR.BNK_END, 1 ' b0, the loop ends. Otherwise, PMC100_CADDR.CA is decremented by 1.
6. For all LOOP operations, the registers are updated as follows:
   a. If the loop is not ending, the PMC100_PCR.PC field is loaded with the PMC100_LSPR.LS field value and execution continues at the start of the loop. Also, if the PMC100_P.UA field is 1 and PMC100_CTRL.BAMEN is 0, then the current address stored in the PMC100_RADDR.RA and PMC100_CADDR.CA fields are incremented or decremented as specified by the PMC100_CTRL.ADDRID bit.
   b. If the loop is ending and the loop operation is not LOOP-Last, then the PMC100_PCR.PC field is incremented by 1, the registers are updated according to the description in *Table 3-2 Loop end behavior* on page 3-24 and execution continues to the next loop.
7. For all LOOP operations, execution either continues or suspends according to the PMC100_CTRL.TCSEN, PMC100_CTRL.TCCEN, PMC100_LSCR.LCSEN, and PMC100_CTRL.BAMEN bit values. Normally these bits are programmed to 0 when executing march algorithms and so execution continues until the loop end condition is true.
8. For LOOP-Last operations only, execution either continues or suspends according to the PMC100_CTRL.TCSEN, PMC100_CTRL.TCCEN, and PMC100_CTRL.EXECO, and PMC100_CTRL.BAMEN values. These bits are usually programmed to 0 when executing March algorithms. Therefore, execution continues until the loop end condition is TRUE. For more information on the state machine, see *4.5.1 PMC-100 state machine* on page 4-46.
9. For LOOP-Last operations only, execution either continues or stops. For more information on the state machine, see *4.5.1 PMC-100 state machine* on page 4-46.

3.5 APB slave interface

The APB slave interface is used to program PMC-100, allowing it to be used by a processor core in wich it is instantiated for self-test or by an external processor to perform a test of the core.

The interface complies with the *AMBA® APB Protocol Version 2.0 Specification* is clocked by the **CLK** input and implements a clock enable input which allows it to be used with an interconnect that is clocked by a slower semi-synchronous clock. PMC-100 performs APB transfers with no wait states, except when software read triggered execution is used, see CTRL.SRTEEN.

Clock enable signal

The **PCLKEN** clock enable input signal allows the APB interface to operate with an N:1 semi-synchronous bus clocking scheme. If the bus is opening at thePMC-100 CLK clock frequency, then 1:1 clocking is used and so PCLKEN must be tied HIGH. The APB interface logic is driven by PMC-100 clock signal, **CLK**. PMC-100 can be connected to an APB bus which is clocked with the same clock or integer division, for example, 3:1 clocking. The clock enable signal, **PCLKEN**, must be used to indicate the relationship between the clock and the bus clock. If the bus is clocked by **PCLK**, then **PCLKEN** must be asserted in the cycle before every **PCLK** rising edge. It is important that the relationship between **PCLK** and **PCLKEN** is maintained.

The following figure shows a timing example in which the CLK:PCLK frequency ratio is 3:1. If the APB interface is to be connected to a bus which is clocked asynchronously to the PMC-100 clock, a synchronizing bridge component must be used to connect the bus to PMC-100.

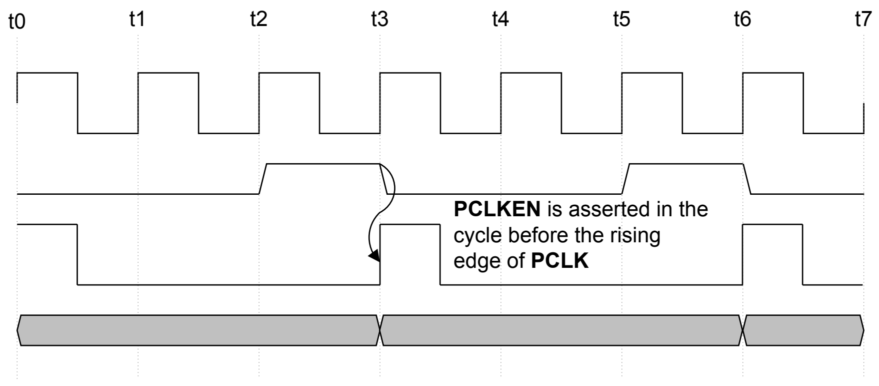

t0
t1
t2
t3
t4
t5
t6
t7
PCLKEN is asserted in the cycle before the rising edge of **PCLK**

Figure 3-2 PCLKEN with CLK:PCLK ratio 3:1

3.6 Reset behavior

PMC-100 is reset by asserting **nSYSRESET** for at least two clock cycles. PMC-100 does not contain a reset synchronizer and must be connected to the synchronized IP core warm reset signal.

Primary control state of PMC-100 is initialized during reset but most of the programmers model registers are not initialized and must be initialized by software, see *4.4 PMC-100 programming* on page 4-37.

3.7 Clock gating

PMC-100 contains an architectural clock gate that generates an internal clock from the input clock signal, **CLKIN**. The clock is automatically enabled when there are transactions on the APB slave interface or when the CTRL.PEEN bit is 1.

3.8 Denial of Service

PMC-100 is a simple IP component, therefore, there are no concerns about denial of service. There are however potential concerns relating to the IP core that is tested by PMC-100. For example, if PMC-100 is programmed incorrectly, it could enter an infinite loop which would permanently lock a memory, preventing progress of the core.

Chapter 4
PMC-100 programmers model

This chapter describes the PMC-100 registers and provides more information on programming the processor.

It contains the following sections:

4.1 PMC-100 register memory map

The PMC-100 register memory map contains CoreSight and control registers. The CoreSight registers are read-only (except for the PMC100_LAR register), and the control registers are read/write.

The following figure shows the PMC-100 register memory map.

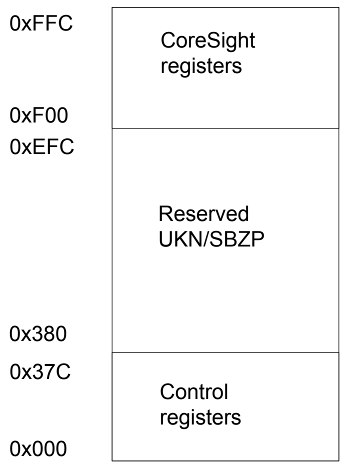


Figure 4-1 PMC-100 register memory map

4.2 PMC-100 register access overview

The PMC-100 software-programmable registers are accessed through the APB slave interface, and occupy a 4KB region. PMC-100 also contains CoreSight registers.

The following information applies to all PMC-100 registers.

- Do not attempt to access reserved or unused address locations. Attempting to access these locations can result in UNPREDICTABLE behavior.
- Unless stated:
  - — Do not modify UNDEFINED register bits.
  - — Ignore UNDEFINED register bits on reads.
  - — All register bits are reset to 0 by the reset signal.
  - — All implemented and non-reserved bits and fields can be written to any value by software.
- The access types used in this chapter are:

  RW

  Read/write

  RO

  Read-only

  WO

  Write-only

  UNK

  UNKNOWN for reads

  SBZP

  Should be zero or preserve for writes

  RAZ

  Read as zero

  RAO

  Read as one

  WI

  Writes ignored

Reserved register bits are implemented as RAZ/WI and software must use them as UNK/SBZP. This approach minimizes the effect on software if new register bits are added in future revisions of PMC-100.

The following rules apply when accessing the registers:

- If implemented by an IP core, only privileged and Secure accesses are supported. If an IP core does not support Secure accesses, then all accesses are indicated as Secure on the APB interface. To hide potentially sensitive data from User mode code, the effect of non-privileged or Non-secure accesses are as follows:
  - — Read accesses to the CoreSight registers returns the register value as normal and a non-error response on the **PSLVERR** signal.
  - — Read accesses to control registers and the reserved region returns 0 data and an error response on the **PSLVERR** signal.
  - — Write accesses to the CoreSight registers are ignored and returns a non-error response on the **PSLVERR** signal.
  - — Write accesses to control registers and the reserved region is ignored and returns an error response on the **PSLVERR** signal.

Note

A processor implementation might generate a BusFault exception or treat the register accesses as RAZ/WI if registers are accessed in Unprivileged or Non-secure state. For more information on how Unprivileged and Non-secure accesses to PMC-100 are handled, see your processor documentation.

- Only word write accesses are supported. Therefore, non-word write accesses return an error response on the **PSLVERR** signal.
- Except for writes to the PMC100_CTRL register, when PMC100_CTRL.PEEN is 1, behavior is UNPREDICTABLE if control registers are written to. Therefore, the control registers must not be written to when PMC100_CTRL.PEEN is 1, except to:
  - — Clear the PMC100_CTRL.PEEN bit, which stops execution.
  - — Clear the PMC100_CTRL.PES bit, which resumes execution.
  - — Write a value to the PMC100_AOR to disable external TC pulse generation.
- When execution is enabled, it is not expected to be useful to read any control registers. The exception is the PMC100_CTRL register which can read to poll PMC-100 to check if a test has successfully completed or failed.

4.3 PMC-100 register summary

The PMC-100 software-programmable registers are accessed through the APB slave interface, and occupy a 4KB region. PMC-100 also contains CoreSight registers.

The following table lists all the PMC-100 registers with their offset in the 4KB region.

Table 4-1 PMC-100 register summary

Offset	Register name	Access type	Description
0x000	PMC100_CTRL	RW	*4.5 Main control register, PMC100_CTRL* on page 4-40
0x004	PMC100_MCR	RW	*4.7 Memory control register, PMC100_MCR* on page 4-53
0x008	PMC100_BER	RW	*4.9 Byte enable register, PMC100_BER* on page 4-59
0x00C	PMC100_PCR	RW	*4.10 Program counter register, PMC100_PCR* on page 4-60
0x010	PMC100_RPR	RO	*4.11 Read pipeline register, PMC100_RPR* on page 4-61
0x014	PMC100_HIGHADDR	RW	*4.13 High address register, PMC100_HIGHADDR* on page 4-64
0x018	PMC100_CADDR	RW	*4.14 Column address register, PMC100_CADDR* on page 4-65
0x01C	PMC100_RADDR	RW	*4.15 Row address register, PMC100_RADDR* on page 4-67
0x020	PMC100_AIR	RW	*4.17 Auxiliary input register, PMC100_AIR* on page 4-71
0x024	PMC100_AOR	RW	*4.18 Auxiliary input register, PMC100_AOR* on page 4-72
0x028	PMC100_MER	RW	*4.19 MBISTOLERR input register, PMC100_MER* on page 4-73
0x02C	PMC100_LSPR	RW	*4.23 Loop start program register, PMC100_LSPR* on page 4-83
0x030	PMC100_LCR	RW	*4.24 Loop counter register, PMC100_LCR* on page 4-84
0x034	PMC100_AR	RW	*4.8 Array register, PMC100_AR* on page 4-56
0x038	PMC100_CFGR	RW	*4.6 MBISTOLCFG output register, PMC100_CFGR* on page 4-52
0x03C	PMC100_TCCR	RW	*4.26 Test continue counter register, PMC100_TCCR* on page 4-88
0x040	PMC100_LOWADDR	RW	*4.12 Low address register, PMC100_LOWADDR* on page 4-63
0x044	PMC100_LCSR	RW	*4.25 Loop suspend counter register, PMC100_LSCR* on page 4-86
0x048-0x07C	-	UNK/SBZP	Reserved
0x080-0x09C	PMC100_X0-PMC100_X7	RW	*4.16 Data registers, PMC100_X0-PMC100_X7 and PMC100_Y0-PMC100_Y7* on page 4-70
0x0A0-0x0FC	-	UNK/SBZP	Reserved
0x100-0x11C	PMC100_Y0-PMC100_Y7	RW	*4.16 Data registers, PMC100_X0-PMC100_X7 and PMC100_Y0-PMC100_Y7* on page 4-70
0x120-0x17C	-	UNK/SBZP	Reserved
0x180-0x19C	PMC100_DM0-PMC100_DM7	RW	*4.20 Data mask, fault bitmap, and data registers, PMC100_DM0-PMC100_DM7* on page 4-74
0x1A0-0x1FC	-	UNK/SBZP	Reserved
0x200-0x21C	PMC100_XM0-PMC100_XM7	RW	*4.21 XOR mask registers, PMC100_XM0-PMC100_XM7* on page 4-76
0x220-0x27C	-	UNK/SBZP	Reserved

Table 4-1 PMC-100 register summary (continued)

Offset	Register name	Access type	Description
0x280-0x2FC	-	UNK/SBZP	Reserved
0x300-0x37C	PMC100_P0-PMC100_P31	RW	*4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77
0x380-0xEFC	-	UNK/SBZP	Reserved
0xF00-0xFFC	CoreSight registers	-	*4.27 CoreSight™ register summary* on page 4-89

4.4 PMC-100 programming

Before testing can start, software must program PMC-100 appropriately.

This includes:

- The attributes for the array under test, for example its array encoding.
- SRAM mux factor.
- Data mask.
- Pipeline depth.
- Cycles per operation.
- The microcode program needed to perform the test algorithm.

PMC-100 can be programmed to interrupt the processor when the test is complete or if a fault is detected. Alternatively, software can poll PMC-100 to see when the test is complete or if a fault is detected.

When PMC-100 programming is complete, microcode execution is initiated by setting the PMC100_CTRL.PEEN bit to 0b1. For more information on PMC100_CTRL, see *4.5 Main control register, PMC100_CTRL* on page 4-40.

All registers must be programmed even if they are not used by a test. Unused registers must be set to zero, including program registers. Also, register bits that are not implemented because of configuration parameter values must also be programmed. Once PMC-100 programming is complete, microcode execution is initiated by setting the PMC100_CTRL.PEEN bit to `0b1`.

This section contains the following subsection:

- *4.4.1 Standard register initialization and programming* on page 4-37.

4.4.1 Standard register initialization and programming

The following table shows the example register programming, where N is the number of elements in the array under test.

Table 4-2 Standard register initialization and programming

Register	Programming
PMC100_CTRL	`0x00000210`. See the table below. BAMEN - `0b0`, bank address mode disabled DMDIS – `0b0`, data masking enabled TCCEN – `0b0`, test continue counter disabled SRTEEN – `0b0`, software read triggered execution disabled TFPCHKUE – `0b0`, uncorrectable error check cleared NOTRANS – `0b0`, no transaction cleared PCHKR – `0b00`, protection error check result cleared PREN – `0b0`, **MBISTOLPREN** disabled FP – `0b00`, fixed data pattern set to all 1s ADDRID – `0b0`, address decrement ADDRCD – `0b1`, x-fast TFSEN – `0b0`, test fail interrupt disabled TF – `0b0`, test fail cleared TESEN – `0b0`, test end interrupt disabled TE – `0b0`, test end cleared STOPF – `0b1`, stop on failure EXECO – `0b0`, execute once disabled TCSEN – `0b0`, TC input ignored PES – `0b0` – program not suspended PEEN – `0b0`, execution disabled
PMC100_MCR	Attributes for array under test
PMC100_BER	`0xFFFFFFFF`
PMC100_PCR	`0x00000000`
PMC100_HIGHADDR	N-1
PMC100_CADDR	`0x00000000`
PMC100_RADDR	`0x00000000`
PMC100_AIR	`0x00000000`
PMC100_AOR	As required by the system
PMC100_MER	`0x00000000`
PMC100_LCR	Loop counter value required by test
PMC100_AR	MBIST array value for the array under test
PMC100_CFGR	`0x00000000`
PMC100_TCCR	Test continue counter initialization value required by test
PMC100_LOWADDR	`0x00000000`

Table 4-2 Standard register initialization and programming (continued)

Register	Programming
PMC100_LSCR	Loop suspend value required by test
PMC100_X0-PMC100_X7	All zero
PMC100_Y0-PMC100_Y7	All zero
PMC100_DM0-PMC100_DM7	Data mask for array under test
PMC100_XM0-PMC100_XM7	XOR mask for array under test
PMC100_P0-PMC100_P31	Microcode to execute test algorithm, unused registers set to zero

4.5 Main control register, PMC100_CTRL

PMC100_CTRL contains the main test control and status bits.

Usage constraints
Software can modify all bit fields except TEN, MBISTACK, and STATE. Additionally, automatic hardware mechanisms can update some bit fields.

Configuration
This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_CTRL bit assignments.

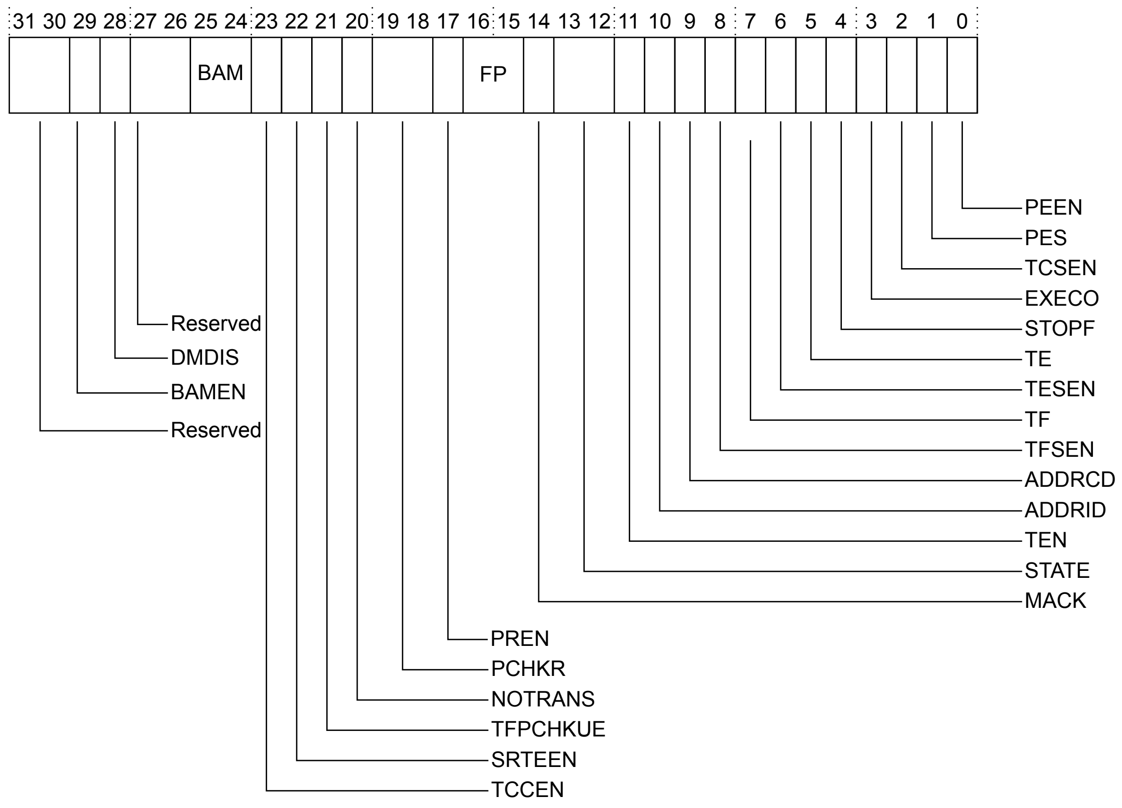


Figure 4-2 PMC100_CTRL bit assignments

The following table describes the PMC100_CTRL bit assignments.

Table 4-3 PMC100_CTRL bit assignments

Field	Name	Type	Reset value	Description
[31:30]	Reserved	-	-	Reserved, RES0
[29]	BAMEN	RW	`0x0`	Bank address mode enable. This bit enables use of the column address register, PMC100_CADDR.CA, as the output address MSBs and or LSBs, which allows PMC100_CADDR.CA to be used to select each bank in turn. For more information, see *4.15.1 Address output value, PMC100_CTRL.BAMEN=0* on page 4-68 and *4.15.2 Address output value, PMC100_CTRL.BAMEN=1* on page 4-68. This mode is only intended to be used with memory protection logic test algorithms. `0b0` Bank address mode disabled `0b1` Bank address mode enabled **Note** This bit also affects the operation of the LOOP-Last OP, PMC100_P.UA and PMC100_LSCR
[28]	DMDIS	RW	`0x0`	Data masking disable. This bit disables data masking of read data using the PMC100_DM register and the fault bitmap functionality. For more information, see *4.20 Data mask, fault bitmap, and data registers, PMC100_DM0-PMC100_DM7* on page 4-74. `0b0` Data masking enabled `0b1` Data masking disabled
[27:26]	Reserved	-	-	Reserved, RES0
[25:24]	BAM	RW	`0x0`	Bank address mode. When BAMEN is b1 , BAM determines the format of the address output on the MBISTOLADDR signal. See section *4.15.1 Address output value, PMC100_CTRL.BAMEN=0* on page 4-68 for further details. • b00 Mode 0 - CADDR.CA address MSB • b01 Mode 1 - CADDR.CA address LSB • b10 Mode 2 - CADDR.CA address MSB and LSB • b11 Reserved
[23]	TCCEN	RW	`0x0`	Test continue counter enable. This enables the internal test continue counter to generate the test continue pulse, see the PMC100_TCCR register. For more information, see *4.26 Test continue counter register, PMC100_TCCR* on page 4-88. The test continue counter can cause a Resume event. TCCEN can be used to enable suspend events. For more information, see *4.5.1 PMC-100 state machine* on page 4-46 `0b0` Test continue counter disabled `0b1` Test continue counter enabled

Table 4-3 PMC100_CTRL bit assignments (continued)

Field	Name	Type	Reset value	Description
[22]	SRTEEN	RW	`0x0`	Software read triggered execution enable. When enabled, this bit causes execution to be triggered when software reads the PMC100_X0 data register. Wait states are inserted on the APB interface until execution is complete and the data register is updated. the new value in the data register is then returned on the bus. `0b0` Software read triggered execution disabled. `0b1` Software read triggered execution enabled. **Note** • A debugger can use this bit to effectively dump memory content. For more information, see *D.3 Memory dumping algorithm* on page Appx-D-133. • The software read triggers execution in a similar way to the test continue counter or **TC** input signal.
[21]	TFPCHKUE	RW	`0x0`	Test fail protection error check result with uncorrectable error. The TFPCHKUE bit is intended to be used with memory scrubbing algorithms. When the PCHKCE operation is executed it causes an uncorrectable error to be treated as a test failure, which sets the TF bit. An uncorrectable error is indicated by **MBISTOLOUTDATA[1]** for reads where **MBISTOLPSEL** is `0b01`. The TFPCHKUE bit encoding is: **0** Uncorrectable error result not treated as test fail when PCHKCE is executed. **1** Uncorrectable error result treated as test fail when PCHKCE is executed **Note** If the memory contains uninitialized entries then TFPCHKUE must not be set to `0b1` when executing memory scrubbing algorithm.
[20]	NOTRANS	RW	0	No MBIST transaction. This bit is used when performing memory scrubbing. It allows conditional execution to be implemented by forcing the **MBISTOLREADEN** and **MBISOLWRITEN** signals LOW, turning the MBIST transactions into NOPs. NOTRANS is automatically set to `0b1` when an instruction is executed with a PCHKCE operation and **MBISTOLOUTDATA[1:0]** is not `0b01`. NOTRANS is automatically cleared to `0b0` when an instruction is executed with a LOOP-Last operation or CLRNT operation. The NOTRANS bit encoding is: `0b0` MBIST transaction performed normally **1** MBIST transaction converted to NOPs
[19:18]	PCHKR	RW	`0x0`	Protection error check result. This is a sticky protection error check result value. This is the **MBISTOLOUTDATA[1:0]** read data value for reads where **MBISTOLPSEL** is `0b01`. The previous PCHKR value is OR-gated with the new value.
[17]	PREN	RW	`0x0`	**MBISTOLPREN** signal control. This enables the IP core error reporting output bus for MBIST read transactions when **MBISTOLPSEL** is `0b01`, which selects the parity or ECC logic error check result. The PREN bit encoding is: `0b0` **MBISTOLPREN** LOW `0b1` **MBISTOLPREN** HIGH for protection check result reads. Otherwise, LOW

Table 4-3 PMC100_CTRL bit assignments (continued)

Field	Name	Type	Reset value	Description

[16:15]	FP	RW	`0x0`	Pattern. This field controls the fixed data pattern used in MBIST write data and expected read data. A fixed pattern is only used when the instruction DREG field is `0b10`. The pattern may be inverted using the instruction DPOL field. The 8-bit data patterns shown below are repeated across the full width of the MBIST data, right justified. The data patterns selected by the FP field encoding are as follows:
				`0b00` `0b11111111`
				`0b01` `0b10101010`
				`0b10` `0b10100101`
				`0b11` Reserved

[14]	MACK	RO	UNKNOWN	Value of the **MBISTOLACK** input signal. For more information, see *E.3 MBIST master interface signals* on page Appx-E-138
[13:12]	STATE	RO	`0x0`	This is the status of PMC-100 state machine. The state encodings are: `0b00` Initial `0b01` Run `0b10` Suspended `0b11` Reserved
[11]	TEN	RO	**TEN** input signal value	Test enable. When the TEN bit is 0, PMC-100 is disabled and cannot be programmed.
[10]	ADDRID	RW	`0x0`	Address increment/decrement control. This bit effects the next PMC100_RADDR.RA and PMC100_CADDR.CA address register values as follows: `0b0` Address registers are decremented `0b1` Address registers are incremented The address registers are updated with the next value when an instruction is executed with a UA bit value of 1. For more information, see *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77.
[9]	ADDRCD	RW	`0x0`	Address change direction. This bit controls the direction that address changes are made with respect to a memory array. It effects the next PMC100_RADDR.RA and PMC100_CADDR.CA address register field values as follows: `0b0` y-fast. PMC100_CADDR.RA is changed first. All PMC100_CADDR.CA values are accessed before the PMC100_RADDR.RA value is changed `0b1` x-fast. PMC100_RADDR.RA is changed first. All PMC100_RADDR values are accessed before the PMC100_CADDR.CA value is changed The address registers are updated with the next value when an instruction is executed with a UA bit value of 1. For more information, see *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77.

Table 4-3 PMC100_CTRL bit assignments (continued)

Field	Name	Type	Reset value	Description
[8]	TFSEN	RW	`0x0`	Test failed signal enable. Controls the behavior of the **TF** signal. For more information, see *E.4 Execution control and status signals* on page Appx-E-139. The values can be: `0b0` **TF** signal is held LOW `0b1` **TF** signal is equal to the value of the TF bit
[7]	TF	RW	`0x0`	Test failed status bit. The values can be: `0b0` Test has not failed `0b1` Test has failed TF is automatically set to 1 if a data comparison in the test program fails. TF can only be cleared to 0 by software.
[6]	TESEN	RW	`0x0`	Test ended signal enable. Controls the behavior of the **TE** signal. For more information, see *E.4 Execution control and status signals* on page Appx-E-139. The values can be: `0b0` **TE** signal is held LOW `0b1` **TE** signal is equal to the value of the TE bit
[5]	TE	RW	`0x0`	Test ended status bit. The values can be: `0b0` Test has not ended `0b1` Test has ended TE is automatically set to 1 when the execution stop event occurs. For more information, see *Resume event* on page 4-48. TE can only be cleared to 0 by software
[4	STOPF	RW	`0x0`	Stop on failure. This bit causes execution to stop when a failure is detected and can take any of the following values: `0b0` Stop on failure mode is disabled `0b1` Stop on failure mode is enabled When STOPF is `0b1` and a data check in the test program fails, the PEEN bit is automatically cleared to `0b0` and the TF bit is set to `0b1`, halting execution at the current instruction. The TE bit is not changed in this case. As data checks occur concurrently with program execution, the value of the PC depends on the instructions that follow the failing read and the value of the PMC100_MCR.PD value. For more information, see *4.6 MBISTOLCFG output register, PMC100_CFGR* on page 4-52. The PMC100_RPR register can be used to determine which read transaction failed. For more information, see *4.11 Read pipeline register, PMC100_RPR* on page 4-61
[3]	EXECO	RW	`0x0`	Execute once. This bit has the following functions: `0b0` Execute once mode is disabled `0b1` Execute once mode is enabled If EXECO is `0b1` and an instruction is executed with a LOOP-Last OP field value, the PEEN bit is automatically cleared to 0 and the TE bit is set to 1. Therefore, this causes instructions to be executed only once.

Table 4-3 PMC100_CTRL bit assignments (continued)

Field	Name	Type	Reset value	Description
[2]	TCSEN	RW	`0x0`	Test continue signal enable. The **TC** input signal can cause a Resume event. **TCSEN** can be used to enable Suspend events. This bit has the following functions: `0b0` **TC** signal is ignored `0b1` Enable **TC** input signal For more information, see *Resume event* on page 4-48 and *Suspend event* on page 4-48 . For more information, see *E.4 Execution control and status signals* on page Appx-E-139

Table 4-3 PMC100_CTRL bit assignments (continued)

Field	Name	Type	Reset value	Description
[1]	PES	RW	`0x0`	Program execution suspended. This bit indicates when execution is suspended and its encoding is as follows: `0b0` Program execution can take place. `0b1` Program execution suspended. PES is automatically set to `0b1` when an event occurs. For more information, see *Start_s event* on page 4-47. PES is automatically cleared to `0b0` when a Resume event occurs. For more information, see *Resume event* on page 4-48. It is also possible for software to cause a Resume event to occur when execution is suspended by clearing PES to `0b0`. **Note** If PEEN is `0b1` software must not set PES to `0b1` because this might cause microcode execution to suspend prematurely and corrupt the target array.
[0]	PEEN	RW	`0x0`	Program execution enable. This is the main execution enable bit and its function is as follows: **0** Program execution is disabled. **1** Program execution is enabled. Program execution takes place if PEEN is `0b1` and PES is `0b0`. PEEN is automatically to `0b0` at the end of a test. For more information, see *Resume event* on page 4-48. When PEEN is `0b0`, the **MBISTOLREQOL** signal is driven LOW. When software sets the PEEN bit to `0b1`, the internal state is cleared, including PMC100_RPR. For more information, see *4.11 Read pipeline register, PMC100_RPR* on page 4-61 **Note** • The PEEN bit must be `0b0` before software changes any register values, except the PMC100_CTRL register when clearing the PEEN bit to `0b0` or the PES bit to `0b0`. • Under normal circumstances, when programming is complete and PEEN is 0, software starts execution by causing a Start_r event by setting PEEN to `0b1` and PES to `0b0`. • It is also possible for software to start PMC-100 in the suspended state when PEEN is 0 by causing a start_s event by setting PEEN to `0b1` and PES to `0b1`. • If PEEN is b1 then software must not set it to `0b0`, except when debugging a microcode program. This can cause **MBISTOLREQ** to be de-asserted before **MBISTOLACK** is asserted, which would violate the MBIST interface protocol.

4.5.1 PMC-100 state machine

PMC-100 has a state machine that controls its exdcution state and the current state can be read from the CTRL.STATE field. There are several events that cause the state machine to transition between states.

The following table describes the three execution states.

Table 4-4 PMC-100 execution states

State	Description
Initial	Program is not executing and **MBISTOLREQOL** is LOW.
Run	In *Memory Built-In Self-Test* (MBIST) entry or exit sequence or program is executing.
Suspended	Program execution is suspended and **MBISTOLREQOL** is LOW.

The following diagram shows the state transitions for the state machine.

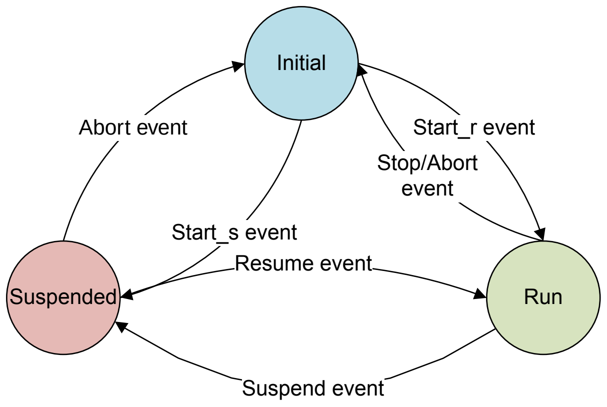


Figure 4-3 PMC100_CTRL bit assignments

Start_r event

When a Start_r event takes place, the following occurs:

- Internal state that software cannot write to is initialized
- The *Memory Built-In Self Test* (MBIST) interface entry sequence is performed
- The Run state is entered
- Program execution starts

The Start_r event is used in the Initial state and is caused when PMC100_CTRL.PEEN is 0 and software writes 1 to PMC100_CTRL.PEEN and 0 to PMC100_CTRL.PES.

Note

- If this event occurs after a test has completed successfully and software has not re-initialized the PMC100_PCR, PMC100_CADDR, or PMC100_RADDR registers, then a stop condition might still be active, and the state machine briefly enters the Run state and then returns to the Initial state. **MBISTOLREQOL** is not asserted.
- This event can be used in other states to force the state machine into the Run state if a deadlock occurs. Arm does not recommend that this behavior is used in normal operation.

Start_s event

When a Start_s event occurs, internal state that is not software writable is initialized and the Suspended state is entered.

This event is used in the Initial state and is caused when PMC100_CTRL.PEEN is 0 and software writes 1 to PMC100_CTRL.PEEN and PMC100_CTRL.PES.

Suspend event

When this event occurs, the *Memory Built-In Self-Test* (MBIST) interface exit sequence is carried out. This event can only occur when the state machine is in the Run state and is caused when any of the following conditions are satisfied:

Table 4-5 Conditions for Suspend event

Condition	PMC100_CTRL	PMC100_LSCR	Instruction execution requirements	Additional notes
Condition 1	• The PEEN bit is 1 • The EXECO bit is 0 • The BAMEN bit is 0 • The TCSEN, TCEEN, or SRTEEN bits are 1	The LSCEN bit is 0	An instruction is executed with the LOOP-Last operation	When this condition occurs, PMC100_CTRL.PES is automatically set to 1
Condition 2	• The PEEN bit is 1 • The TCSEN or TCEEN bits are 1	• The LSCEN bit is 1 • The LSC bit is 0	An instruction is executed with either LOOP-LAL, LOOP-LAH, or LOOP-LCR operations	When this condition occurs, PMC100_CTRL.PES is automatically set to 1
Condition 3	• The PEEN bit is 1 • The BAMEN bit is 0 • The TCSEN or TCEEN bits are 1	• The LSCEN bit is 1 • The LSC bit is 0	An instruction is executed with LOOP-Last operation	When this condition occurs, PMC100_CTRL.PES is automatically set to 1

Note

If a stop and a suspend event occurs at the same time, the stop event takes priority.

Resume event

When this event occurs, the *Memory Built-In Self-Test* (MBIST) interface entry sequence is carried out. This event can only when the state machine is in the Suspended state and is caused by the following conditions:

Table 4-6 Conditions for Resume event

Condition	PMC100_CTRL	PMC100_TCCR	Additional notes
Condition 1	• The PEEN bit is 1 • The PES bit is 0 • The TCSEN bit is 1 • **TC** signal is high	-	When this condition occurs, PMC100_CTRL.PES is automatically set to 0
Condition 2	• The PEEN bit is 1 • The PES bit is 0 • The TCEEN bit is 1	The TCC bit is 0	When this condition occurs, PMC100_CTRL.PES is automatically set to 0 and PMC100_TCCR.TCC is reloaded with PMC100_TCCR.TCCI

Table 4-6 Conditions for Resume event (continued)

Condition	PMC100_CTRL	PMC100_TCCR	Additional notes
Condition 3	• The PEEN bit is 1, and software sets PMC100_CTRL.PEEN to 1 • The PES bit is 1, and software sets the PES bit to 0 • Software sets TCSEN or TCCEN bit to 1	-	-
Condition 4	• The PEEN bit is 1 • The PES bit is 1 • The STREEN bit is 1	-	Software reads PMC100_X0. when this occurs, PMC100_CTRL.PES is automatically set to 0

Stop event

This event occurs when a test ends normally. It causes the *Memory Built-In Self-Test* (MBIST) interface exit sequence to be carried out, PMC100_CTRL.PEEN is set to 0 and PMC100_CTRL.TE is set to 1. This event can only occur in the Run state and is caused by any of the following conditions being satisfied:

Table 4-7 Conditions for Stop event

Condition	PMC100_CTRL	PMC100_LCR	Instruction execution requirements
Condition 1	• The PEEN bit is 1 • The ADDRID bit is 1 • The BAMEN bit is 0	-	An instruction is executed with a LOOP-Last operation field value and the current address is equal to the PMC100_HIGHADDR register
Condition 2	• The PEEN bit is 1 • The ADDRID bit is 0 • The BAMEN bit is 0	-	An instruction is executed with a LOOP-Last operation field value and the current address is equal to the PMC100_LOWADDR register
Condition 3	The PEEN bit is 1	• The LLEN bit is 1 • The LC bit is 0	An instruction is executed with a LOOP-Last operation.
Condition 4	• The PEEN bit is 1 • The BAMEN bit is 1 • The STREEN bit is 1	-	An instruction is executed with a LOOP-Last operation and PMC100_CADDR.CA is {PMC100 _ CADDR.BNK_END . 1 ' b0}.
Condition 5	• The PEEN bit is 1 • The EXECO bit is 1	-	An instruction is executed with a LOOP-Last operation.

Note

After this event the PMC100_PCR register points to the LOOP-Last instruction and the PMC100_CADDR and PMC100_RADDR registers are not updated, even if the instruction's UA bit is 1. If a stop and a suspend event occurs at the same time, the stop event takes priority.

Abort event

This event can occur in either the Run or Suspended states and is used to prematurely abort a test and return to the Initial state. It sets PMC100_CTRL.TE to 1 and if it is caused by a read data check failure then PMC100_CTRL.TF is also set to 1. This event is caused when any of the following conditions are satisfied:

Table 4-8 Conditions for Abort event

Condition	PMC100_CTRL	Additional notes
Condition 1	• The PEEN bit is 1 • The STOPF bit is 1	A read data check fails
Condition 2	• The PEEN bit is 1 • Software writes 0 to the PEEN but and PES bit	-

Note

Execution is stopped immediately when this event occurs and the PMC100_PCR, PMC100_CADDR, PMC100_RADDR, PMC100_RPR, PMC100_LCR, PMC100_LSCR, and PMC100_LSPR registers are not updated. This allows software to determine the failing RAM address and read check microcode instruction that detected a failure. Arm does not recommended that the software cause the abort event to occur when the PMC100_CTRL.MBISTACK is 0 and PMC100_CTRL.STATE is `0b01` because it could violate the MBIST interface protocol if it occurs during MBIST entry.

Program execution modes

The program execution mode is configured by the following PMC100_CTRL bits:

- PEEN
- PES
- EXECO
- TCSEN
- TCCEN
- SRTEEN

The following table summarizes these modes.

Table 4-9 Program execution modes

PEEN	PES	EXECP	TCSEN	TCCEN	SRTEEN	Description
0	X	X	X	X	X	Program is not executed
1	0	0	1	X	X	Program is executes until a Suspend, Stop, or Abort event occurs. Used to suspend execution for certain loop conditions. When suspended, a Resume event is generated when the **TC** input is HIGH or when software writes 0 to the PES bit.
1	0	0	X	1	X	Program is executed until a Suspend, Stop, or Abort event occurs. Used to suspend execution for certain looping conditions. When suspended, a Resume event is generated when PMC100_TCCR.TCC is equal to 0 or when software writes 0 to the PES bit.
1	0	0	X	X	1	Program is executed until a Suspend, Stop, or Abort event occurs. Used to suspend execution for certain looping conditions. When suspended, a Resume event is generated when software reads the PMC100_X0 data register or writes 0 to the PES bit.
1	0	1	X	X	X	Program is executed until a Stop or Abort event occurs. Used to execute instructions once.

Table 4-9 Program execution modes (continued)

PEEN	PES	EXECP	TCSEN	TCCEN	SRTEEN	Description
1	0	0	0	0	0	Program is executed until a Stop or Abort event occurs. Used to loop through all required memory locations or data bits or SRAM banks without suspending.
1	1	X	X	X	X	Program execution is suspended waiting for a Resume or Abort event to occur.

4.6 MBISTOLCFG output register, PMC100_CFGR

PMC100_CFGR sets the value of **MBISTOLCFG**.

Usage constraints

This register can only be modified by software. This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1. For more information, see *4.5 Main control register, PMC100_CTRL* on page 4-40

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_CFGR bit assignments.

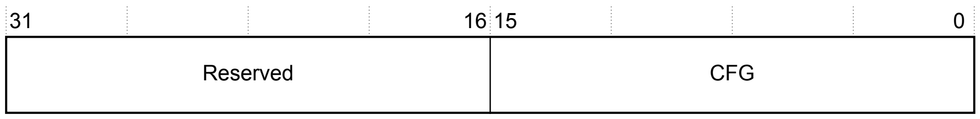


Figure 4-4 PMC100_CFGR bit assignments

The following table describes the PMC100_CFGR bit assignments.

Table 4-10 PMC100_CFGR bit assignments

Field	Name	Type	Reset value	Description
[31:16]	Reserved	UNK/ SBZP	-	Reserved, RES0
[15:0]	CFG	RW	UNKNOWN	This is the **MBISTOLCFG** signal value. For more information, see *E.3 MBIST master interface signals* on page Appx-E-138. ——— **Note** ——— • This value is normally used in production testing to enable MBIST All mode. In some IP cores this is also used to set LATENCY/SETUP controls for the array under test. During on-line MBIST testing, it might be possible to set the LATENCY/ SETUP bits to 1, but the AllMode bits must be set to 0. • Usused field bits are reserved and must be treated as UNK/SBZP. • CFG field width is set by the `MCWIDTH` parameter. For more information, see *3.2 RTL parameters* on page 3-21. ———————

4.7 Memory control register, PMC100_MCR

PMC100_MCR configures the attributes for the memory array under test.

Usage constraints

All fields can be modified by software. This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to. For more information, see *4.5 Main control register, PMC100_CTRL* on page 4-40

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_MCR bit assignments.

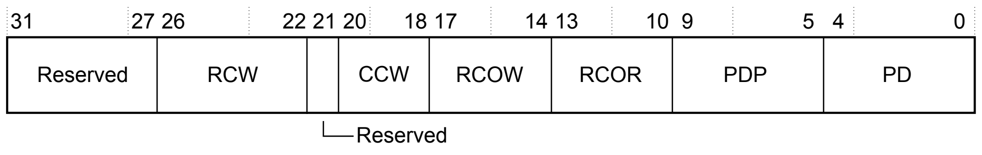


Figure 4-5 PMC100_MCR bit assignments

The following table describes the PMC100_MCR bit assignments.

Table 4-11 PMC100_MCR bit assignments

Field	Name	Type	Reset value	Description
[31:27]	Reserved	UNK/SBZP	-	Reserved, RES0
[26:22]	RCW	RW	UNKNOWN	Row counter width. This must be set to the width of the row section of the RAM under test address bus, which is 2. A value of `0b00000` corresponds to 2 row address bits. **Note** 1. If PMC100_CTRL.BAMEN is 0, the maximum value that this field can be programmed to is MAWIDTH-CCW-2. 2. If PMC100_CTRL.BAMEN is 1, and PMC100_CTRL.BAM is 0 the maximum value that this field can be programmed to is MAWIDTH-CCW-1. 3. If PMC100_CTRL.BAMEN is 1, and PMC100_CTRL.BAM is 1 the maximum value that this field can be programmed to is MAWIDTH-CCW-1. 4. If PMC100_CTRL.BAMEN is 1 and PMC100_CTRL.BAM is 2 the maximum value that this field can be programmed to is MAWIDTH-(2*CCW).
[21]	Reserved	UNK/SBZP	-	Reserved, RES0

Table 4-11 PMC100_MCR bit assignments (continued)

Field	Name	Type	Reset value	Description
[20:18]	CCW	RW	UNKNOWN	Column counter width. When running SRAM tests, this field must be set to the width of the column section of the address bus of the RAM under test. The options are: `0b000` 0 bits, 1 RAM column `0b001` 1 bit, 2 RAM columns `0b010` 2 bits, 4 RAM columns `0b011` 3 bits, 8 RAM columns `0b100` 4 bits, 16 RAM columns `0b101` 5 bits, 32 RAM columns `0b110` Reserved `0b111` Reserved **Note** 1. Memory protection logic testing uses PMC100_CTRL.BAMEN set to 1. 2. If PMC100_CTRL.BAMEN is 0 the maximum value that this field may be programmed to is MAWIDTH-2. 3. If PMC100_CTRL.BAMEN is 1 and PMC100_CTRL.BAM is 0 the maximum value that this field can be programmed to is MAWIDTH-1. 4. If PMC100_CTRL.BAMEN is 1 and PMC100_CTRL.BAM is 1 the maximum value that this field can be programmed to is MAWIDTH-1. 5. If PMC100_CTRL.BAMEN is 1 and PMC100_CTRL.BAM is 2 the maximum value that this field can be programmed to is MAWIDTH/2. 6. If PMC100_CTRL.BAMEN is 1, CCW must be >= 1 because PMC100_CADDR.CA [0] is used as a loop counter and PMC100_CADDR.CA [CCW-1:1] is used as the bank select value in the output address when CCW>1.
[17:14]	RCOW	RW	UNKNOWN	RAM cycles per operation for writes. This must be set to the number of cycles that the RAM under test requires for each access, which is 1. A value of `0b000` corresponds to one cycle per access. **Note** • Unused field bits are reserved and must be treated as UNK/SBZP • RCOW and RCOR widths are set by the `RCOWIDTH` parameter. For more information, see *3.2 RTL parameters* on page 3-21.

Table 4-11 PMC100_MCR bit assignments (continued)

Field	Name	Type	Reset value	Description
[13:10]	RCOR	RW	UNKNOWN	RAM cycles per operation for reads. This must be set to the number of cycles that the RAM under test requires for each access, which is 1. A value of `0b000` corresponds to one cycle per access. **Note** • When using the SRAM test algorithms that issue MBIST read and write operations in the same cycle, RCOR and RCOW must be set to the same value. • Unused field bits are reserved and must be treated as UNK/SBZP • RCOW and RCOR widths are set by the `RCOWIDTH` parameter. For more information, see *3.2 RTL parameters* on page 3-21.
[9:5]	PDP	RW	UNKNOWN	Pipeline depth for protection logic. This is the number of pipeline stages in the MBIST path from **MBISTOLADDR** to **MBISTOLOUTDATA** for the protection logic associated with the memory under test, for transactions where **MBISTOLPSEL** is not `0b00`. For example, if there are five pipeline stages for corrected data reads then this field must be set to 5. **Note** • PDP must be greater than or equal to PD. • Unused field bits are reserved and must be treated as UNK/SBZP • PD and PDP wdths are set by the `PDWIDTH` parameter. For more information, see *3.2 RTL parameters* on page 3-21.
[4:0]	PD	RW	UNKNOWN	Pipeline depth. This is the number of pipeline stages in the MBIST path from **MBISTOLADDR** to **MBISTOLOUTDATA** for the memory under test, for transactions where **MBISTOLPSEL** is `0b00`. For example, if there are three pipeline stages, then this field must be set to 3. **Note** • Unused field bits are reserved and must be treated as UNK/SBZP • PD and PDP wdths are set by the `PDWIDTH` parameter. For more information, see *3.2 RTL parameters* on page 3-21.

4.8 Array register, PMC100_AR

PMC_AR controls the value of the **MBISTOLARRAY** output signal.

The **MBISTOLARRAY** signal is driven by the PMC100_AR register depending on the PMC100_P.PSEL value of the current instruction, see *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77 and if the access is a read or a write, see *Table 4-13 PSEL and MBISTOLARRAY values* on page 4-57.

The **MBISTOLARRAY** value is divided into two sections, the lower section contains the memory controller and sub-array encoding and the upper section contains the protection logic unit encoding. The lower section is always driven by the PMC100_AR.ARR field and the upper section is driven by the PMC100_AR.ARD and PMC100_AR.ARC fields, depending on whether the MBIST transaction is read or write and the microcode instruction PSEL field value. The PMC100_AR.ARD and PMC100_AR.ARC fields are two bits wide and these drive **MBISTOLARRAY[MARWIDTH-1:MARWIDTH-2]** depending on the microcode value for the current instruction.

The protection logic section of the **MBISTOLARRAY** value might be one or two bits wide:
- If it is one bit wide, then the lower bit of the PMC100_AR.ARD and PMC100_AR.ARC fields must be programmed to the MSB of the **MBISTOLARRAY** and the upper bit of the PMC100_AR.ARD and PMC100_AR.ARC fields must be programmed to select the required protection logic unit within the target array.
- If it is two bits wide, then both bits of the PMC100_AR.ARD and PMC100_AR.ARC fields must be programmed to select the required protection logic unit within the target array. In this case, PMC100_AR.ARD is normally set to `0b00`.

Usage constraints

All fields can only be modified by software. This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1. For more information, see *4.5 Main control register, PMC100_CTRL* on page 4-40

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_AR bit assignments.

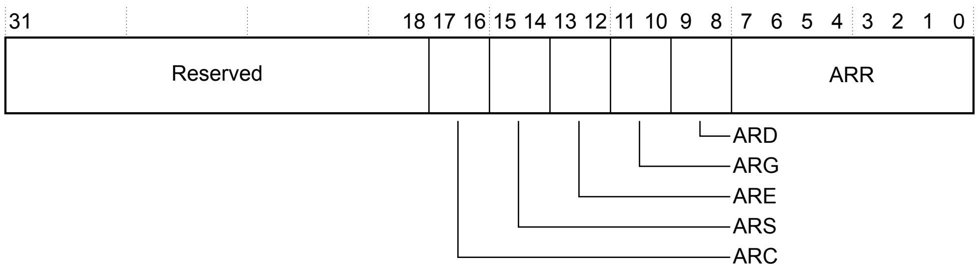


Figure 4-6 PMC100_AR bit assignments

The following table describes the PMC100_AR bit assignments.

Table 4-12 PMC100_AR bit assignments

Field	Name	Type	Reset value	Description
[31:18]	Reserved	UNK/ SBZP	-	Reserved, RES0
[17:16]	ARC	RW	UNKNOWN	Protection logic array field for ECC correction data accesses. MBISTOLARRAY[MARWIDTH-1:MARWIDTH-2] signal value, for reads when PMC100_P.PSEL is 11. For more information, see *E.3 MBIST master interface signals* on page Appx-E-138.
[15:14]	ARS	RW	UNKNOWN	Protection logic array field for *Error Correcting Code* (ECC) syndrome or parity accesses. **MBISTOLARRAY[MARWIDTH-1: MARWIDTH-2]** signal value, for reads when PMC100_P.PSEL is `0b10`. For more information, see *E.3 MBIST master interface signals* on page Appx-E-138.
[13:12]	ARE	RW	UNKNOWN	Protection logic array field for ECC or parity error check accesses. **MBISTOLARRAY[MARWIDTH-1: MARWIDTH-2]** signal value, for reads when PMC100_P.PSEL is `0b01`. For more information, see *E.3 MBIST master interface signals* on page Appx-E-138.
[11:10]	ARG	RW	UNKNOWN	Protection logic array field for ECC or parity generation logic accesses. **MBISTOLARRAY[MARWIDTH-1: MARWIDTH-2]** signal value, for writes when PMC100_P.PSEL is `0b01`. For more information, see *E.3 MBIST master interface signals* on page Appx-E-138.
[9:8]	ARD	RW	UNKNOWN	Protection logic array field for direct SRAM accesses. **MBISTOLARRAY[MARWIDTH-1: MARWIDTH-2]** signal value, for reads and writes when PMC100_P.PSEL is `0b00`. For more information, see *E.3 MBIST master interface signals* on page Appx-E-138. **Note** If the MBISTOLARRAY protection logic unit encoding section is two bits wide, then this field is usually set to `0b00`.
[7:0]	ARR	RW	UNKNOWN	**MBISTOLARRAY[MARWIDTH-3:0]** signal value. This contains the array memory encoding and sub-array encoding fields of the array value. **Note** • Unused field bits are reserved and must be treated as UNK/SBZP • ARR width is set by the `MARWIDTH` parameter-2. For more information, see *3.2 RTL parameters* on page 3-21

The following table shows the **PSEL** and **MBISTOLARRAY** values.

Table 4-13 PSEL and MBISTOLARRAY values

PSEL	Access type	MBISTOLARRAY		Description
		Bits	Value	
`0b00`	Read and write	**[MARWIDTH-1:MARWIDTH-2]**	ARD	Direct SRAM access, protection logic bypassed.
		[MARWIDTH-3:0]	ARR	

Table 4-13 PSEL and MBISTOLARRAY values (continued)

PSEL	Access type	MBISTOLARRAY		Description
		Bits	**Value**	
`0b01`	Write	**[MARWIDTH-1:MARWIDTH-2]**	ARG	ECC or parity generation logic path
		[MARWIDTH-3:0]	ARR	
`0b01`	Read	**[MARWIDTH-1:MARWIDTH-2]**	ARE	ECC or parity error check result.
		[MARWIDTH-3:0]	ARR	
`0b10`	Read	**[MARWIDTH-1:MARWIDTH-2]**	ARS	ECC syndrome or parity value
		[MARWIDTH-3:0]	ARR	
`0b11`	Read	**[MARWIDTH-1:MARWIDTH-2]**	ARC	ECC correction data value
		[MARWIDTH-3:0]	ARR	
`0b10`, `0b11`	Write	**[MARWIDTH-1:MARWIDTH-2]**	ARD	Reserved
		[MARWIDTH-3:0]	ARR	

4.9 Byte enable register, PMC100_BER

PMC100_BER sets the value of the **MBISTOLBE** signal.

Usage constraints

All fields can only be modified by software. This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1. For more information, see *4.5 Main control register, PMC100_CTRL* on page 4-40

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_BER bit assignments.

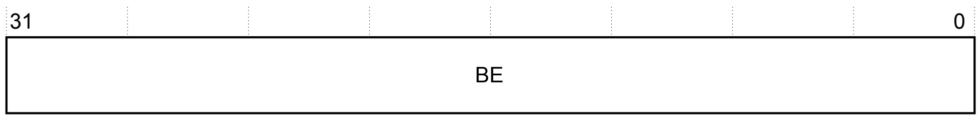


Figure 4-7 PMC100_BER bit assignments

The following table describes the PMC100_BER bit assignments.

Table 4-14 PMC100_BER bit assignments

Field	Name	Type	Reset value	Description
[31:0]	BE	RW	UNKNOWN	**MBISTOLBE** signal value. For more information, see *E.3 MBIST master interface signals* on page Appx-E-138 **Note** • Unused bit fields are reserved and must be treated as UNK/SBZP. • BE width is set byt the `MBWIDTH` parameter. For more information, see *3.2 RTL parameters* on page 3-21

4.10 Program counter register, PMC100_PCR

PMC100_PCR contains the program counter field.

Usage constraints

All fields can only be modified by software. This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1. For more information, see *4.5 Main control register, PMC100_CTRL* on page 4-40

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_PCR bit assignments.

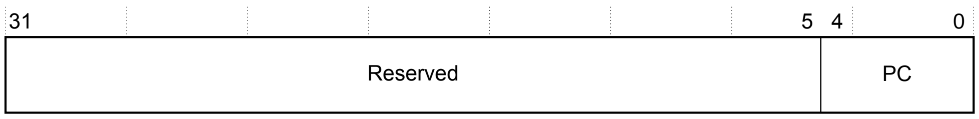


Figure 4-8 PMC100_PCR bit assignments

The following table describes the PMC100_PCR bit assignments.

Table 4-15 PMC100_PCR bit assignments

Field	Name	Type	Reset value	Description
[31:5]	Reserved	UNK/ SBZP	-	Reserved, RES0
[4:0]	PC	RW	UNKNOWN	Program counter. This field points to the program register that is executed. PC is automatically incremented when each instruction is executed. It is automatically loaded with the LPSR value when a loop operation is executed and its end condition is not true. For more information, see *3.4 Loop operations* on page 3-24 **Note** • Unused field bits are reserved and must be treated as UNK/SBZP • PC width is set by the $\log_2$`PROGSIZE`. For more information, see *3.2 RTL parameters* on page 3-21

4.11 Read pipeline register, PMC100_RPR

If a memory fault or a protection logic fault is detected and the PMC100_CTRL.STOPF bit is 1, then PMC100_RPR allows software to determine which read instruction the fault relates to and therefore the memory address containing the fault.

Usage constraints
All fields can be modified by software.

Configuration
This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_RPR bit assignments.

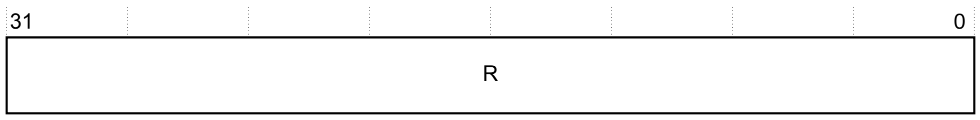


Figure 4-9 PMC100_RPR bit assignments

The following table describes the PMC100_RPR bit assignments.

Table 4-16 PMC100_RPR bit assignments

Field	Name	Type	Reset value	Description
[31:0]	R	RO	UNKNOWN	This is the current value of the read pipeline register in the execution unit. When a read is executed R[0] is set to 1 and R is shifted left every clock cycle. When a data comparison fails the corresponding read has reached R[PMC100_MCR.PD] or R[PMC100_MCR.PDP]. Only R[PMC100_MCR.PD:0] or R[PMC100_MCR.PDP:0] bits are valid, depending on the read operations in the pipeline, and the unused MSBs are set to 0. To determine which read instruction failed: 1. Count the number of bits set in the R field. 2. Count back this number of read instructions, from the instruction pointed to by the PMC100_PCR.PC. Using the example microcode in *A.3.1 Microcode* on page Appx-A-107, if the test stopped and PMC100_PCR.PC = 8 (row 9) and R had 2 bits set, then the failing read was instruction 4 (row 5). The failing address can be determined from the PMC100_CADDR.CA, PMC100_RADDR.RA, PMC100_CTRL.ADDRID, PMC100_CTRL.ADDRCD and Px.UA values of the instructions executed since the failing read. See *4.15.1 Address output value, PMC100_CTRL.BAMEN=0* on page 4-68 and *4.15.2 Address output value, PMC100_CTRL.BAMEN=1* on page 4-68 for more information on how **MBISTOLOUTADDR** is calculated. **Note** • Unused field bits are reserved and must be treated as UNK/WI • R width is $2^{\texttt{PDWIDTH}}$. For more information, see *3.2 RTL parameters* on page 3-21

4.12 Low address register, PMC100_LOWADDR

PMC100_LOWADDR contains the low or minimum address for the memory array under test.

Usage constraints

This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1. All fields can be modified by software.

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_LOWADDR bit assignments.

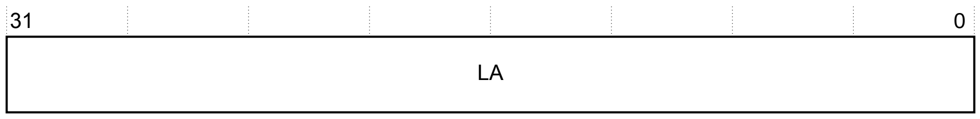


Figure 4-10 PMC100_LOWADDR bit assignments

The following table describes the PMC100_LOWADDR bit assignments.

Table 4-17 PMC100_LOWADDR bit assignments

Field	Name	Type	Reset value	Description
[31:0]	LA	RW	UNKNOWN	Low address. It is used by LOOP operations to check if the end of the loop has been reached. When the current address update mode is increment, the current address is compared against PMC100_LOWADDR. For more information, see *4.15.1 Address output value, PMC100_CTRL.BAMEN=0* on page 4-68 and *4.15.2 Address output value, PMC100_CTRL.BAMEN=1* on page 4-68. **Note** • Unused field bits are reserved and must be treated as UNK/WISBZP • The width of LA is set by the `MAWIDTH` parameter, see *3.2 RTL parameters* on page 3-21. The MSBs are unused depending on this parameter.

4.13 High address register, PMC100_HIGHADDR

PMC100_HIGHADDR contains the high or maximum address for the memory array under test.

Usage constraints

This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1. All fields can be modified by software.

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_HIGHADDR bit assignments.

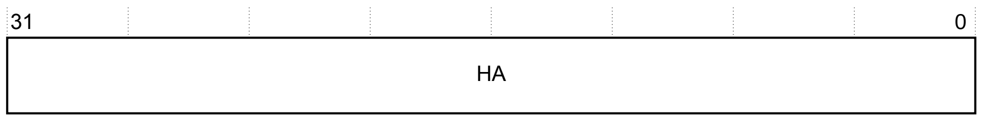


Figure 4-11 PMC100_HIGHADDR bit assignments

The following table describes the PMC100_HIGHADDR bit assignments.

Table 4-18 PMC100_HIGHADDR bit assignments

Field	Name	Type	Reset value	Description
[31:0]	HA	RW	UNKNOWN	High address. It is used by LOOP operations to check if the end of the loop has been reached. When the current address update mode is increment, the current address is compared against PMC100_HIGHADDR. For more information, see *4.15.1 Address output value, PMC100_CTRL.BAMEN=0* on page 4-68 and *4.15.2 Address output value, PMC100_CTRL.BAMEN=1* on page 4-68. **Note** • Unused field bits are reserved and must be treated as UNK/WISBZP • The width of HA is set by the `MAWIDTH` parameter, see *3.2 RTL parameters* on page 3-21. The MSBs are unused depending on this parameter.

4.14 Column address register, PMC100_CADDR

PMC100_CADDR contains column address register field.

Usage constraints

This register can only be modified by software and is automatically updated. This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_CADDR bit assignments.

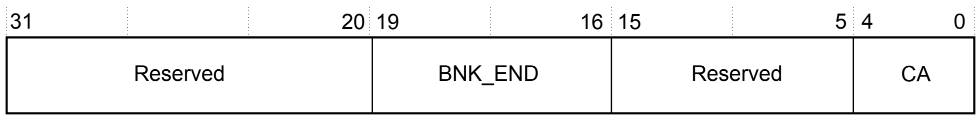


Figure 4-12 PMC100_CADDR bit assignments

The following table describes the PMC100_CADDR bit assignments.

Table 4-19 PMC100_CADDR bit assignments

Field	Name	Type	Reset value	Description
[31:20]	Reserved	-	-	Reserved, RES0
[19:16]	BNK_END	RW	UNKNOWN	Bank end. This field contains the value that is compared against the CA value to determine when a LOOP-Last loop ends when PMC100_CTRL.BAMEN is 1, See section *3.4 Loop operations* on page 3-24 for more information. **Note** The loop ends when CA={BNK_END, 1'b0}
[15:5]	-	UNK/ SBZP	UNKNOWN	Reserved, RES0
[4:0]	CA	RW	UNKNOWN	Column address. This field contains the current value of the column bits of the memory address value. It is updated when an instruction is executed with the UA bit set to 1, see *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77, depending on the values of the PMC100_CTRL register ADDRID and ADDRCD bits, see *4.5 Main control register, PMC100_CTRL* on page 4-40. If PMC100_MCR.CCW is 0 then PMC100_CADDR.CA is not used in the array address and is not automatically updated. If PMC100_MCR.CCW is > 0 then only CA[PMC100_MCR.CCW-1:0] bits are used and the remaining MSBs are cleared to 0 when an instruction is executed with the UA bit set to 1. For more information on calculating the output address, see For more information, see *4.15.1 Address output value, PMC100_CTRL.BAMEN=0* on page 4-68 and *4.15.2 Address output value, PMC100_CTRL.BAMEN=1* on page 4-68.

Note

1. When software writes to this register only bits CA[PMC100_MCR.CCW-1:0] are updated and the remaining MSBs are cleared to 0. If PMC100_MCR.CCW is 0 then all CA bits will be automatically cleared to 0 when software writes to this register.
2. When PMC100_CTRL.BAMEN is 1 and PMC100_MCR.CCW>1, CA[PMC100_MCR.CCW-1:1] is used as the MSBs or LSBs of the MBIST address, depending on the PMC100_CTRL.BAM value.
3. When an instruction is executed with a LOOP-Last OP and PMC100_CTRL.BAMEN is 1 and PMC100_P.UA is 1, CA is decremented by 1 if it is not equal to {BNK_END, 1'b0}.
4. When an instruction is executed with a PUP OP and PMC100_CTRL.BAMEN is 1 and PMC100_P.UA is 1, the CA value is not changed. Therefore, in this case PMC100_P.UA is ignored.

4.15 Row address register, PMC100_RADDR

PMC100_RADDR contains row address register field.

Usage constraints

This register can be modified by software and is automatically updated. This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40. Arm recommends that the PMC100_MCR.RCW field is set correctly for the memory under test before writing to this register.

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_RADDR bit assignments.

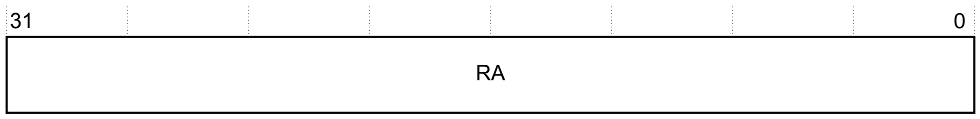


Figure 4-13 PMC100_RADDR bit assignments

The following table describes the PMC100_RADDR bit assignments.

Table 4-20 PMC100_RADDR bit assignments

Field	Name	Type	Reset value	Description
[31:0]	RA	RW	UNKNOWN	Row address. This field contains the current value of the row bits of the memory address value. It is updated when an instruction is executed with the UA bit set to 1, see *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77. depending on the values of the PMC100_CTRL register ADDRID and ADDRCD bits *4.5 Main control register, PMC100_CTRL* on page 4-40. Only the RA[PMC100_MCR.RCW+1:0] bits are used and the remaining MSBs are cleared to 0 when an instruction is executed with the UA bit set to 1. For more information, see *4.15.1 Address output value, PMC100_CTRL.BAMEN=0* on page 4-68 and *4.15.2 Address output value, PMC100_CTRL.BAMEN=1* on page 4-68. **Note** • When software writes to this register only bits RA[PMC100_MCR.RCW+1:0] are updated and the remaining MSBs are cleared to 0. • When a PUP OP is executed and PMC100_P.UA is 1, then RA is set to ~RA, see section Program registers, PMC100_P0-PMC100_P31, See section *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77. This takes priority over the normal incrementing or decrementing behavior of RA when PMC100_P.UA is 1. • When an instruction is executed with a LOOP-Last OP and PMC100_CTRL.BAMEN is 1 and PMC100_P.UA is 1, RA is not changed. • Unused field bits are reserved and must be treated as UNK/SBZP. • The width of field RA is set by the `MAWIDTH` parameter, see *3.2 RTL parameters* on page 3-21

4.15.1 Address output value, PMC100_CTRL.BAMEN=0

If PMC100_CTRL.BAMEN is 0 the **MBISTOLADDR** output value is either the current or the next address depending on the value AO bit in the current instruction. The PMC100_RADDR.RA and PMC100_CADDR.CA register fields are combined to form the address. The next value of these registers is determined by the PMC100_CTRL.ADDRID and CTL.ADDRCD bits.

For more information, see:

- *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77
- *4.5 Main control register, PMC100_CTRL* on page 4-40

The following formula, using Verilog syntax, shows how the PMC100_RADDR.RA and PMC100_CADDR.CA register fields are used to generate the address. The current address uses the current values of the PMC100_RADDR.RA and PMC100_CADDR.CA register fields and the next address uses the next values of these registers:

```
if (PMC100_MCR.CCW > 0)
address = {{MAWIDTH-aw{1’b0}}, PMC100_RADDR.RA[PMC100_MCR.RCW+1:0],
PMC100_CADDR.CA[PMC100_MCR.CCW-1:0]}
else
address = {{MAWIDTH-aw{1’b0}}, PMC100_RADDR.RA[PMC100_MCR.RCW+1:0]}
```

Where `aw = PMC100_MCR.RCW+PMC100_MCR.CCW+2`.

To calculate the current address, software must concatenate the PMC100_RADDR.RA and PMC100_CADDR.CA register fields together in the same way as shown.

4.15.2 Address output value, PMC100_CTRL.BAMEN=1

This mode is used in memory protection logic testing. If PMC100_CTRL.BAMEN is 1 the PMC100_CADDR.CA value is used as the address MSBs or LSBs depending on the PMC100_CTRL.BAM[1:0] value. PMC100_CADDR.CA[0] and PMC100_CADDR.CA[PMC100_MCR.CCW-1:1] are used as the inner and outer loop counting mechanism in the protection logic test algorithms respectively. PMC100_CADDR.CA[PMC100_MCR.CCW-1:1] is also used to allow each memory bank in an array to be selected in turn.

For more information, see:

- *Appendix C On-line MBIST Memory Protection Logic Test Algorithms* on page Appx-C-114
- *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77
- *4.5 Main control register, PMC100_CTRL* on page 4-40

Note

When PMC100_CTRL.BAMEN is 1:

1. PMC100_MCR.CCW must be set to the width of the memory array bank select field in the address+1. Hence, SW must program the PMC100_MCR.CCW field a value of 1 or greater.
2. PMC100_CADDR.CA must be set to (Number of banks*2)-1. Hence, SW must program the PMC100_CADDR.CA field to an odd number that is 1 or greater.

When PMC100_CTRL.BAM[1:0] is b00 the address output value is:

```
if (PMC100_MCR.CCW > 1)
  address = {{MAWIDTH-aw{1’b0}}, PMC100_CADDR.CA[PMC100_MCR.CCW-1:1],
PMC100_RADDR.RA[PMC100_MCR.RCW+1:0]}
else
  address = {{MAWIDTH-aw{1’b0}}, PMC100_RADDR.RA[PMC100_MCR.RCW+1:0]}
```

Where aw = PMC100_MCR.RCW+PMC100_MCR.CCW+1

When PMC100_CTRL.BAM[1:0] is b01 the address output value is:

```
if (PMC100_MCR.CCW > 1)
  address = {{MAWIDTH-aw{1’b0}}, PMC100_RADDR.RA[PMC100_MCR.RCW+1:0],
PMC100_CADDR.CA[PMC100_MCR.CCW-1:1]}
```

```
else
  address = {{MAWIDTH-aw{1’b0}}, PMC100_RADDR.RA[PMC100_MCR.RCW+1:0]}
```

Where aw = PMC100_MCR.RCW+PMC100_MCR.CCW+1

When PMC100_CTRL.BAM[1:0] is b10 the address output value is:

```
if (PMC100_MCR.CCW > 1)
  address = {{MAWIDTH-aw{1’b0}}, PMC100_CADDR.CA[PMC100_MCR.CCW-1:1],
PMC100_RADDR.RA[PMC100_MCR.RCW+1:0], PMC100_CADDR.CA[PMC100_MCR.CCW-1:1]}
else
  address = {{MAWIDTH-aw{1’b0}}, PMC100_RADDR.RA[PMC100_MCR.RCW+1:0]}
```

Where aw = PMC100_MCR.RCW+(2*PMC100_MCR.CCW)

4.16 Data registers, PMC100_X0-PMC100_X7 and PMC100_Y0-PMC100_Y7

These are two independent data registers, PMC100_X and PMC100_Y, and their descriptions are the same. The PMC100_X and PMC100_Y register widths are configured to the data width of the MBIST interface. From a software perspective they are split into several 32-bit wide registers, PMC100_X0-PMC100_X7 and PMC100_Y0-PMC100_Y7 as shown in the following table.

Usage constraints

These registers can be modified by software and are automatically updated. These registers must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40. The total width of the PMC100_X and PMC100_Y registers is set by the `MDWIDTH` parameter, see *3.2 RTL parameters* on page 3-21. Unused register bits are reserved and must be treated as UNK/SBZP. The number of 32-bit locations occupied by each register is int(MDWIDTH/32)+1.

Configuration

These registers are always implemented.

Attributes

These are 32-bit registers.

The following figure shows the PMC100_X and PMC100_Y register bit assignments.

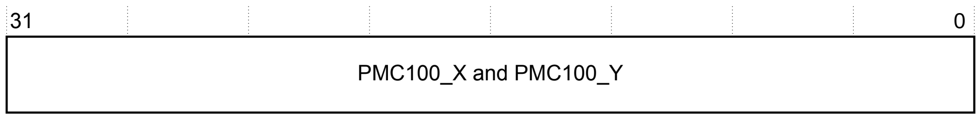


Figure 4-14 PMC100_X and PMC100_Y register bit assignments

The following table describes the PMC100_X and PMC100_Y register bit assignments.

Table 4-21 PMC100_X and PMC100_Y register bit assignments

Field	Name	Type	Reset value	Description
[31:0]	X0, Y0	RW	UNKNOWN	Bits [31:0]
[31:0]	X1, Y1	RW	UNKNOWN	Bits [63:32]
[31:0]	X2, Y2	RW	UNKNOWN	Bits [95:64]
[31:0]	X3, Y3	RW	UNKNOWN	Bits [127:96]
[31:0]	X4, Y4	RW	UNKNOWN	Bits [159:128]
[31:0]	X5, Y5	RW	UNKNOWN	Bits [191:160]
[31:0]	X6, Y6	RW	UNKNOWN	Bits [223:192]
[31:0]	X7, Y7	RW	UNKNOWN	Bits [255:224]

4.17 Auxiliary input register, PMC100_AIR

This register stores the value of the **AUXIN** signal when **MBISTOLACK** is asserted. The **AUXIN** signal is general purpose and could be used, for example, to record the value of the on-line MBIST error signal provided by some processors.

Usage constraints

The bits are sticky and so if a bit is set in a clock cycle it will remain set until cleared by software.

This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40.

The function of the PMC100_AIR register will depend on PMC-100 integration with the IP core. Therefore, its function will be described in the IP core documentation. If it is not mentioned, then it can be assumed that the **AUXIN** signal is not used in the integration. In this case, Arm recommends that software initialize the PMC100_AIR register to 0.

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_AIR bit assignments.

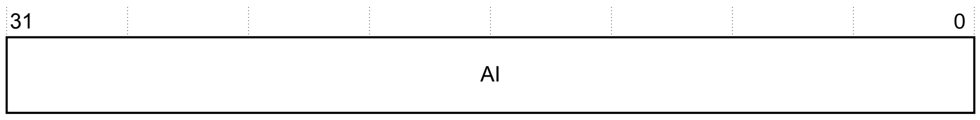


Figure 4-15 PMC100_AIR bit assignments

The following table describes the PMC100_AIR bit assignments.

Table 4-22 PMC100_AIR bit assignments

Field	Name	Type	Reset value	Description
[31:0]	AI[1,2]	RW	UNKNOWN	Sticky **AUXIN** signal value. Function is IP core specific, consult IP core documentation for details. **Note** • Unused field bits are reserved and must be treated as UNK/SBZP. • PMC100_AIR width is set by the `AIWIDTH` parameter, see *3.2 RTL parameters* on page 3-21

4.18 Auxiliary input register, PMC100_AOR

This register controls the value of the **AUXOUT** signal.

The **AUXOUT** signal is general purpose and could be used, for example, to control external logic associated with on-line MBIST testing, such as enabling **TC** signal pulse generation or the frequency of these pulses.

Usage constraints

This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40.

The function of the PMC100_AOR register will depend on PMC-100 integration with the IP core. Therefore, its function will be described in the IP core documentation. If it is not mentioned, then it can be assumed that the **AUXOUT** signal is not used in the integration. In this case, Arm recommends that SW initialize the PMC100_AOR register to 0.

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_AOR bit assignments.

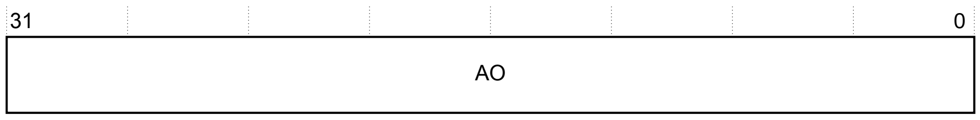


Figure 4-16 PMC100_AOR bit assignments

The following table describes the PMC100_AOR bit assignments.

Table 4-23 PMC100_AOR bit assignments

Field	Name	Type	Reset value	Description
[31:0]	AO[1,2]	RW	UNKNOWN	**AUXOUT** signal value. Function is IP core specific, consult IP core documentation for details. —— **Note** —— • Unused field bits are reserved and must be treated as UNK/SBZP. • AO width is set by the `AOWIDTH` parameter, see *3.2 RTL parameters* on page 3-21 ————

4.19 MBISTOLERR input register, PMC100_MER

This register stores the value of the **MBISTOLERR** signal when **MBISTOLACK** is asserted.

Usage constraints

The bits are sticky and so if a bit is set in a clock cycle it will remain set until cleared by software.

This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40.

Configuration

This register is always implemented.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_MER bit assignments.

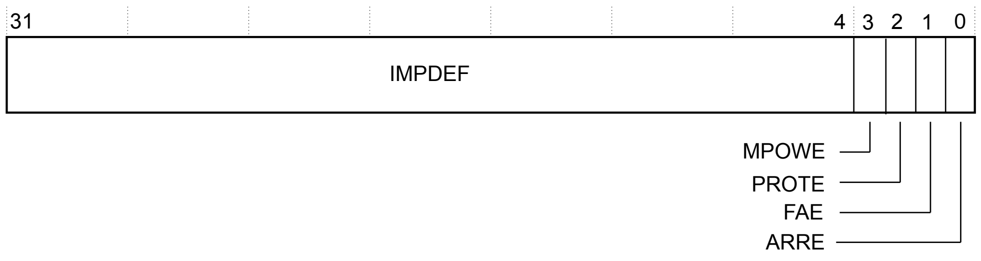


Figure 4-17 PMC100_MER bit assignments

The following table describes the PMC100_MER bit assignments.

Table 4-24 PMC100_MER bit assignments

Field	Name	Type	Reset value	Description
[31:4]	IMPDEF	RW	UNKNOWN	Implementation defined error register bits. Width of this field is set by the `MERWIDTH` parameter -4, see *3.2 RTL parameters* on page 3-21. See your IP core documentation for descriptions of these bits. **Note** Unused field bits are reserved and must be treated as UNK/SBZP.
[3]	MPOWE	RW	UNKNOWN	Indicates that the memory selected by **MBISTOLARRAY** is powered down and so cannot be tested.
[2]	PROTE	RW	UNKNOWN	Indicates that the protection logic selected by the combination of **MBISTOLPSEL** and **MBISTOLARRAY** is not implemented in the configuration of the IP core.
[1]	FAE	RW	UNKNOWN	Indicates that a functional access attempted to a memory currently selected for testing by **MBISTOLARRAY**.
[0]	ARRE	RW	UNKNOWN	Indicates that the memory selected by MBISTOLARRAY is not implemented in the configuration of the IP core.

4.20 Data mask, fault bitmap, and data registers, PMC100_DM0-PMC100_DM7

This register holds a value that is used to mask SRAM data values and store the fault bitmap when a test fails. It can also be used as a general-purpose data register in read and write operations in a similar way to the PMC100_X and PMC100_Y registers.

This is for use in address protection logic test algorithms, see *C.1 Address Protection Logic Latent Fault Detection algorithm* on page Appx-C-117

When PMC100_CTRL.DMDIS is `0b0`, the PMC100_DM register has the following behavior:
- It only masks the source data and memory read data values before they are compared against each other during compare operations. It does not mask read data that is stored in a register. When a PMC100_DM bit is 0 the corresponding data bit is masked to 0 and when a PMC100_DM bit is 1 the corresponding data bit is not masked.
- When a read check fails and the PMC100_CTRL.STOPF bit is 1, the PMC100_DM register is loaded with a 1 in each bit position that was not correct.

PMC100_DM register masking and fault bitmap behavior do not apply to MBIST reads performed by PCHKCEF, PCHKUEF and PCHKCE OPs.

When PMC100_CTRL.DMDIS is `0b1`, the PMC100_DM register the masking and fault bitmap features are disabled.

The PMC100_DM register width is configured to the data width of the MBIST interface. From a software perspective the PMC100_DM register is split into several 32-bit wide registers, PMC100_DM0-PMC100_DM7.

Usage constraints

These registers can be modified by software and are automatically updated. These registers must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40

The total width of this register is set by the `MDWIDTH` parameter, see *3.2 RTL parameters* on page 3-21.

Unused register bits are reserved and must be treated as UNK/SBZP.

The number of 32-bit locations occupied by this register is int(`MDWIDTH`/32)+1.

Configuration

These registers are always implemented.

Attributes

These are 32-bit registers.

The following figure shows the PMC100_DM0-PMC100_DM7 register bit assignments.

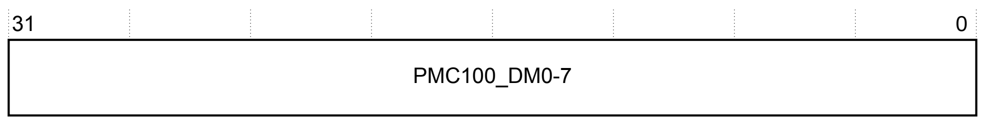


Figure 4-18 PMC100_DM0-PMC100_DM7 register bit assignments

The following table describes the PMC100_DM0-PMC100_DM7 register bit assignments.

Table 4-25 PMC100_DM0-PMC100_DM7 register bit assignments

Field	Name	Type	Reset value	Description
[31:0]	DM0	RW	UNKNOWN	Bits [31:0]
[31:0]	DM1	RW	UNKNOWN	Bits [63:32]
[31:0]	DM2	RW	UNKNOWN	Bits [95:64]
[31:0]	DM3	RW	UNKNOWN	Bits [127:96]
[31:0]	DM4	RW	UNKNOWN	Bits [159:128]
[31:0]	DM5	RW	UNKNOWN	Bits [191:160]
[31:0]	DM6	RW	UNKNOWN	Bits [223:192]
[31:0]	DM7	RW	UNKNOWN	Bits [255:224]

4.21 XOR mask registers, PMC100_XM0-PMC100_XM7

This register holds a value that is used to XOR mask data and address values.

This register is used with the XORD, XORA and SXM operations, see *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77. It is used in protection logic test algorithms.

The PMC100_XM register width is configured to the data width of the MBIST interface. From a software perspective the PMC100_XM register is split into eight 32-bit wide registers, PMC100_XM0-PMC100_XM7, as shown in the following table.

Usage constraints

These registers can be modified by software and are automatically updated by the SXM operation. These registers must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40

The total width of this register is set by the `MDWIDTH` parameter, see *3.2 RTL parameters* on page 3-21

Unused register bits are reserved and must be treated as UNK/SBZP.

The number of 32-bit locations occupied by this register is int(MDWIDTH/32)+1.

Configuration

These registers are always implemented.

Attributes

These are 32-bit registers.

The following figure shows the PMC100_XM0-PMC100_XM7 register bit assignments.

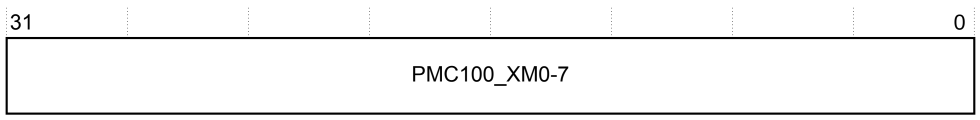


Figure 4-19 PMC100_XM0-PMC100_XM7 register bit assignments

The following table describes the PMC100_XM0-PMC100_XM7 register bit assignments.

Table 4-26 PMC100_XM0-PMC100_XM7 register bit assignments

Field	Name	Type	Reset value	Description
[31:0]	XM0	RW	UNKNOWN	Bits [31:0]
[31:0]	XM1	RW	UNKNOWN	Bits [63:32]
[31:0]	XM2	RW	UNKNOWN	Bits [95:64]
[31:0]	XM3	RW	UNKNOWN	Bits [127:96]
[31:0]	XM4	RW	UNKNOWN	Bits [159:128]
[31:0]	XM5	RW	UNKNOWN	Bits [191:160]
[31:0]	XM6	RW	UNKNOWN	Bits [223:192]
[31:0]	XM7	RW	UNKNOWN	Bits [255:224]

4.22 Program registers, PMC100_P0-PMC100_P31

The PMC-100 includes up to 32 program registers which hold microcode instructions.

These instructions constitute the algorithm that is executed to generate the required MBIST transactions that test the memory array or memory protection logic that the PMC100_AR register selects. For more information, see *4.8 Array register, PMC100_AR* on page 4-56

Usage constraints

These registers must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40

Unused register bits are reserved and must be treated as UNK/SBZP.

Configuration

The number of registers implemented is set by the `PROGSIZE` parameter. For more information, see *3.2 RTL parameters* on page 3-21

Attributes

These registers are 32-bits.

The following figure shows the PMC100_P0-PMC100_P31 register bit assignments.

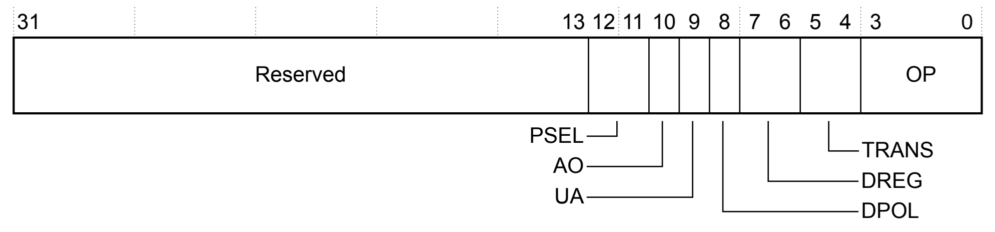


Figure 4-20 PMC100_P0-PMC100_P31 register bit assignments

The following table describes the PMC100_P0-PMC100_P31 register bit assignments.

Table 4-27 PMC100_P0-PMC100_P31 register bit assignments

Field	Name	Type	Reset value	Description
[31:13]	Reserved	UNK/ SBZP	-	Reserved, RES0
[12:11]	PSEL	RW	UNKNOWN	This is the output value of the **MBISTOLPSEL** signal. For more information, see *E.3 MBIST master interface signals* on page Appx-E-138. This signal allows access to the memory protection logic values in the IP core. For more information on PSEL encoding, see *4.22.1 PSEL encoding values* on page 4-80.

Table 4-27 PMC100_P0-PMC100_P31 register bit assignments (continued)

Field	Name	Type	Reset value	Description
[10]	AO	RW	UNKNOWN	Address output. This bit determines which address value is output on the MBISTOLADDR signal. The encoding is as follows: **0** Current address **1** Next address The address is determined by the PMC100_RADDR.RA and PMC100_CADDR.CA registers. For more information see, *4.14 Column address register, PMC100_CADDR* on page 4-65 and *4.15 Row address register, PMC100_RADDR* on page 4-67. **Note** When the PMC100_P operation is XORA, AO determines the value of A that is used in the XOR operation with the PMC100_XM register.
[9]	UA	RW	UNKNOWN	Update address. this bit causes the PMC100_CADDR.CA and PMC100_RADDR.RA address registers to be updated with the next value according to the values of PMC100_CTRL register ADDRID and ADDRCD bits. For more information, see *4.5 Main control register, PMC100_CTRL* on page 4-40. The encoding is as follows: `0b0` Hold address. Do not update the address registers. `0b1` Update address. Load the address registers with the next address value. **Note** • If PMC100_CTRL.BAMEN is 1, and LOOP-Last operation is executed the behavior is changed to decrement PMC100_CADDR.CA when UA is 1. In this case, PMC100_RADDR.RA does not change and PMC100_CADDR.CA is not decremented if it is 0. • If PMC100_CTRL.BAMEN is 1, then PMC100_CADDR.CA is only updated when UA is 1 and a LOOP-Last operation is executed. • If UA is 1 and a PUP operation is executed, then PMC100_RADDR.RA is set to ~PMC100_RADDR.RA This takes priority over the normal PMC100_RADDR.RA increment or decrement behavior when UA is1.

Table 4-27 PMC100_P0-PMC100_P31 register bit assignments (continued)

Field	Name	Type	Reset value	Description
[8]	DPOL	RW	UNKNOWN	Data polarity. This bit controls whether the data value is inverted or not. For read transactions it is used to determine if the non-inverted or inverted value of the specified DREG is checked against the data returned. For write transactions, it is used to determine if the non-inverted or inverted value of the specified DREG is written to the memory array. The encoding of this bit is as follows: `0b0` Use non-inverted DREG data value. `0b1` Use inverted DREG data value **Note** • When the TRANS field is `0b11`, DPOL controls the read data and the write data is controlled by the inverse DPOL value respectively. • For the other TRANS values the read and write data is controlled by the unmodified DPOL value. • DPOL is ignored for read instructions that save data in the PMC100_X, PMC100_Y or PMC100_DM register. • DPOL is also used by the PCHKCEF and PCHKUEF operations, see OP field description. • The DPOL value is ignored and the PMC100_CTRL.ADDRID value is used instead when PMC100_CTRL.BAMEN is 1 and PMC100_P.OP is either b0000 (CHKR/ NONE) or `0b1110` (XORD) and PMC100_P.DREG is `0b11` (FP). • The PCHKCEF and PCHKUCEF operations use DPOL to select different **MBISOLOUTDATA** bits to be checked.
[7:6]	DREG	RW	UNKNOWN	Data register used by the instruction. This selects either the PMC100_X data register, PMC100_Y data register, PMC100_DM, or the fixed patter value, FP. The encoding is as follows: `0b00` PMC100_X `0b01` PMC100_Y `0b10` PMC100_DM `0b11` FP **Note** • For more information on data register, see *4.16 Data registers, PMC100_X0-PMC100_X7 and PMC100_Y0-PMC100_Y7* on page 4-70. • For more information on the PMC100_DM register, see *4.20 Data mask, fault bitmap, and data registers, PMC100_DM0-PMC100_DM7* on page 4-74 • For more information on the PMC100_CTRL.FP field, see *4.5 Main control register, PMC100_CTRL* on page 4-40 • The PMC100_DM register is used as a data register in address protection logic test algorithms that do not require data masking. In this case the data masking function must be disabled by setting PMC100_CTRL.DMDIS to `0b1`. • It is not expected to be useful to set DREG to PMC100_DM when PMC100_CTRL.DMDIS to `0b0`.

Table 4-27 PMC100_P0-PMC100_P31 register bit assignments (continued)

Field	Name	Type	Reset value	Description
[5:4]	TRANS	RW	UNKNOWN	MBIST transaction generated. The encoding of this field is as follows: **00** None. No transaction generated **01** Read **10** Write **11** Read and write **Note** • Encoding `0b11` is only supported for two port SRAMs (1R1W). In this case the DPOL field controls the read data and the write data in controlled by the inverse DPOL value. • Encoding `0b11` must only be used with DREG set to `0b11` (FP). The behavior is UNPREDICTABLE for other DREG values. • Encoding `0b11` must only be used with PSEL set to `0b00`. • Encoding `0b11` must only be used when PMC100_CTRL.BAMEN is 0.
[3:0]	OP	RW	UNKNOWN	Operation field. This field specifies the operation that is performed by an instruction in addition to the MBIST transaction. For more information on encoding and function of each code, see *4.22.2 OP encoding values* on page 4-80.

4.22.1 PSEL encoding values

The PSEL encoding values can be:

Table 4-28 PSEL encoding values

`0b00`, and access type is a read or write	Direct SRAM access, protection logic bypassed
`0b01`, and access type is a write	ECC/parity generation logic path selected. Software must program the PMC100_BER register to all 1s when writing through the ECC/parity logic.
`0b01`, and access type is a read	ECC/parity error check result selected.
`0b10`, and access type is a read	ECC syndrome/parity value selected.
`0b11`, and access type is a read	ECC correction data value selected. Reserved for memories with protection logic that does not correct ECC errors (for example, instruction cache) or uses parity. **MBISTOLOUTDATAx** is RAZ and the error is indicated on **MBISTOLERR[2]** in this case.
`0b10` or `0b11`, and access type is a write	Reserved. Writes ignored, IP core indicates an error on **MBISTOLERR[2]**.

Note

When TRANS is `0b11`, PSEL must be `0b00`.

4.22.2 OP encoding values

The OP encoding values are:

Table 4-29 OP encoding values

`0b0000`	CHKR/NONE. Check read data/No operation. For MBIST read transactions the data returned is checked that it is equal to the contents of the data register selected by DREG. If they are not equal then the PMC100_CTRL.TF bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40. For MBIST write transactions no additional operation is performed.
`0b0001`	PUP. Pattern update. This operation is used in memory protection logic test algorithms.
`0b0010`	SAVERD. Save read data to data register specified by the DREG field. The read data returned is not checked. This operation must only be used with reads.
`0b0011`	WAITRU. Wait register updated. This operation is split into two parts that are performed in seperate clock cycles. The first part waits if there is a read that matches the DREG field or a read that will update the PMC100_CTRL.NOTRANS bit in the read pipeline. The second part issues the MBIST transaction specified in the TRANS field. For read transactions this operation also acts like SAVERD.
`0b0100`	LOOP-Last. This loop operation stops execution when the end condition is true and must be present in the last instruction in a microcode program, see *3.4 Loop operations* on page 3-24. Reads are also checked in the same way as the CHKR operation. Also causes the PMC100_CTRL.NOTRANS bit to be cleared to 0.
`0b0101`	LOOP-LAL. This loop operation loads the array address with PMC100_LOWADDR when the loop end condition is true, see *4.12 Low address register, PMC100_LOWADDR* on page 4-63. Reads are also checked in the same way as the CHKR operation. **Note** UA does not need to be 1 to enable the address load on end condition.
`0b0110`	LOOP-LAH. This loop operation loads the array address with the PMC100_HIGHADDR register value when the loop end condition is true, see *4.13 High address register, PMC100_HIGHADDR* on page 4-64. Reads are also checked in the same way as the CHKR operation. **Note** UA does not need to be 1 to enable the address load on end condition.
`0b0111`	LOOP-LCR. This loop operation either decrements PMC100_LCR.LC when PMC100_LCR.LC is not equal to 0 or loads PMC100_LCR.LC with the PMC100_LCR.LCI when PMC100_LCR.LC is equal to 0, see *4.24 Loop counter register, PMC100_LCR* on page 4-84. Reads are also checked in the same way as the CHKR operation.
`0b1000`	PCHKCEF. Protection error check correctable error bit fail. When DPOL is 1, this OP checks that the `0bMBISTOLOUTDATA[2]` value is 1 and if it is not then test fail is indicated, causing the PMC100_CTRL.TF bit to be set to 1. When DPOL is 0, this OP checks that the `0bMBISTOLOUTDATA[4]` value is 1 and if it is not then test fail is indicated, causing the PMC100_CTRL.TF bit to be set to 1. This OP must only be used with reads.
`0b1001`	PCHKUEF – Protection error check uncorrectable error bit fail. When DPOL is 1 this OP checks that the `0bMBISTOLOUTDATA[3]` value is 1 and if it is not then test fail is indicated, causing the PMC100_CTRL.TF bit to be set to 1. When DPOL is 0 this OP checks that the `0bMBISTOLOUTDATA[5]` value is 1 and if it is not then test fail is indicated, causing the PMC100_CTRL.TF bit to be set to 1. This OP must only be used with reads.
`0b1010`	PCHKCE. Protection error check correctable error. This OP checks **MBISTOLOUTDATA[1:0]** read value and if it is not equal to `0b01`, sets the PMC100_CTRL.NOTRANS bit to 1, causing subsequent MBIST transactions to be converted to NOPs. If PMC100_CTRL.TFPCHKUE is 1 and **MBISTOLOUTDATA[1]** is 1 then test fail is indicated, causing the PMC100_CTRL.TF bit to be set to 1. This OP must only be used with reads.
`0b1011`	CLRNT. Clear NOTRANS. This OP clears the PMC100_CTRL.NOTRANS bit to 0.

`0b1100`	SXM. Shift PMC100_XM register. If PMC100_LCR.LC != 0, then shift the PMC100_XM register: • When PM100_CTRL.ADDRID is 0, SXM shifts the PMC100_XM register left. • When PMC100_CTRL.ADDRID is 1, SXM shifts the PMC100_XM register right. **Note** • The TRANS field must be set to `0b00` with this OP. • When PMC100_LCR.LC==0, no operation is performed.
`0b1101`	Reserved
`0b1110`	XORD, XOR read or write data. DREG XOR PMC100_XM. For MBIST read transactions the MBISOLOUTDATA input data value is checked that it is equal to DREG XOR PMC100_XM. If it is not equal then the PMC100_CTRL.TF bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40. For MBIST write transactions the DREG XOR PMC100_XM operation generates the MBISOLINDATA output data value.
`0b1111`	XORA, XOR address. Address output is address XOR PMC100_XM. May be used with reads and writes. For reads also acts like the SAVERD OP.

Note

1. When PMC100_CTRL.PEEN is set to 1 by software, there must be at least one instruction with a LOOP Last operation field value.
2. Operation WAITRU notes:
   a. WAITRU waits for all previous reads in the pipeline that update DREG to complete. It is not useful to have more than one read in the pipeline that updates the same data register. Therefore, it is programming error if this is the case.
   b. If there are no reads in the pipeline that update DREG, then program execution does not wait but no operation is performed on the MBIST interface for one cycle, then the read or write is performed as indicated by the OP WAITRU instruction.
   c. WAITRU is not needed for writes that source data from a data register that was updated by an instruction that used OP SAVERD when the pipeline depth is less than the number of instructions between the read and the write multiplied by the cycles per operation of the MBIST transactions. Indeed, the OP WAITRU should be avoided in this case because it adds a redundant cycle to the program execution.
   d. WAITRU might be used with all four DREG encodings.
3. If an SXM operation is used after an XORD read operation, then a WAITRU operation with DREG set to the same value as the XORD operation must be used before SXM. This prevents the PMC100_XM register being updated before the read data is checked.
4. OP PUP updates the pattern used for testing and is used in memory protection logic test algorithms, see *Appendix C On-line MBIST Memory Protection Logic Test Algorithms* on page Appx-C-114. The behavior is as follows:
   • PMC100_LSPR.LS is set to 0 PMC100_CTRL.ADDRID is set to ~PMC100_CTRL.ADDRID
   • If (PMC100_P.UA==b1) PMC100_RADDR.RA is set to ~PMC100_RADDR.RA
5. For PCHKCEF and PCHKUEF operations, if DREG[0] is 1, address out is A XOR PMC100_XM

4.23 Loop start program register, PMC100_LSPR

This register contains the loop start field.

Usage constraints

This register cannot be modified by software, but is automatically updated.

Configuration

The number of registers implemented is set by the PROGSIZE parameter. For more information, see *3.2 RTL parameters* on page 3-21

Attributes

This is a 32-bit register.

The following figure shows the PMC100_LSPR register bit assignments.

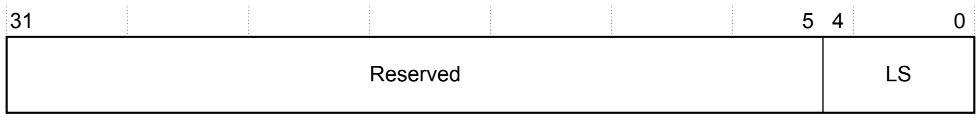


Figure 4-21 PMC100_LSPR register bit assignments

The following table describes the PMC100_LSPR register bit assignments.

Table 4-30 PMC100_LSPR register bit assignments

Field	Name	Type	Reset value	Description
[31:5]	Reserved	UNK/ SBZP	-	Reserved, RES0
[4:0]	LS	RO	UNKNOWN	Loop start. This field points to the program register at the start of the current loop. LS is automatically: • Loaded with 0 when a start_r or start_s event occurs. For more information, see *Start_r event* on page 4-47 and *Start_s event* on page 4-47 • Updated to point to the next microcode instruction when LOOP-LAL, LOOP-LAH, or LOOP-LCR loop ends. For more information, see *3.4 Loop operations* on page 3-24. **Note** • Unused field bits are reserved and must be treated as UNK/SBZP. • LS width is set by the $\log_2$(`PROGSIZE`).

4.24 Loop counter register, PMC100_LCR

This register contains the loop counter fields.

The PMC100_LCR allows simple C style for loops to be implemented and is mainly intended to be used in memory protection logic test algorithms.

Usage constraints

This register is modifiable by software and is automatically updated.

This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40.

The PMC100_LCR works in conjunction with the LOOP-Last and LOOP-LCR OPs, for further information on LOOP operations, see *3.4 Loop operations* on page 3-24.

Configuration

To use the PMC100_LCR SW must set PMC100_CTRL.EXECO to 0 and PMC100_LCR.LCI to the number loop iterations required minus 1 and optionally LCEN.LLEN to 1. PMC100_LCR.LC is loaded with the PMC100_LCR.LCI at the start of execution. :

- When a LOOP-Last and LCEN.LLEN is 1 or LOOP-LCR OP is executed and the loop counter is 0, execution either continues to the next instruction or execution stops, depending on the LOOP OP.
- When a LOOP-Last and LCEN.LLEN is 1 or LOOP-LCR OP is executed and the loop counter is decremented by 1 and execution continues at the start of the loop. If execution is suspended, then it continues at the start of the loop.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_LCR register bit assignments.

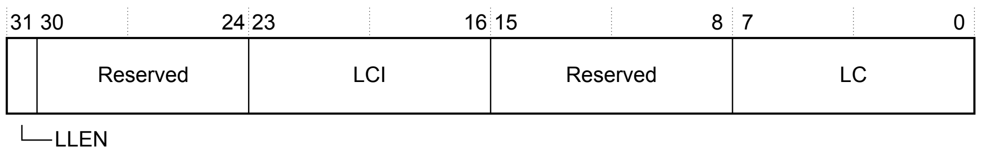


Figure 4-22 PMC100_LCR register bit assignments

The following table describes the PMC100_LCR register bit assignments.

Table 4-31 PMC100_LCR register bit assignments

Field	Name	Type	Reset value	Description
[31]	LLEN	RW	UNKNOWN	LOOP-Last enabled. This bit enables the loop counter behavior with the LOOP-Last OP. `0b0` Loop counter with LOOP-Last disabled `0b1` Loop counter with LOOP-Last enabled
[30:24]	Reserved	UNK/ SBZP	-	Reserved, RES0
[23:16]	LCI	RW	UNKNOWN	Loop counter initialization value.

Table 4-31 PMC100_LCR register bit assignments (continued)

Field	Name	Type	Reset value	Description
[15:8]	Reserved	UNK/ SBZP	-	Reserved, RES0
[7:0]	LC	RO	UNKNOWN	Loop counter value. The behavior is as follows: • If a start_r or start_s event occurs, then LC is loaded with LCI. For more information, see *Start_r event* on page 4-47 and *Start_s event* on page 4-47. • When an instruction with a LOOP-LCR OP is executed and LC is not equal to 0, then LC is decremented by 1 and execution continues at the start of the loop. • When an instruction with a LOOP-Last OP is executed and LLEN is b1 and LC is not equal to 0, then LC is decremented by 1 and execution continues at the start of the loop. • When an instruction with a LOOP-Last OP is executed and LLEN is b1 and LC is equal to 0, then execution stops. • When an instruction with a LOOP-LCR OP is executed and LC is equal to 0, then LC is loaded with LCI and execution continues at the next instruction.

4.25 Loop suspend counter register, PMC100_LSCR

This register contains the loop suspend counter fields.

The PMC100_LSCR allows simple C style for loops to be implemented and is mainly intended to be used in memory protection logic test algorithms.

Usage constraints

This register is modifiable by software and is automatically updated.

This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1, see *4.5 Main control register, PMC100_CTRL* on page 4-40.

If the PMC100_LSCR is not required by a test, then sofware must set PMC100_LSCR.LSCI to `0x0` and PMC100_LSCR.LSCEN to `0b0`.

The PMC100_LSCR works in conjunction with all LOOP operations. For more information on LOOP operations, see *3.4 Loop operations* on page 3-24.

Configuration

The loop suspend counter is typically used with on-line MBIST short burst SRAM and protection logic test algorithms. This is useful for IP cores that have a larger performance penalty due to on-line MBIST entry and so the loop suspend counter allows multiple passes through a loop to be executed back to back in the same MBIST session. Hence, execution will only be suspended when the loop suspend counter reaches 0.

To use the PMC100_LSCR SW must set PMC100_LSCR.LSCEN to 1, PMC100_CTRL.EXECO to 0, either PMC100_CTRL.TCSEN or PMC100_CTRL.TCCEN to 1 and PMC100_LSCR.LSCI to the number of times the loop is required to be executed before it is suspended minus 1. PMC100_LSCR.LSC is loaded with the PMC100_LSCR.LSCI value at the start of execution.

When a LOOP OP is executed, execution is suspended if the loop suspend counter is 0, else the loop suspend counter is decremented by 1 and execution will continue at the start of the loop. Execution will stop when the Stop event occurs, typically this is when a LOOP-Last OP is executed and either the memory address is equal to PMC100_HIGHADDR, PMC100_LOWADDR, or when PMC100_LCR.LC reaches 0 depending on PMC100_CTRLand PMC100_LCR register programing.

When PMC100_CTRL.BAMEN is 1, the normal operation of the loop suspend counter with LOOP-Last operations is disabled.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_LSCR register bit assignments.

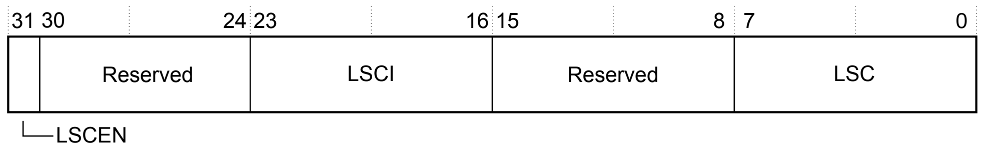


Figure 4-23 PMC100_LSCR register bit assignments

The following table describes the PMC100_LSCR register bit assignments.

Table 4-32 PMC100_LSCR register bit assignments

Field	Name	Type	Reset value	Description
[31]	LSCEN	RW	UNKNOWN	Loop suspend counter enable. `0b0` Loop suspend counter disabled `0b1` Loop suspend counter enabled
[30:24]	Reserved	UNK/ SBZP	-	Reserved, RES0
[23:16]	LSCI	RW	UNKNOWN	Loop suspend counter initialization value.
[15:8]	Reserved	UNK/ SBZP	-	Reserved, RES0
[7:0]	LSC	RO	UNKNOWN	Loop suspend counter value. The behavior is as follows: • When a start_r or a start_s or a resume event occurs then LSC is loaded with LSCI. • When an instruction with a LOOP OP is executed and LSCEN is `0b1` and LSC is not equal to 0, then LSC is decremented by 1 and execution continues at the start of the loop. • When an instruction with a LOOP OP is executed and LSCEN is `0b1` and LSC is equal to 0, then execution suspended. • When PMC100_CTRL.BAMEN is 1 and a LOOP-Last OP is executed, LSC is neither decremented nor is a suspend event generated due to LSC being equal to 0.

4.26 Test continue counter register, PMC100_TCCR

This register contains the test continue counter fields.

Usage constraints

This register is modifiable by software and is automatically updated.

This register must be initialized by software before the PMC100_CTRL.PEEN bit is set to 1.

The PMC100_TCCR is used to generate an internal test continue pulse to cause a resume event in the same way as the external **TC** input pin.

Configuration

If the PMC100_TCCR is not required by a test then SW must set PMC100_TCCR.TCCI to `0x0` and PMC100_CTRL.TCCEN to `0b0`.

Attributes

This is a 32-bit register.

The following figure shows the PMC100_TCCR register bit assignments.

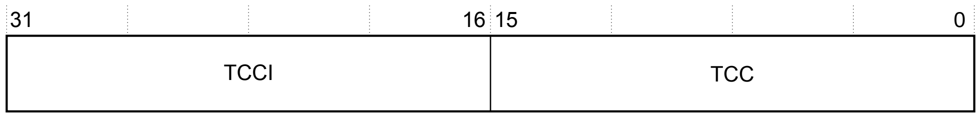


Figure 4-24 PMC100_TCCR register bit assignments

The following table describes the PMC100_TCCR register bit assignments.

Table 4-33 PMC100_TCCR register bit assignments

Field	Name	Type	Reset value	Description
[31:16]	TCCI	RW	UNKNOWN	Test continue counter initialization value.
[15:0]	TCC	RO	UNKNOWN	Test continue counter value. When CTRL.TCCENTCC is loaded with TCCI whena start_r or start_s event occurs. When PMC100_CTRL.TCCEN is `0b1` and PMC100_CTRL.PEEN is `0b1`, TCC is decremented every clock cycle and when it reaches 0 the internal test continue pulse is generated and TCC is reloaded with TCCI. If PMC100_CTRL.PES is `0b1` then a resume event will be generated.

4.27 CoreSight™ register summary

The following table shows the PMC-100 CoreSight registers.

Table 4-34 PMC-100 CoreSight registers

Offset	Name	Type	Reset value	Description
`0xF00`	PMC100_ITCTRL	RO	`0x00000000`	*4.28 Integration Mode Control register, PMC100_ITCTRL* on page 4-91
`0xF04-0xF9C`	-	UNK/ SBZP	UNKNOWN	Reserved
`0xFA0`	PMC100_CLAIMSET	RW	`0x0000000F`	*4.29 Claim Tag Set register, PMC100_CLAIMSET* on page 4-92
`0xFA4`	PMC100_CLAIMCLR	RW	`0x00000000`	*4.30 Claim Tag Clear register, PMC100_CLAIMCLR* on page 4-93
`0xFA8`	PMC100_DEVAFF0	RO	**CFGDEVAFF[31:0]**	*4.31 Device Affinity register 0, PMC100_DEVAFF0* on page 4-94
`0xFAC`	PMC100_DEVAFF1	RO	**CFGDEVAFF[63:32]**	*4.32 Device Affinity register 1, PMC100_DEVAFF1* on page 4-95
`0xFB0`	PMC100_LAR	WO	UNKNOWN	Software Lock Access Register. This register is not implemented.
`0xFB4`	PMC100_LSR	RO	`0x00000000`	Software Lock Status Register. This register is not implemented.
`0xFB8`	PMC100_AUTHSTATUS	RO	`0x00000000`	*4.33 Authentication Status register, PMC100_AUTHSTATUS* on page 4-96
`0xFBC`	PMC100_DEVARCH	RO	`0x47710A55`	*4.34 Device Architecture register, PMC100_DEVARCH* on page 4-97
`0xFC0`	PMC100_DEVID2	RO	`0x00000000`	Device Configuration Register 2. This register is RES0.
`0xFC4`	PMC100_DEVID1	RO	UNKNOWN **Note** The reset value depends on your configuration.	*4.35 Device Configuration Register 1, PMC100_DEVID1* on page 4-98
`0xFC8`	PMC100_DEVID	RO	UNKNOWN **Note** The reset value depends on your configuration.	*4.36 Device Configuration Register, PMC100_DEVID* on page 4-99
`0xFCC`	PMC100_DEVTYPE	RO	`0x00000055`	Device Type Identifier Register

Table 4-34 PMC-100 CoreSight registers (continued)

Offset	Name	Type	Reset value	Description
`0xFD0`	PMC100_PIDR4	RO	`0x00000004`	Peripheral identification registers
`0xFD4`	PMC100_PIDR5	RO	`0x00000000`	
`0xFD8`	PMC100_PIDR6	RO	`0x00000000`	
`0xFDC`	PMC100_PIDR7	RO	`0x00000000`	
`0xFE0`	PMC100_PIDR0	RO	`0x000000BA`	
`0xFE4`	PMC100_PIDR1	RO	`0x000000B9`	
`0xFE8`	PMC100_PIDR2	RO	`0x0000000B`	
`0xFEC`	PMC100_PIDR3	RO	`0x000000r0`	
`0xFF0`	PMC100_CIDR0	RO	`0x0000000D`	Component identification registers
`0xFF4`	PMC100_CIDR1	RO	`0x00000090`	
`0xFF8`	PMC100_CIDR2	RO	`0x00000005`	
`0xFFC`	PMC100_CIDR3	RO	`0x000000B1`	

4.28 Integration Mode Control register, PMC100_ITCTRL

The PMC100_ITCTRL register indicates that the PMC-100 only enter functional mode.

Usage constraints
This register is RO.

Configuration
This register is not required.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_ITCTRL register bit assignments.

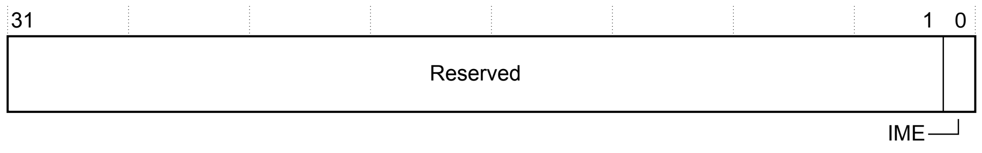


Figure 4-25 PMC100_ITCTRL register bit assignments

The following table describes the PMC100_ITCTRL register bit assignments.

Table 4-35 PMC100_ITCTRL register bit assignments

Field	Name	Type	Description
[31:1]	Reserved	-	Reserved, RES0.
[0]	IME	RO	Integration mode enable. The value of this field is: **0** PMC-100 must enter functional mode and is not in integration mode.

4.29 Claim Tag Set register, PMC100_CLAIMSET

The PMC100_CLAIMSET register provides various bits that can be separately set to indicate whether functionality is in use by the debug agent.

Usage constraints
This register is RW.

Configuration
This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_CLAIMSET register bit assignments.

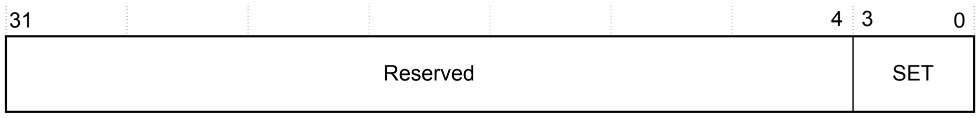


Figure 4-26 PMC100_CLAIMSET register bit assignments

The following table describes the PMC100_CLAIMSET register bit assignments.

Table 4-36 PMC100_CLAIMSET register bit assignments

Field	Name	Type	Description
[31:4]	Reserved	-	Reserved, RES0.
[3:0]	SET	RW	Sets Claim Tag bits. The read value is `0b1111`. The write behavior is: On reads, for each bit: **0** Claim tag bit is not implemented. **1** Claim tag bit is implemented. On writes, for each bit: **0** Has no effect. **1** Sets the relevant bit of the claim tag.

4.30 Claim Tag Clear register, PMC100_CLAIMCLR

The PMC100_CLAIMCLR register provides various bits that can be separately cleared to indicate whether functionality is in use by the debug agent.

Usage constraints
This register is RW.

Configuration
This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_CLAIMCLR register bit assignments.

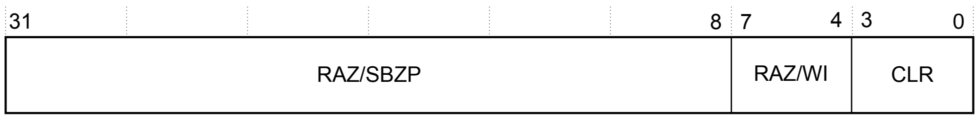


Figure 4-27 PMC100_CLAIMCLR register bit assignments

The following table describes the PMC100_CLAIMCLR register bit assignments.

Table 4-37 PMC100_CLAIMCLR register bit assignments

Field	Name	Type	Description
[31:8]	RAZ/SBZP	-	RAZ/SBZP
[7:4]	RAZ/WI	-	RAZ/WI
[3:0]	CLR	RW	Clear Claim Tag bits. On reads, for each bit: **0** Claim tag bit is not set. **1** Claim tag bit is set. On writes, for each bit: **0** Has no effect. **1** Clears the relevant bit of the claim tag.

4.31 Device Affinity register 0, PMC100_DEVAFF0

The PMC100_DEVAFF0 register enables a debugger to determine whether two components have an affinity with each other

Usage constraints
This register is RO.

Configuration

This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_DEVAFF0 register bit assignments.

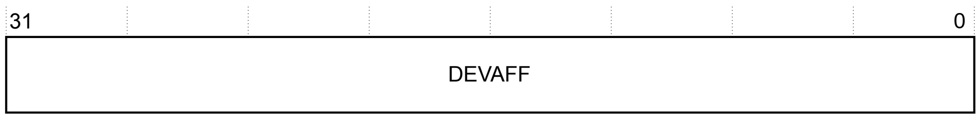


Figure 4-28 PMC100_DEVAFF0 register bit assignments

The following table describes the PMC100_DEVAFF0 register bit assignments.

Table 4-38 PMC100_DEVAFF0 register bit assignments

Field	Name	Type	Description
[31:0]	DEVAFF	RO	Indicates the value read This field holds the value read from the **CFGDEVAFF[31:0]** configutation signal.

4.32 Device Affinity register 1, PMC100_DEVAFF1

The PMC100_DEVAFF1 register enables a debugger to determine whether two components have an affinity with each other

Usage constraints
This register is RO.

Configuration
This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_DEVAFF1 register bit assignments.

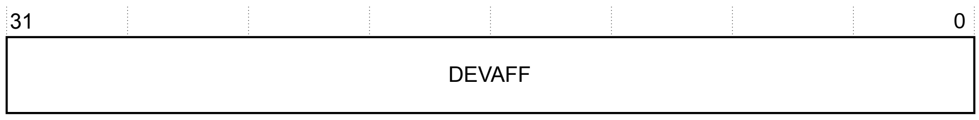


Figure 4-29 PMC100_DEVAFF1 register bit assignments

The following table describes the PMC100_DEVAFF1 register bit assignments.

Table 4-39 PMC100_DEVAFF1 register bit assignments

Field	Name	Type	Description
[31:0]	DEVAFF	RO	Indicates the value read This field holds the value read from the **CFGDEVAFF[63:32]** configutation signal.

4.33 Authentication Status register, PMC100_AUTHSTATUS

The PMC100_AUTHSTATUS register determines what debug levels are supported. This register is `0x00000000` indicating that authentication is not implemented.

Usage constraints
This register is RO.

Configuration

This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_AUTHSTATUS register bit assignments.

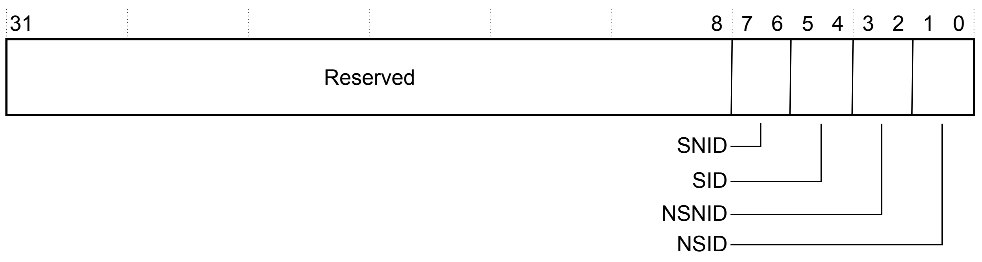


Figure 4-30 PMC100_AUTHSTATUS register bit assignments

The following table describes the PMC100_AUTHSTATUS register bit assignments.

Table 4-40 PMC100_AUTHSTATUS register bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	Reserved, RES0.
[7:6]	SNID	RO	Secure non-invasive debug, This field is `0b00`, indicating that this debug level is not supported.
[5:4]	SID	RO	Secure invasive debug. This field is `0b00`, indicating that this debug level is not supported.
[3:2]	NSNID	RO	Non-secure non-invasive debug. This field is `0b00`, indicating that this debug level is not supported.
[1:0]	NSID	RO	Non-secure invasive debug. This field is `0b00`, indicating that this debug level is not supported.

4.34 Device Architecture register, PMC100_DEVARCH

The PMC100_DEVARCH register identifies and architecture of a CoreSight component.

Usage constraints
This register is RO.

Configuration

This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_DEVARCH register bit assignments.

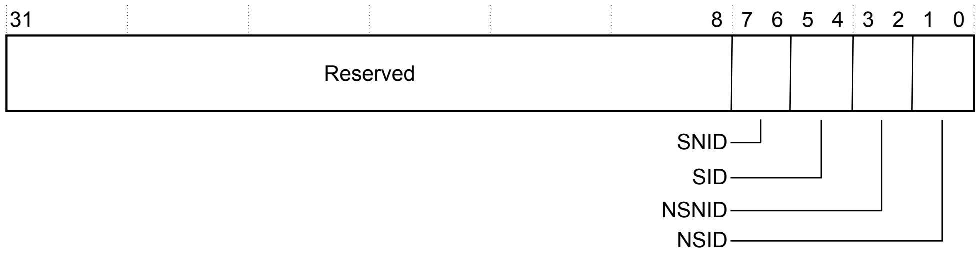


Figure 4-31 PMC100_DEVARCH register bit assignments

The following table describes the PMC100_DEVARCH register bit assignments.

Table 4-41 PMC100_DEVARCH register bit assignments

Field	Name	Type	Description
[31:21]	ARCHITECT	RO	This field defines the architect of the component, which in this case, is Arm. The value of this field is `0x23B`
[20]	PRESENT	RO	This field indicates the presence of this register. This field is `0b1`, indicating that this is register is present.
[19:16]	REVISION	RO	This field indicates the architecture revision. The value of this field is `0b0001`
[15:0]	ARCHID	RO	This field indicates the architecture ID, and holds the value `0x0A55`, indicating the PMC component.

4.35 Device Configuration Register 1, PMC100_DEVID1

The PMC100_DEVID1 register indicates the capabilities of the component.

Usage constraints
This register is RO.

Configuration

This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_DEVID1 register bit assignments.

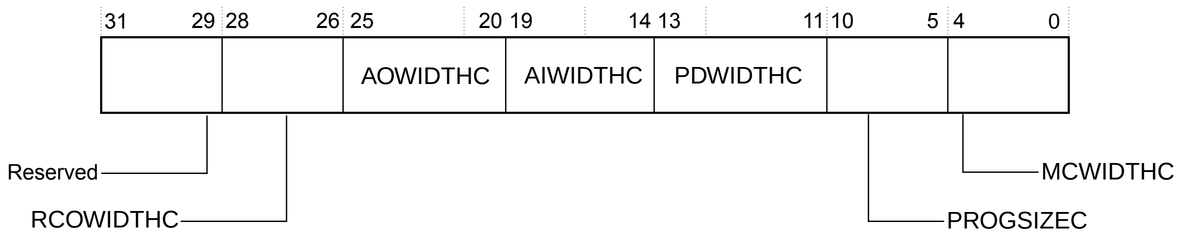


Figure 4-32 PMC100_DEVID1 register bit assignments

The following table describes the PMC100_DEVID1 register bit assignments.

Table 4-42 PMC100_DEVID1 register bit assignments

Field	Name	Type	Description
[31:29]	Reserved	-	Reserved, RES0.
[28:26]	RCOWIDTHC	RO	PMC100_MCR.RCOW and PMC100_MCR.RCOR field widths.
[25:20]	AOWIDTHC	RO	PMC100_AOR register and **AUXOUT** signal width configuration.
[19:14]	AIWIDTHC	RO	PMC100_AIR register and **AUXIN** signal width configuration.
[13:11]	PDWIDTHC	RO	Pipeline depth field width configuration.
[10:5]	PROGSIZEC	RO	Microcode program size configuration.
[4:0]	MCWIDTHC	RO	MBIST interface configuration signal width configuration

4.36 Device Configuration Register, PMC100_DEVID

The PMC100_DEVID1 register indicates the capabilities of the component.

Usage constraints
This register is RO.

Configuration

This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_DEVID register bit assignments.

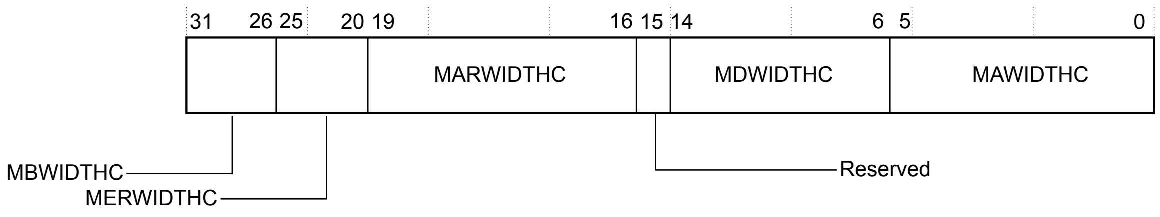


Figure 4-33 PMC100_DEVID register bit assignments

The following table describes the PMC100_DEVID register bit assignments.

Table 4-43 PMC100_DEVID register bit assignments

Field	Name	Type	Description
[31:26]	MBWIDTHC	RO	MBIST interface byte enable width configuration.
[25:20]	MERWIDTHC	RO	PMC100_MER register and MBIST interface error signal width configuration.
[19:16]	MARWIDTHC	RO	MBIST interface array width configuration.
[15]	Reserved	RO	Reserved, RES0.
[14:6]	MDWIDTHC	RO	MBIST interface data width configuration.
[5:0]	MAWIDTHC	RO	MBIST interface address width configuration.

4.37 Device Type Register, PMC100_DEVTYPE

The PMC100_DEVITYPE provides information about the PMC-100 component.

Usage constraints
This register is RO.

Configuration
This register is always implemented.

Attributes
This is a 32-bit register.

The following figure shows the PMC100_DEVTYPE register bit assignments.

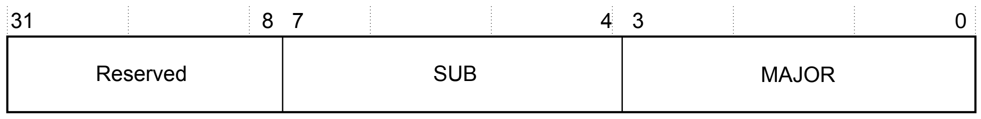


Figure 4-34 PMC100_DEVTYPE register bit assignments

The following table describes the PMC100_DEVTYPE register bit assignments.

Table 4-44 PMC100_DEVTYPE register bit assignments

Field	Name	Type	Description
[31:8]	-	RO	Reserved, RES0.
[7:4]	SUB	RO	The subtype of the component. The value of this field is `0x5`, indicating memory, tightly coupled device such as *Built-In Self Test* (BIST).
[3:0]	MAJOR	RO	The main type of the component. The value of this field is `0x5`, indicating debug logic.

4.38 PMC100_PIDR0-7, Peripheral Identification Registers

The PMC100_PIDR0-7 provides the standard Peripheral IDs that are required to identify the PMC-100 component.

Usage constraints
Only bits[7:0] of each register are used. This means that PMC100_PIDR0-7 define a single 64-bit *Peripheral ID*, as the following figure shows.

Configurations
Available in all configurations.

Attributes
These registers are individually 32-bits wide.

The following figure shows the mapping between PMC100_PIDR0-7 and the single 64-bit *Peripheral ID* value.

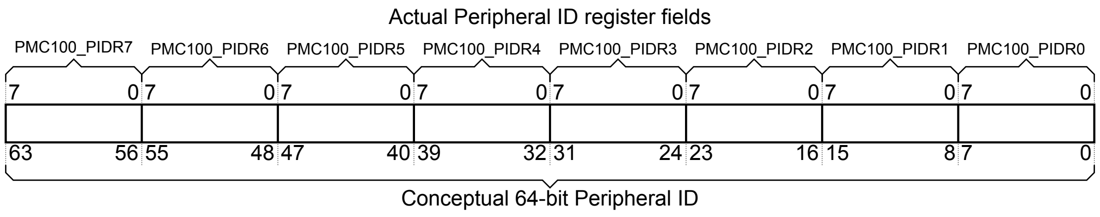


Figure 4-35 Mapping between PMC100_PIDR0-7 and the Peripheral ID value

The following figure shows the Peripheral ID bit assignments in the single conceptual Peripheral ID register.

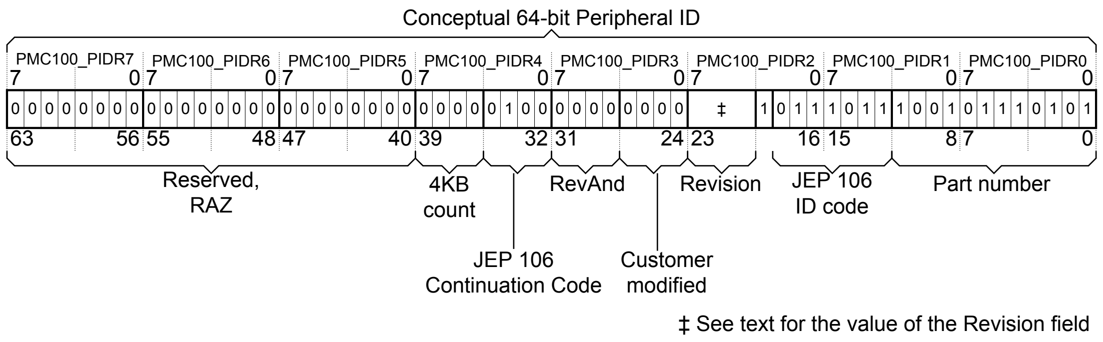


Figure 4-36 Peripheral ID fields

The following table shows the values of the fields when reading this set of registers.

The registers are listed in order of register name, from most significant (PMC100_PIDR7) to least significant (PMC100_PIDR0). This does not match the order of the register offsets.

Table 4-45 PMC100_PIDR0-7 bit assignments

Register	Bits	Name	Description
PMC100_PIDR7	[31:0]	-	Reserved, RES0.
PMC100_PIDR6	[31:0]	-	Reserved, RES0.
PMC100_PIDR5	[31:0]	-	Reserved, RES0.

Table 4-45 PMC100_PIDR0-7 bit assignments (continued)

Register	Bits	Name	Description
PMC100_PIDR4	[31:8]	-	Reserved, RES0.
	[7:4]	SIZE	This field indicates the memory size that is used by the component. The value of this field is `0x0`, indicating a 4KB block.
	[3:0]	DES_2	JEP 106 continuation code. The value of this field is 4, indicating Arm JEP106 continuation code.
PMC100_PIDR3	[31:8]	-	Reserved, RES0.
	[7:4]	REVAND	Part minor revision. This is the **ECOREVNUM** input signal value sampled at reset.
	[3:0]	JEDEC	Customer Modified. `0x0` indicates from Arm.
PMC100_PIDR2	[31:8]	-	Reserved, RES0.
	[7:4]	REVISION	Revision Number of `0x0` r0p0 Peripheral.
	[3]	JEDEC	This field is `0x1`, indicating that the JEDEC assigned value is used.
	[2:0]	DES_1	JEP 106 identity code [6:4]. This field is `0x3`, indicating Arm ID.
PMC100_PIDR1	[31:8]	-	RES0.
	[7:4]	DES_0	JEP 106 identity code [3:0]. This field is `0xB`, indicating Arm ID.
	[3:0]	PART_1	Part Number[11:8]. This field is `0x9`, indicating PMC-100.
PMC100_PIDR0	[31:8]	-	RES0.
	[7:0]	PART_0	Part Number [7:0]. This field is `0xBA`, indicating PMC-100.

4.39 PMC100_CIDR0-3, Component Identification Registers

The PMC100_CIDR0-3 provides the standard Component IDs that are required to identify the PMC-100 component.

Usage constraints
Only bits[7:0] of each register are used. This means that PMC100_CIDR0-3 define a single 32-bit Component ID, as the following figure shows.

Configurations
Available in all configurations.

Attributes
These registers are each 32-bits wide.

The following figure shows the mapping betweenPMC100_CIDR0-3 and the single 64-bit *Component ID* value.

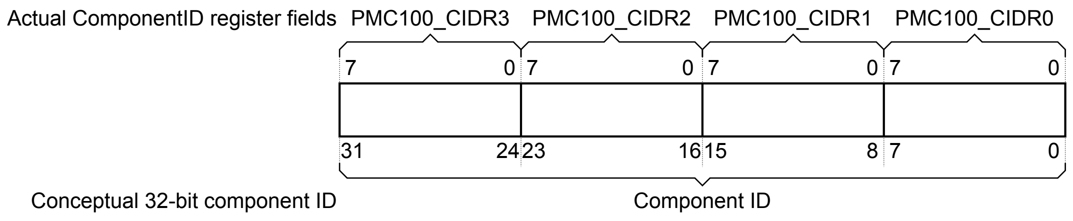


Figure 4-37 Mapping between PMC100_CIDR0-3 and the Component ID value

The following table shows the Component ID bit assignments in the single conceptual Component ID register.

The registers are listed in order of register name, from most significant (PMC100_CIDR3) to least significant (PMC100_CIDR0). This does not match the order of the register offsets.

Table 4-46 PMC100_CIDR0-3 bit assignments

Register	Bits	Name	Description
PMC100_CIDR3	[31:8]	-	Reserved, RES0.
	[7:0]	PRMBL_3	Preamble, segment 3. The value of this field is `0xB1`.
PMC100_CIDR2	[31:8]	-	Reserved, RES0.
	[7:0]	PRMBL_2	Preamble, segment 2. The value of this field is `0x05`.
PMC100_CIDR1	[31:8]	-	Reserved, RES0.
	[7:4]	CLASS	The component class value, which is, `0x9` to indicate that it is a CoreSight component.
	[3:0]	PRMBL_1	Preamble, segment 1. The value of this field is `0x0`.
PMC100_CIDR0	[31:8]	-	Reserved, RES0.
	[7:0]	PRMBL_0	Preamble, segment 0. The value of this field is `0x0D`.

Appendix A
Short-burst software-transparent algorithm

This chapter describes the short-burst software-transparent algorithm.

It contains the following sections:

A.1 Short-burst software-transparent overview

The short burst SRAM test algorithms are non-destructive and can be run so that they are transparent to software running on the processor.

The algorithms consist of a loop that tests two SRAM entries at time. The algorithms then increment the address by two at the end of the loop, ready to test the next two entries. The algorithms stop if a fault is detected or the maximum address of the SRAM is reached.

PMC-100 can be programmed to suspend execution at the end of a loop or after executing a specific number of loops. PMC-100 can be programmed to trigger execution again when it's internal TCCR counter expires or using the external TC input pin. It can also be programmed not to suspend and in this case will continue execution until the maximum address is reached.

A.2 SRAM faults

The short-burst software transparent *Memory Built-In Self Test* (MBIST) algorithm covers delay and stuck-at faults because of transistor ageing and electromigration in the following SRAM circuits:

- Individual bit cells
- Word lines
- Timing circuits
- Sense amps
- Data multiplexors

Note

These algorithms do not cover all address decoder faults . The production MBIST March C - SRAM algorithm has better coverage of address decoder faults , see *Appendix B Production test March Algorithm* on page Appx-B-111.

A.3 Single ported SRAM test algorithm

This algorithm tests two entries in each burst, which are in adjacent rows in the SRAM array. Therefore, it is carried out N/2 times for each SRAM, where N is the number of entries in the SRAM under test. The algorithm is intended to test two locations in adjacent rows in the SRAM array. The algorithm toggles the wordlines, bitlines, address signals, data signals, and control signals at the maximum functional frequency. The algorithm can be used with incrementing or decrementing addresses.

When using incrementing address, each burst uses two entries `n` and `n+m`, where `n` is a variable containing the address of the first entry and `n+m` is the address of the second entry. Where `m` is the mux factor for the SRAM under test and is a constant power of 2 that is programmed into the PMC100_MCR.CCW field. After each burst, `n` is set to `n+2m` and at the start of a test it is initialized to 0 or `m`. When m is used to initialize `n`, the two entries are in different SRAM columns when the top of an SRAM row is reached. This improves test coverage because the test switches between different rows and columns at full functional frequency.

When using a decrementing address, each burst uses two entries `n` and `n-m`. After each burst `n` is set to `n-2m` and at the start of a test it is initialized to `N-1` or `N-1-m`.

The algorithm uses a *Fixed Pattern* (FP), and its inverse, ~FP. For example, a checkerboard pattern such as, `0b10101010` and `0b01010101` respectively, see the PMC100_CTRL.FP field. The algorithm shown in the table below uses an incrementing address but it might be rewritten using a decrementing address.

Table A-1 Example short burst software transparent on-line MBIST algorithm

Operation	Notes
Read entry `n` and store in PMC100_X register.	Save entry `n`.
Read entry `n+m` and store in PMC100_Y register.	Save entry `n+m` and activates next wordline.
Write FP to entry `n+m`.	-
Write ~FP to entry `n`.	Switches wordlines and all write data bitlines.
Read entry `n` and check that it is equal to ~FP.	Verifies that no bit is stuck in entry `n`.
Read entry `n+m` and check that it is equal to FP.	Verifies that no bit is stuck in entry `n+m`, switches wordlines and all read data bitlines.
Write ~FP to entry `n+m`.	-
Write FP to entry `n`.	Switches wordlines and all write data bitlines.
Read entry `n` and check that it is equal to FP.	Verifies that no bit is stuck in entry `n`.
Read entry `n+m` and check that it is equal to ~FP.	Verifies that no bit is stuck in entry `n+m`, and switches wordlines and all read data bitlines.
Write PMC100_X to entry `n`.	This restores entry `n`.
Write PMC100_Y to entry `n+m`.	Restores entry `n+m`.
Read entry `n` and check that it is equal to PMC100_X.	This checks that PMC100_X was restored correctly.
Read entry `n+m` and checks that it is equal to PMC100_Y.	Checks that PMC100_Y was restored correctly.

A.3.1 Microcode

The microcode assumes that PMC-100 is configured in the x-fast address update mode, which updates the row address (word line) before the column address.

The following table shows the PMC-100 microcode that implements the test algorithm that is described in *A.3 Single ported SRAM test algorithm* on page Appx-A-107. In the following table, the instruction field are:

- PSEL
- AO
- UA
- DPOL
- DREG
- TRANS
- OP

Table A-2 Microcode for short-burst sofware transparent on-line MBIST algorithm

Register	Instruction							Address out	Address update	Data polarity	Data	Trans	Operation
PMC100_P0	00	0	0	0	00	01	0010	Address	Hold	-	PMC100_X	Read	Save read data
PMC100_P1	00	1	0	0	01	01	0010	Next address	Hold	-	PMC100_Y	Read	Save read data
PMC100_P2	00	1	0	0	11	10	0000	Next address	Hold	No inverse	FP	Write	Write
PMC100_P3	00	0	0	1	11	10	0000	Address	Hold	Inverse	FP	Write	Write
PMC100_P4	00	0	0	1	11	01	0000	Address	Hold	Inverse	FP	Read	Check read data
PMC100_P5	00	0	0	0	11	01	0000	Next address	Hold	No inverse	FP	Read	Check read data
PMC100_P6	00	0	0	1	11	10	0000	Next address	Hold	Inverse	FP	Write	Write
PMC100_P7	00	0	0	0	11	10	0000	Address	Hold	No inverse	FP	Write	Write
PMC100_P8	00	0	0	0	11	01	0000	Address	Hold	No inverse	FP	Read	Check read data
PMC100_P9	00	1	0	1	11	01	0000	Next address	Hold	Inverse	FP	Read	Check read data
PMC100_P10	00	0	0	0	00	10	0011	Address	Hold	No inverse	PMC100_X	Write	Wait for register update and write
PMC100_P11	00	1	0	0	01	10	0000	Next address	Hold	No inverse	PMC100_Y	Write	Write
PMC100_P12	00	0	1	0	00	10	0000	Address	Update	No inverse	PMC100_X	Read	Check read data
PMC100_P13	00	0	1	0	01	01	0100	Address	Update	No inverse	PMC100_Y	Read	Check read data, LOOP-Last

A.4 Two ported SRAM test algorithm

In the two port SRAM test algorithm, one port is write-only and the other is read-only. PMC-100 has an additional write address signal, **MBISTOLWADDR**, to support two ported SRAMs.

The algorithm has been adapted to comply with the PMC-100 two port SRAM test rules, as follows:

- Read and write addresses must be different, one must be `n` and the other must be `n+m`.
- Read and write data values must be different, one must be the inverse of the other.

Table A-3 Short burst software transparent on-line MBIST test algorithm for two port SRAMs

Write port	Read port
No operation (NOP)	Read location `n` and store in PMC100_X register
No operation (NOP)	Read location `n+m` and store in PMC100_Y register
Write FP to location `n`	No operation (NOP)
Write ~FP to location `n+m`	Read location `n` and check that it is equal to FP
Write FP to location `n*`	Read location `n+m` and check that it is equal to ~FP
Write ~FP to location `n`	No operation (NOP)
Write FP to location `n+m`	Read location `n` and check that it is equal to ~FP
Write ~FP to location `n*`	Read location `n+m` and check that it is equal to FP
Write PMC100_X to location `n`	No operation (NOP)
Write PMC100_Y to location `n+m`	No operation (NOP)
No operation (NOP)	Read location `n` and check that it is equal to PMC100_X
No operation (NOP)	Read location `n+m` and check that it is equal to PMC100_Y

Note

* indicates redundant operations used to keep the interface active instead of using a NOP instruction.

A.4.1 Microcode

The microcode shown in the following table assumes that PMC-100 is configured in the x-fast address update mode, which updates the row address (word line) before the column address.

The following table shows the PMC-100 microcode that implements the test algorithm that is described in *A.4.1 Microcode* on page Appx-A-109.

In the following table, the instruction field are:

- PSEL
- AO
- UA
- DPOL
- DREG
- TRANS
- OP

Table A-4 Microcode for short-burst sofware transparent on-line MBIST algorithm

Register	Instruction							Address out		Address update	Data polarity		Data	Trans		Operation
								Write	Read		Write	Read		Write	Read	
PMC100_P0	00	0	0	0	00	01	0010	-	Address	Hold	-	-	PMC100_X	-	Read	Save read data
PMC100_P1	00	1	0	0	01	01	0010	-	Next address	Hold	-	-	PMC100_Y	-	Read	Save read data
PMC100_P2	00	1	0	0	11	10	0000	Address	-	Hold	No inverse	-	FP	Write	-	Write
PMC100_P3	00	0	0	1	11	10	0000	Next address	Address	Hold	Inverse	No invese	FP	Write	-	Write and check read data
PMC100_P4	00	1	0	1	11	11	0000	Address	Next address	Hold	No inverse	Inverse	FP	Write	Read	Write and check read data
PMC100_P5	00	1	0	1	11	10	0000	Address	-	Hold	Inverse	-	FP	Write	-	Write
PMC100_P6	00	0	0	1	11	11	0000	Next address	Address	Hold	No inverse	Inverse	FP	Write	Read	Write and check read data
PMC100_P7	00	1	0	0	11	11	0000	Address	Next address	Hold	Inverse	No inverse	FP	Write	Read	Write and check read data
PMC100_P8	00	1	0	0	00	10	0011	Address	-	Hold	No inverse	-	PMC100_X	Write	-	Wait for register update and write
PMC100_P9	00	0	0	0	01	10	0000	Next address	-	Hold	No inverse	-	PMC100_Y	Write	-	Write
PMC100_P10	00	0	0	0	00	01	0000	-	Address	Hold	Inverse	No inverse	PMC100_X	-	Read	Check read data
PMC100_P11	00	1	1	0	01	01	0100	-	Next address	Update	-	No inverse	PMC100_Y	-	Read	Check read data, LOOP-Last

Appendix B
Production test March Algorithm

Ths chapter describes the March *Memory Buile-In Self-Test* (MBIST) algorithm. It also provides information on how to program PMC-100 to perform an example March MBIST algorithm called March C-, and this information can be used as the basis to implement other production test MBIST algorithms.

It contains the following sections:

B.1 Production test March algorithm overview

March *Memory Built-In Self Test* (MBIST) algorithms are normally used in production test, but they can also be used for in-field SRAM testing at Cold reset or periodically during operation and PMC-100 can execute these test algorithms.

March algorithms destroy the memory contents. Therefore, they can only be run using the on-line MBIST, off-line memory, and production MBIST use models.

A March MBIST algorithm tests an SRAM by filling all its entries test patterns and it carries out several passes through an SRAM checking the patterns and writing new patterns. The SRAM read and write operations performed on each pass is called a March element and each element is repeated for each entry in an SRAM. The direction the address is incremented or decrement for each entry is indicated in the March notation.

B.1.1 March notation

March algorithms are described using the notation shown in the following table.

Table B-1 March notation

Notation	Description
()	March element. Contains read and write write operations to be carried out on each SRAM entry
r0	Read SRAM and check that the value is equal to pattern 0
r1	Read SRAM and check that the value is equal to pattern 1
w0	Write pattern 0 to SRAM
w1	Write pattern 1 to SRAM
$\Uparrow$	Increment address after the operations in a March element are carried out
$\Downarrow$	Decrement address after the operations in a March element are carried out

Pattern 0 and 1 can be any value, but they must be the inverse of each other.

The operations in a March element are performed on the current SRAM address, then the address is incremented or decremented as specified and the operations are repeated for the new address. This is repeated for all entries in an SRAM. Then the next March element is executed for all SRAM entries. When all elements have been executed the test ends.

B.2 March C- algorithm

The March C- *Memory Built-In Self Test* (MBIST) algorithm covers the majority of SRAM faults, including address decoder faults and it only requires 10 accesses per SRAM entry. The algorithm is as follows:

$\{\Uparrow (w0); \Uparrow (r0, w1); \Uparrow (r1, w0); \Downarrow (r0, w1); \Downarrow (r0, w1); \Downarrow (r0)\}$

B.2.1 Microcode

The following table shows the micrcode programming for the March C-algorithm.

Note

- The example microcode uses a fixed data value FP, but it is also possible to use an arbitarty data value programmed into the PMC100X or PMC100_Y registers.
- In the Operation column in the following table, LAL is the load address with LOWADDR and LAH is the load address with PMC100_HIGHADDR.
- In the Instruction column in the following table, the instruction fields are:
  - PSEL
  - AO
  - UA
  - DPOL
  - DREG
  - TRANS
  - OP

Table B-2 Microcode for March C- algorithm

Register	Instruction							Address out	Address update	Data polarity	Data	Trans	Operation
PMC100_P0	00	0	1	0	11	10	0101	Address	Update	No inverse	FP	Write	Write, LOOP-LAL
PMC100_P1	00	0	0	0	11	01	0000	Address	Hold	No inverse	FP	Read	Check read data
PMC100_P2	00	0	1	1	11	10	0101	Address	Update	Inverse	FP	Write	Write, LOOP-LAL
PMC100_P3	00	0	0	1	11	01	0000	Address	Hold	Inverse	FP	Read	Check read data
PMC100_P4	00	0	1	0	11	10	0110	Address	Update	No inverse	FP	Write	Write, LOOP-LAL
PMC100_P5	00	0	0	0	11	01	0000	Address	Hold	No inverse	FP	Read	Check read data
PMC100_P6	00	0	1	1	11	10	0110	Address	Update	Inverse	FP	Write	Write, LOOP-LAL
PMC100_P7	00	0	0	1	11	01	0000	Address	Hold	Inverse	FP	Read	Check read data
PMC100_P8	00	0	1	0	11	10	0110	Address	Update	No inverse	FP	Write	Write, LOOP-LAL
PMC100_P9	00	0	1	0	11	01	0100	Address	Update	No inverse	FP	Read	Check read data, LOOP Last

Appendix C
On-line MBIST Memory Protection Logic Test Algorithms

The on-line *Memory Built-In Self Test* (MBIST) test algorithms described in this section show how *Error Correcting Code* (ECC) generation, checking, correction logic, parity generation, and checking logic can be tested.

The following algorithms in combination thoroughly test memory ECC and parity logic. These algorithms test for single-point and latent faults in the data and address parts of the ECC logic and data parts of parity logic.

———— **Note** ————

The algorithms described in this section require that PMC100_BER is programmed to all 1s. The algorithms test the logic to detect stuck-at faults. They can also detect delay faults in the logic because back-to-back reads are performed using two RAM entries.

————————————

The algorithms generate: test patterns, TP and use a fixed base pattern, BP, its inverse, ~BP, a mask value XM and an XOR function.

- Test patterns, TPs, which drive the address or data signals
- TPs use a fixed base pattern, BP
- Inverse of BP, ~BP
- Mask value, XM
- XOR function

Algorithm Overview

Each algorithm is carried out twice, once using BP and once using ~BP, and the order is not important. TP is generated as either of the following:

- TP=BP XOR XM
- TP=~BP XOR XM

BP can have any value and the algorithms might be run multiple times with different BP values to increase fault coverage. The algorithms are normally carried out with one XM bit set to 1 and the others set to 0 but they can also be carried out with multiple bits set to one to test ECC schemes that can detect two or more faults, or when an MBIST controller accesses data containing multiple protection code fields.

The algorithms contain a loop that tests each address or data bit in turn by using a different XM value for each loop iteration. The order that the address or data bits are tested in is not important and neither is how XM is generated during each iteration of the loop. XM generation methods can include, shift registers, counters, and LFSRs.

The example algorithms shown in this section use a shift register that is shifted by one position left in each loop iteration. The algorithms generate test patterns that guarantee that each TP bit is tested with 0 and 1 by inverting the base pattern one bit at a time. Therefore, the value of at least two bits changes between each loop iteration. There are four test methods:

- Latent fault detection in address protection logic
- Single point fault detection in address protection logic
- Latent fault detection in data protection logic
- Single point fault detection in data protection logic

The test methods have the following common features:

- Test methods can save and restore memory data modified during testing. This ensures that memory contents are preserved, including any errors present in the entries. Therefore, memory is not corrupted by generating valid protection codes for faulty data. If an error is present in the data, it is dealt with in the normal way during functional reads.
- Test methods can initialize the ECC or parity code field in memory entries before they are used. Therefore, no assumption is made that the protection code field is valid. Cache tag RAMs are normally initialized but cache data RAMs are not. Therefore, data RAMs may contain a mixture of initialized and uninitialized entries and the algorithms automatically handle both cases. This also ensures that any soft errors in the data do not affect the testing.
- Test methods can be suspended after each loop iteration to minimize the time that a memory is locked, reducing interrupt latency.
- Test methods use the error detection and correction logic to check for faults in the protection code generation logic and vice versa.

The example algorithms contain the following variables:

- Three data variables X, Y, and Z. These can contain the data and protection code fields.
- An address variable, A.
- A mask variable, XM.
- A data pattern variable `P` which is either BP or ~BP.
- A fixed data pattern FP. This can be any value.

Testing relies on the memory to be fault free. Therefore, a single event upset that occurs during testing might cause a test to fail. Therefore, if a test fails it must be run again to ensure that a single event upset did not cause the failure.

Microcode loops

The pseudo code for the algorithms shows how the protection logic is tested. These algorithms contain two loops, an inner loop to test each address or data bit and an outer loop to repeat the algorithm with the true and inverted base pattern. The microcode is an implementation of the pseudo code for execution by

PMC-100. Typically, an MBIST memory array is made up of multiple memory banks. Each bank has its own protection logic and a bank is selected using the MSBs of the address.

The pseudo code only shows how the protection logic is tested for one bank. Hence, to test the protection logic for all banks within an array, the microcode for a test algorithm must be repeated for each memory bank within an array. PMC-100 has a mode called bank address mode that allows efficient implementation of bank selection loops, see *4.15 Row address register, PMC100_RADDR* on page 4-67. This mode modifies the behavior of the LOOP-Last operation and the column address register, PMC100_CADDR.CA. The PMC100_CADDR.CA register acts as a loop counter and provides the address MSBs or LSBs, depending on the PMC100_CTRL.BAM value, to select the current bank. See section *4.15.2 Address output value, PMC100_CTRL.BAMEN=1* on page 4-68 for further details of how the address is generated when PMC100_CTRL.BAMEN is 1.

The outer pseudo code loop and bank selection loop are combined into one loop using the LOOP-Last operation and by initializing the PMC100_CADDR.CA value to twice the number of banks – 1. The pattern update operation, PUP, is used to control BP inversion for each iteration of the loop.

It contains the following sections:

C.1 Address Protection Logic Latent Fault Detection algorithm

This algorithm detects faults in the ECC and parity code generation and address error detection logic that prevent it from detecting faults in memory address decoder circuits. It generates addresses that are based on BP and injects one or more errors into each address bit by inverting them in turn by copying one entry to another. Therefore, it uses two memory entries, A XOR XM and A, which are the source and destination addresses of the copy operation respectively. The data field value is not important and is the value already present in entry A XOR XM. The XM variable is initialized to 1 for single address bit error injection.

The pseudocode for this algorithm operates as follows:

```
foreach A(BP, ~BP) {
    for(i=0; i<protected address width; i++) {
        1. Read RAM entry (A XOR XM) direct and store in X (save entry )
        2. Write FP to RAM entry (A XOR XM) through ECC or parity generation logic
( initialize entry )
        3. Read RAM entry A direct and store in Y (save entry )
        4. Read RAM entry (A XOR XM) direct and store in Z
        5. Write Z to RAM entry A direct (injects address error )
        6. Read RAM entry A through ECC or parity checking logic and checking logic and
check that a non-correctable ECC or party error is reported
        7. Write X to RAM entry (A XOR XM) direct (restore entry )
        8. Write Y to RAM entry A direct (restore entry )
        9. Read RAM entry (A XOR XM) direct and check that it is equal to X
        10. Read RAM entry A direct and check that it is equal to Y
        11. XM<<1 (generate next XM value )
        }
    }
```

C.1.1 Microcode

The microcode contains an outer loop that repeats the algorithm core twice the number of banks in the target array.

In addition to the standard register initialization and programming by software, the algorithm-specific programming for the array under test is shown in the following table.

Note

- Register DM is used as variable Z in the pseudocode
- To inject a double-bit error in the address XM can be set to `0x3` and LCR.LCI set to the number of address bits to be test minus 2

Table C-1 Address protection logic latent fault detection algorithm specific programming

Register/field	Programming
CTRL.DMDIS	`0b1`
CTRL.BAMEN	`0b1`
CTRL.ADDRID	`0b0`
CTRL.FP	Select required data value
CTRL.EXECO	`0b0`
MCR.CCW	Width of the bank select field in the address plus 1
MCR.RCW	Width of the physical SRAM address minus 2
LCR.LLEN	`0b0`
LCR.LCI	Number of address bits to be tested minus 1
AR.ARR	Memory controller and sub-array array encoding

Table C-1 Address protection logic latent fault detection algorithm specific programming (continued)

Register/field	Programming
AR.ARD	Direct SRAM access array MSBs, normally `0b00`
AR.ARG	ECC or parity generation logic array MSBs for writes
AR.ARE	ECC or parity error check logic array MSBs for reads
XM0-XM7	`0x1`
RADDR.RA	`0x0` (BP)
CADDR.CA	Start bank number, set to (start bank number << 1) + 1, to test all banks in array, set to ((number of banks-1)>>1) + 1 **Note** PMC100 _ CADDR.CA must be greater than (PMC100 _ CADDR.BNK_END << 1)
CADDR.BNK_END	End bank number, when testing all banks in an array this must be set to 0

The following table shows the microcode for address protection logic latent fault detection algorithm.

Table C-2 Microcode for address protection logic latent fault detection algorithm

Register	Instruction								Address output	Address update	Data polarity	Data	Transaction	Operation
P0	00	0	0	0	00	1	1		A XOR XM	Hold	-	X	Read	XORA. Save read data in X
P1	01	0	0	0	11	10	1111		A XOR XM	Hold	No inverse	FP	Write	XORA. Write FP through ECC/parity generation logic
P2	00	0	0	0	10	01	1111		A XOR XM	Hold	-	DM	Read	XORA. Save read data in DM
P3	00	0	0	0	01	01	0010		Address	Hold	-	Y	Read	Save read data in Y
P4	00	0	0	0	10	10	0011		Address	Hold	No inverse	DM	Write	Wait for register update and then write
P5	01	0	0	1	00	01	1001		Address	Hold	Data[3]	-	Read	Check uncorrectable error reported for entry A (PCHKUEF)
P6	01	0	0	0	00	01	1000		Address	Hold	Data[4]	-	Read	Check correctable error not reported for entry A (PCHKCEF)
P7	00	0	0	0	00	10	1111		A XOR XM	Hold	No inverse	X	Write	XORA. Restore X
P8	00	0	0	0	01	10	0000		Address	Hold	No inverse	Y	Write	Restore Y
P9	00	0	0	0	01	10	1111		A XOR XM	Hold	No inverse	X	Read	XORA. Check read data
P10	00	0	0	0	00	00	1100		-	Hold	-	-	None	SXM, shift left even loops, shift right odd loops

Table C-2 Microcode for address protection logic latent fault detection algorithm (continued)

Register	Instruction							Address output	Address update	Data polarity	Data	Transaction	Operation
P11	00	0	0	0	01	01	0111	Address	Hold	No inverse	Y	Read	Check read data, LOOP-LCR, LCR.LC-1
P12	00	0	1	0	00	00	0001	-	Update RADDR.RA	-	-	None	PUP.~RADDR
P13	00	0	1	0	00	00	0100	-	Updated CADDR.CA	-	-	None	LOOP-Last, CADDR-1

C.2 Address Protection Logic Single-point Detection algorithm

This algorithm detects faults in the code generation and address error detection logic that cause an error to be reported when the memory does not contain a fault in its address decoder circuits. It generates addresses that are based on BP by inverting each address bit in turn. The error detection logic is used to check for faults in the generation logic and vice versa. The data field value is not important and is a fixed pattern, FP. The XM variable is initialized to 1.

The pseudocode for this algorithm operates as follows:

```
foreach A(BP, ~BP) {
    for(i=0; i<protected address width; i++) {
        1. Read RAM entry (A XOR XM) direct and store in X (save entry)
        2. Write FP to RAM entry (A XOR AM) through ECC/parity generation logic (initialize
entry )
        3. Read RAM entry (A XOR AM) through ECC/parity checking logic and check that no
error is reported
        4. Write X to RAM entry ( A XOR AM) direct (restore entry )
        5. Read RAM entry (A XOR AM) direct and check that it is equal to X
        6. XM<<1 (generate next XM value)
        }
    }
```

C.2.1 Microcode

The microcode contains an outer loop that repeats the algorithm core twice the number of banks in the target array.

In addition to the standard register initialization and programming by software, see *4.4 PMC-100 programming* on page 4-37, the algorithm-specific programming for the array under test is shown in the following table.

In the following table, the instruction field are:

- PSEL
- AO
- UA
- DPOL
- DREG
- TRANS
- OP

Table C-3 Address protection logic single fault detection algorithm specific programming

Register/field	Programming
CTRL.DMDIS	`0b1`
CTRL.BAMEN	`0b1`
CTRL.ADDRID	`0b0`
CTRL.FP	Select required data value
CTRL.EXECO	`0b0`
MCR.CCW	Width of the bank select field in the address plus 1
MCR.RCW	Width of the physical SRAM address minus 2
LCR.LLEN	`0b0`
LCR.LCI	Number of address bits to be tested minus 1
AR.ARR	Memory controller and sub-array array encoding
AR.ARD	Direct SRAM access array MSBs, normally `0b00`

Table C-3 Address protection logic single fault detection algorithm specific programming (continued)

Register/field	Programming
AR.ARG	ECC or parity generation logic array MSBs for writes
AR.ARE	ECC or parity error check logic array MSBs for reads
XM0-XM7	`0x1`
RADDR.RA	`0x0` (BP)
CADDR.CA	Start bank number, set to (start bank number << 1) + 1, to test all banks in array, set to ((number of banks-1)>>1) + 1 ——— **Note** ——— PMC100 _ CADDR.CA must be greater than (PMC100 _ CADDR.BNK_END << 1) ———————
CADDR.BNK_END	End bank number, when testing all banks in an array this must be set to 0

The following table shows the microcode for address protection logic latent fault detection algorithm.

Table C-4 Microcode for address protection logic latent fault detection algorithm

Register	Instruction							Address output	Address update	Data polarity	Data	Transaction	Operation
PMC100_P0	00	0	0	0	00	01	1111	A XOR XM	Hold	-	X	Read	XORA. Save read data in PMC100_X
PMC100_P1	01	0	0	0	11	10	1111	A XOR XM	Hold	No inverse	FP	Write	XORA. Write FP through ECC/ parity generation logic
PMC100_P2	01	0	0	0	11	01	1000	A XOR XM	Hold	Data[4]	-	Read	XORA. Save read data in PMC100_DM
PMC100_P3	01	0	0	0	11	01	1001	A XOR XM	Hold	Data[5]	-	Read	Save read data in PMC100_Y
PMC100_P4	00	0	0	0	00	10	1111	A XOR XM	Hold	No inverse	X	Write	Wait for register update and then write
PMC100_P5	00	0	0	0	00	01	1111	A XOR XM	Hold	No inverse	X	Read	Check uncorrectable error reported for entry A (PCHKUEF)
PMC100_P6	00	0	0	0	00	00	1100	-	Hold	-	-	None	Check correctable error not reported for entry A (PCHKCEF)
PMC100_P7	00	0	0	0	00	00	0111	-	Hold	-	-	None	XORA. Restore PMC100_X

Table C-4 Microcode for address protection logic latent fault detection algorithm (continued)

Register	Instruction							Address output	Address update	Data polarity	Data	Transaction	Operation
PMC100_P8	00	0	1	0	00	00	0001	-	Update PMC100_RADDR.RA	-	-	None	Restore PMC100_Y
PMC100_P9	00	0	1	0	00	00	0100	-	Update PMC100_CADDR.CA	-	-	None	XORA. Check read data

C.3 Data Protection Logic Latent Fault Detection algorithm

This algorithm uses a data pattern, P, based on BP and injects errors into the data and the code fields. The algorithm detects faults in the code generation, error detection and correction logic that prevent it from detecting faults in code and data bits stored in memory. The address A variable can be initialized to any value, for example 0 and the XM variable is initialized to 1 for single bit error injection.

The pseudocode for this algorithm operates as follows:

```
foreach P (BP, ~BP) {
    for (i=0; i<data + code width; i++) {
        1. Read RAM entry A direct and store in X (save entry)
        2. Write P to RAM entry A via ECC generation logic (initialize entry)
        3. Read RAM entry A direct and store in Y
        4. Write (Y XOR XM) to entry A (inject error)
        5. Read RAM entry A via error detection logic and check that a correctable error is
indicated
        6. Read RAM entry A via the error correction logic and check that it is equal to Y
        7. Write X to RAM entry A (restore entry)
        8. Read RAM entry A and check that it is equal to X
        9. XM << 1 (generate next XM value)
        }
    }
```

C.3.1 Microcode

The microcode contains an outer loop that repeats the algorithm core twice the number of banks in the target array.

In addition to the standard register initialization and programming by software, the algorithm-specific programming for the array under test is shown in the following table.

In the following table, the instruction field are:
- PSEL
- AO
- UA
- DPOL
- DREG
- TRANS
- OP

Note

To inject a double-bit error in the address XM can be set to `0x3` in each data chunk and LCR.LCI set to the number of data bits to be tested in a chunk (including ECC/parity bits)-2.

Table C-5 Data protection logic latent fault detection algorithm specific programming

Register/field	Programming
CTRL.DMDIS	`0b0`
CTRL.BAMEN	`0b1`
CTRL.ADDRID	`0b0`
CTRL.FP	`0b00` (BP) - All 1s
CTRL.EXECO	`0b0`
MCR.CCW	Width of the bank select field in the address plus 1
MCR.RCW	Width of the physical SRAM address minus 2
LCR.LLEN	`0b0`

Table C-5 Data protection logic latent fault detection algorithm specific programming (continued)

Register/field	Programming
LCR.LCI	Number of data bits to be tested in a chunk (including ECC/parity bits) minus 1
AR.ARR	Memory controller and sub-array array encoding
AR.ARD	Direct SRAM access array MSBs, normally `0b00`
AR.ARG	ECC or parity generation logic array MSBs for writes
AR.ARE	ECC or parity error check logic array MSBs for reads
AR.ARC	ECC correction data logic array MSBs for reads
XM0-XM7	`0x1` in each data chunk (if multiple banks are read together)
RADDR.RA	`0x0`
CADDR.CA	Start bank number, set to (start bank number << 1) + 1, to test all banks in array, set to ((number of banks-1)>>1) + 1 ——— **Note** ——— PMC100 _ CADDR.CA must be greater than (PMC100 _ CADDR.BNK_END << 1) ———————
CADDR.BNK_END	End bank number, when testing all banks in an array this must be set to 0

The following table shows the microcode for data protection logic latent fault detection algorithm.

Table C-6 Microcode for data protection logic latent fault detection algorithm

Register	Instruction							Address output	Address update	Data polarity	Data	Transaction	Operation
PMC100_P0	00	0	0	0	00	01	0010	Address	Hold	-	X	Read	Save read data in X
PMC100_P1	01	0	0	0	11	10	0000	Address	Hold	CTRL.ADDRID	FP	Write	XORD, Write FP through ECC/parity generation logic
PMC100_P2	00	0	0	0	01	01	0010	Address	Hold	-	Y	Read	Save read data in Y
PMC100_P3	01	0	0	0	10	00	0011	-	Hold	-	Y	None	Wait for register update
PMC100_P4	00	0	0	0	01	10	1110	Address	Hold	No inverse	Y	Write	Y XOR XM (XORD)
PMC100_P5	00	0	0	0	00	10	0011	Address	Hold	Data[5]	-	Read	Check uncorrectable error not reported for entry A (PCHKUEF)
PMC100_P6	01	0	0	1	00	01	1000	Address	Hold	Data[2]	-	Read	Check correctable error reported for entry A (PCHKCEF)
PMC100_P7	00	0	0	0	00	01	0000	Address	Hold	No inverse	Y	Read	Check corrected data value
PMC100_P8	00	0	0	0	00	10	0000	Address	Hold	No inverse	X	Write	Restore X
PMC100_P9	00	0	0	0	00	00	1100	-	Hold	-	-	None	SXM, shift left even loops, shift right odd loops

Table C-6 Microcode for data protection logic latent fault detection algorithm (continued)

Register	Instruction							Address output	Address update	Data polarity	Data	Transaction	Operation
PMC100_P10	00	0	0	0	00	01	0111	Address	Hold	No inverse	X	Read	Check read data, LOOP-LCR, LCR.LC-1
PMC100_P11	00	0	0	0	00	00	0001	-	-	-	-	None	PUP. Invert CTRL.ADDRID
PMC100_P12	00	0	1	0	00	00	0100	-	Update CADDR	-	-	None	LOOP-Last, CADDR-1

C.4 Data Protection Logic Single-point Fault Detection algorithm

This algorithm uses a data pattern, P, based on BP and it inverts all bits of P in turn. The algorithm detects faults in the code generation, error detection and correction logic that cause an error to be reported when the code and data bits stored in memory do not contain an error. Variable A can be initialized to any value, for example 0 and the XM variable is initialized to 1.

The pseudocode for this algorithm operates as follows:

```
foreach P (BP, ~BP) {
    for (i=0; i<data field width; i++) {
        1. Read RAM entry A direct and store in X (save entry)
        2. Write (P XOR XM) to RAM entry A through ECC generation logic (initialize entry)
        3. Read RAM entry A via the error detection logic and check that no error is reported
        4. Read RAM entry A via the correction logic and check that it is equal to (P XOR XM)
        5. Write X to RAM entry A direct (restore entry)
        6. Read RAM entry X direct and check that it is equal to X
        7. XM << 1 (generate next XM value)
        }
    }
```

This section contains the following subsection:
- *C.4.1 Microcode* on page Appx-C-126.

C.4.1 Microcode

The microcode contains an outer loop that repeats the algorithm core twice the number of banks in the target array.

In addition to the standard register initialization and programming by software, the algorithm-specific programming for the array under test is shown in the following table.

In the following table, the instruction field are:
- PSEL
- AO
- UA
- DPOL
- DREG
- TRANS
- OP

Note

To inject a double-bit error in the address XM can be set to `0x3` in each data chunk and LCR.LCI set to the number of data bits to be tested in a chunk (including ECC/parity bits)-2.

Table C-7 Data protection logic latent fault detection algorithm specific programming

Register/field	Programming
CTRL.DMDIS	`0b0`
CTRL.BAMEN	`0b1`
CTRL.ADDRID	`0b0`
CTRL.FP	`0b00` (BP) - All 1s
CTRL.EXECO	`0b0`
MCR.CCW	Width of the bank select field in the address plus 1
MCR.RCW	Width of the physical SRAM address minus 2
LCR.LLEN	`0b0`

Table C-7 Data protection logic latent fault detection algorithm specific programming (continued)

Register/field	Programming
LCR.LCI	Number of data bits to be tested in a chunk (including ECC/parity bits) minus 1
AR.ARR	Memory controller and sub-array array encoding
AR.ARD	Direct SRAM access array MSBs, normally `0b00`
AR.ARG	ECC or parity generation logic array MSBs for writes
AR.ARE	ECC or parity error check logic array MSBs for reads
AR.ARC	ECC correction data logic array MSBs for reads
XM0-XM7	`0x1` in each data chunk (if multiple banks are read together)
RADDR.RA	`0x0`
CADDR.CA	Start bank number, set to (start bank number << 1) + 1, to test all banks in array, set to ((number of banks-1)>>1) + 1 **Note** PMC100 _ CADDR.CA must be greater than (PMC100 _ CADDR.BNK_END << 1)
CADDR.BNK_END	End bank number, when testing all banks in an array this must be set to 0

The following table shows the microcode for data protection logic latent fault detection algorithm.

Table C-8 Microcode for data protection logic latent fault detection algorithm

Register	Instruction							Address output	Address update	Data polarity	Data	Transaction	Operation
PMC100_P0	00	0	0	0	00	01	0010	Address	Hold	-	X	Read	Save read data in X
PMC100_P1	01	0	0	0	11	10	1110	Address	Hold	CTRL.ADDRID	FP	Write	Write FP XOR XM through ECC generation logic (XORD)
PMC100_P2	01	0	0	0	00	01	1001	Address	Hold	Data[5]	-	Read	Check uncorrectable error not reported for entry A (PCHKUEF)
PMC100_P3	01	0	0	0	00	01	1000	Address	Hold	Data[4]	-	Read	Check correctable error not reported for entry A (PCHKCEF)
PMC100_P4	11	0	0	0	11	01	1110	Address	Hold	CTRL.ADDRID	FP	Read	Check corrected data value is FP XOR XM (XORD)
PMC100_P5	00	0	0	0	00	10	0011	Address	Hold	No inverse	X	Write	Wait for register update and restore X
PMC100_P6	01	0	0	0	00	00	1100	-	Hold	-	-	None	SXM, shift left even loops, shift right odd loops

Table C-8 Microcode for data protection logic latent fault detection algorithm (continued)

Register	Instruction							Address output	Address update	Data polarity	Data	Transaction	Operation
PMC100_P7	00	0	0	0	00	01	0111	Address	Hold	No inverse	X	Read	Check read data, LOOP-LCR, LCR.LC-1
PMC100_P8	00	0	0	0	00	00	0001	-	-	-	-	None	PUP. Invert CTRL.ADDRID
PMC100_P9	00	0	1	0	00	00	0100	-	Update CADDR	-	-	None	LOOP-Last, CADDR-1

Appendix D
Miscellaneous Algorithms

This section describes miscellaneous algorithms.

It contains the following sections:

D.1 Memory scrubbing algorithm

This algorithm corrects single bit ECC errors in the SRAM and it assumes that the ECC code fields stored in SRAM are valid.

If necessary, it is possible to initialize the ECC code fields using a modified version of this algorithm, see section 7.8. As correction is rare, to save power, the algorithm does not update entries that do not have an error.

If an uncorrectable error is detected, then execution is halted, and test fail is indicated, which can be configured to interrupt the processor.

The software transparent memory scrubbing algorithm is as follows:

```
foreach (SRAM entry) {
        1. Read SRAM entry A via the ECC checking logic and record the error check signal
value.
        2. If ECC error check indicates a single bit error then read SRAM entry A via ECC
correction logic and record the corrected data value. Else end.
        3. Write the corrected data value to SRAM entry A via the ECC generation logic. This
corrects an error in the ECC or data field in the entry.
        4. Read SRAM entry A via the ECC checking logic and record the error check signal
value. Check that no error is indicated.
        }
```

D.1.1 Microcode

In addition to the standard register initialization and programming by software the algorithm-specific programming for the array under test is shown in the following table.

In the following table, the instruction field are:

- PSEL
- AO
- UA
- DPOL
- DREG
- TRANS
- OP

Table D-1 Memory scrubbing algorithm specific programming

Register/field	Programming
CTRL.DMDIS	`0b0`
CTRL.BAMEN	`0b0`
CTRL.ADDRID	`0b0`
CTRL.FP	`0b00` (BP) - All 1s
CTRL.EXECO	`0b0`
MCR.CCW	Width of the bank select field in the address plus 1
MCR.RCW	Width of the physical SRAM address minus 2
LCR.LLEN	`0b0`
LCR.LCI	Number of data bits to be tested in each data chunk (including ECC/parity bits) minus 1
AR.ARR	Memory controller and sub-array array encoding
AR.ARD	Direct SRAM access array MSBs, normally `0b00`
AR.ARG	ECC or parity generation logic array MSBs for writes

Table D-1 Memory scrubbing algorithm specific programming (continued)

Register/field	Programming
AR.ARE	ECC or parity error check logic array MSBs for reads
DM0-DM7	Valid data bit mask for each data chunk (excluding ECC/parity)
RADDR.RA	`0x0`
CADDR.CA	Start bank number, set to (start bank number << 1) + 1, to test all banks in array, set to ((number of banks-1)>>1) + 1 —— **Note** —— PMC100 _ CADDR.CA must be greater than (PMC100 _ CADDR.BNK_END << 1) ————
CADDR.BNK_END	End bank number, when testing all banks in an array this must be set to 0

The following table shows the microcode for memory scrubbing algorithm.

Table D-2 Microcode for memory scrubbing algorithm

Register	Instruction							Address output	Address update	Data polarity	Data	Transaction	Operation
PMC100_P0	01	0	0	0	00	01	1010	Address	Hold	No inverse	X	Read	Read protection error check result and set CTRL.NOTRANS if a single bit error is not indicated (PCHKCE)
PMC100_P1	11	0	0	0	00	01	0011	Address	Hold	No inverse	X	Read	Read corrected data value but wait for CTRL.NOTRANS to be updated first.
PMC100_P2	01	0	0	0	00	10	0011	Address	Hold	No inverse	X	Write	Write corrected data value through ECC generation.
PMC100_P3	01	0	0	0	01	01	1000	Address	Hold	Data[4]	Y	Read	Read protection error check result, fail if not corrected (PCHKCEF).
PMC100_P4	00	0	1	0	00	00	0100	-	Update	-	-	No operation	LOOP-Last

D.2 ECC/parity code field initialization algorithm

Before running the memory scrubbing algorithm, it is necessary to initialize the ECC/parity code fields stored in SRAM first. This normally this is not required for cache tag RAMs as they are initialized by cache invalidation but may be necessary for cache data RAMs. The algorithm will work with a mixture of initialized and uninitialized SRAM entries. Hence, valid data already stored in SRAM entries it will not be corrupted. Each entry is checked after it is initialized and if a correctable or un-correctable error is indicated then execution is halted, and test fail is indicated.

The algorithm is as follows:

```
foreach (SRAM entry) {
        1. Read SRAM entry A via ECC correction logic and record the corrected data value.
        2. Write the corrected data value to SRAM entry A via the ECC generation logic. This
generates a correct ECC code field for an uninitialized entry or writes the same ECC code
field back and data for an initialized entry.
        3. Read SRAM entry A via the ECC checking logic and record the error check signal
value. Check that no error is indicated.
        }
```

D.2.1 Microcode

The following table shows the microcode for memory initialization algorithm.

In the following table, the instruction field are:
- PSEL
- AO
- UA
- DPOL
- DREG
- TRANS
- OP

Table D-3 Microcode for memory initialization algorithm

Register	Instruction							Address output	Address update	Data polarity	Data	Transaction	Operation
PMC100_P1	11	0	0	0	00	01	0011	Address	Hold	No inverse	X	Read	Read corrected data value
PMC100_P2	01	0	0	0	00	01	0011	Address	Hold	No inverse	X	Write	Write corrected data value through ECC generation
PMC100_P3	01	0	0	0	01	10	1000	Address	Hold	Data[4]	Y	Read	Read protection error check result, fail if an error is indicated (PCHKCEF)
PMC100_P4	00	0	1	0	01	01	0100	-	Update	-	-	No operation	LOOP-Last

D.3 Memory dumping algorithm

This algorithm is used to dump memory to a debugger, typically if the IP core is in a deadlock state. It is assumed that the IP core is placed in production MBIST mode by the debugger using the CTRL.MRESET and CTRL.MREQ bits before executing this algorithm. Also, this algorithm uses software read triggered execution mode, which is enabled by setting CTRL.SRTEEN to 1. After PMC-100 is setup as described in this section, the debugger repeatedly reads the X data register to retrieve the value stored in each memory entry.

Note

After programming PMC-100, the debugger places itin the suspended state by writing CTRL.PEEN 1 and CTRL.PES 1.

The following table shows the memory dumping algorithm specific programming.

Table D-4 Memory dumping algorithm specific programming

Register/field	Programming
PMC100_CTRL.SRTEEN	`0b1`
PMC100_CTRL.DMDIS	`0b0`
PMC100_CTRL.BAMEN	`0b0`
PMC100_CTRL.ADDRID	`0b1`
PMC100_CTRL.EXECO	`0b0`
PMC100_MCR.CCW	`0x0`
PMC100_MCR.RCW	Width of the target array address minus 2
PMC100_LCR.LLEN	`0b0`
PMC100_AR.ARR	Memory controller and sub-array array encoding
PMC100_RADDR	Start address
PMC100_CADDR	`0x0`
PMC100_HIGHADDR	End address

The following table shows the microcode for memory dumping algorithm.

Table D-5 Microcode for memory dumping algorithm

Register	Instruction							Address output	Address update	Data polarity	Data	Transaction	Operation
PMC100_P0	00	0	0	0	00	01	0010	Address	Hold	No inverse	PMC100_X	Read	SAVERD
PMC100_P1	00	0	1	0	00	00	0100	-	Update	-	-	None	LOOP-Last

Appendix E
Signal descriptions

This appendix describes the PMC-100 signals.

It contains the following sections:

E.1 Clock and reset signals

The following table shows the PMC-100 clock and reset signals

Table E-1 Clock and reset signals

Signal	Direction	Description
CLKIN	Input	APB interface and processor clock. All signals in and out of PMC-100 are processed on the positive/rising edge of this clock.
nSYSRESET	Input	Active low reset signal.
DFTCGEN	Input	Override all internal architectural clock gates. **0** Normal operation **1** Architectural clock gates forced open

E.2 APB slave interface signals

The following table shows the PMC-100 APB slave interface signals.

Table E-2 APB slave interface signals

Signal	Direction	Description
PCLKEN	Input	Clock enable. Allows the APB interface to be driven from an interconnect that is clocked at a lower synchronous frequency to PMC-100. **Note** This signal must be tied HIGH if 1:1 clocking is used.
PADDR[11:2]	Input	Address. This is the APB address bus. **Note** PMC-100 only supports 32-bit transfers.
PPROT[2:0]	Input	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access. **Note** • If an IP core does not support secure accesses or security is not present in a configuration of an IP core, then **PPROT[1]** must be tied LOW. Otherwise **PPROT[1]** must be driven by the IP core. • PPROT[2] is not used by PMC-100
PSEL	Input	Select. This signal indicates that the slave device is selected and that a data transfer is required.
PENABLE	Input	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
PWRITE	Input	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
PWDATA[31:0]	Input	Write data. This bus is valid during write cycles when PWRITE is HIGH.

Table E-2 APB slave interface signals (continued)

Signal	Direction	Description
PSTRB[3:0]	Input	Write strobes. This signal indicates which byte lanes to update during a write transfer. There is one write strobe for each eight bits of the write data bus. Therefore, **PSTRB[n]** corresponds to **PWDATA[(8n + 7):(8n)]**. Write strobes must not be asserted during a read transfer. ——— **Note** ——— **PSTRB** is ignored by PMC-100 when writing to software registers but an error response is signaled on the **PSLVERR** signal for non-word write accesses.
PRDATA[31:0]	Output	Read Data. The selected slave drives this bus during read cycles when **PWRITE** is LOW.
PREADY	Output	Ready. The slave uses this signal to insert wait states to extend an APB transfer.
PSLVERR	Output	Error response. This signal is used by a slave to indicate a transfer failure.

E.3 MBIST master interface signals

The following table shows the PMC-100 MBIST master interface signals

Table E-3 MBIST master interface signals

Signal	Direction	Description
MBISTOLREQOL	Output	Request from MBIST controller to enter on-line MBIST mode.
MBISTOLACK	Input	MBIST mode acknowledgment Acknowledge from the processor indicating that it has entered MBIST mode and is ready to accept MBIST transactions.
MBISTOLADDR[MAWIDTH-1:0]	Output	MBIST transaction address.
MBISTOLINDATA[MDWIDTH-1:0]	Output	MBIST transaction write data.
MBISTOLOUTDATA[MDWIDTH-1:0]	Input	MBIST transaction read data.
MBISTOLWRITEEN	Output	MBIST transaction write enable. **Note** A *No Operation* (NOP) occurs if both read and write enables are de-asserted.
MBISTOLREADEN	Output	MBIST transaction read enable.
MBISTOLERR[MERWIDTH-1:0]	Input	**MBISTOLERR** input. The value of this signal is stored in the PMC100_MER register, see *4.19 MBISTOLERR input register, PMC100_MER* on page 4-73
MBISTOLPSEL[1:0]	Output	Used with **MBISTOLARRAY** and **MBISTOLADDR** to select ECC/parity logic associated with the target array for testing using on-line MBIST. This signal is controlled by the instruction PSEL field, see *4.22 Program registers, PMC100_P0-PMC100_P31* on page 4-77
MBISTOLPREN	Output	Protection error reporting enable. This signal is used during an MBIST read transaction to enable an IP core's protection error output bus to report an error detected during the transaction, see *4.5 Main control register, PMC100_CTRL* on page 4-40
MBISTOLWADDR[MAWIDTH-1:0]	Output	Write address. This used when testing two-port memories. In this case, MBISTOLADDR is used as the memory read address, allowing read and write transactions to be performed in parallel, see *3.3 Two-port SRAM support* on page 3-23.
MBISTOLARRAY[MARWIDTH-1:0]	Output	MBIST memory array and protection logic selector.
MBISTOLBE[MBWIDTH-1:0]	Output	MBIST transaction byte enables.
MBISTOLCFG[MCWIDTH-1:0]	Output	MBISTOLCFG controls might include example attributes such as AllMode or LATENCY/SETUP controls for the array under test. **Note** During on-line MBIST testing the LATENCY/SETUP bits may be set HIGH but the AllMode bits must be set LOW.

E.4 Execution control and status signals

The following table shows the PMC-100 execution and control status signals.

Table E-4 Execution control and status signals

Signal	Direction	Description
TEN	Input	Test enable. This is the master hardware enable for PMC-100.
TC	Input	Test continue pulse signals. Test continue pulse. This is a single cycle pulse and when enabled by the PMC100_CTRL.TCSEN bit, causes a suspended test to continue execution.
TE	Output	Test ended. When PMC100_CTRL.TESEN bit is 1 this signal indicates that the test program has completed.
TF	Output	Test failed. When PMC100_CTRL.TFSEN bit is 1 this signal indicates that a memory fault has been detected.
TA	Output	Test active. This signal indicates that PMC-100 is in the run or suspended state. This signal can be connected to the clock and power control logic within the processor to prevent it powering down during a test, but the internal clocks may be gated between test bursts.

E.5 Miscellaneous signals

The following table shows the PMC-100 miscellaneous signals

Table E-5 Execution control and status signals

Signal	Direction	Description
CFGDEVAFF[63:0]	Input	PMC-100 affinity value. This value can be read from the PMC100_DEVAFF0 and PMC100_DEVAFF1 registers.
ECOREVNUM[3:0]	Input	ECO revision number. This is the value of the RevAnd field in the Peripheral ID3 register.
AUXIN[AIWIDTH-1:0]	Input	Auxiliary input. The value of this signal is stored in the PMC100_AIR register.
AUXOUT[AOWIDTH-1:0]	Output	Auxiliary output. The value of this signal is set by the PMC100_AOR register.
FPERR	Output	Flop parity error. Parity error from the flip-flop protection logic. When an error is detected, this signal will be assured for one or more clock cycles. When the FLOPPARITY parameter is 0, FPERR is driven low.

Appendix F
PMC-100 software library

This section describes the PMC-100 software library.

It contains the following sections:

F.1 PMC-100 software library overview

The PMC-100 software library provides a suite of functions that program a PMC-100 programmable MBIST controller to perform SRAM and ECC logic test algorithms on an attached IP core.

The library includes the following functions:

- Perform memory error injection
- Test an IP core error reposting bus
- Perform PMC-100 self-testing
- Dump memory to a debugger
- Scrub memory
- Initialize memory
- Initialize PMC-100 registers
- Check the PMC-100 ID
- Provide the PMC-100 software library version
- Check the results of a test

Tests can be run on-line during functional operation. All tests, except for PMC100_MarchCm and PMC100_Memory_Init, do not corrupt the contents of the target memory. The tests are intended to be run periodically as part of an STL or during poweron or poweroff.

F.2 PMC-100 software library configuration and usage

This section describes how the library is configured for a particular IP core and how it can be used as part of the execution testbench for the core. All core execution testbenches should contain three PMC-100 test called `pmc100_all.c`, `pmc100_ecc.c`, and `pmc100_checktest.c`, that demonstrate the usage of the library in the context of the IP core. For more information, see the *Integration and Implmentation Manual* or *Configuration and Integration Manual* for the IP core that you will be using PMC-100 with.

The execution testbench supports both logic simulation and fault simulation using the example `pmc100_ecc.c` and `pmc100_checktest.c` tests. `The pmc100_all.c` test is only intended for use with logic simulation.

The PMC-100 software library is configured for an IP core memory configuration by a C header file that is rendered by the `<core>_pmc100_render_memory_info.pl` and generates scripts that are delivered with an IP core.

The PMC-100 software library supports two use models:

1. Self-use – This use model is for tests run on a processor to test itself; for example, with STLs and execution TB tests.
2. External-use – This use model is for tests run on an external processor; for example, with a safety agent that performs testing of other IP cores in a system.

Two PMC-100 software library configuration C header files are rendered, one for each use model:

1. `core_pmc100_mem_description.h` – This file is for the self-use model and uses IP core agnostic generic file and object names. All IP cores will render a header file that the same file and object names.
2. `<core>_pmc100_mem_description<unique>.h` – This file is for the external-use model and uses IP core specific file and object names. This allows multiple PMC-100 MBIST controllers to be controlled from one application. As such, multiple different types of IP cores with different memory configurations can be supported from one application at the same time.

See *Context structure* on page Appx-F-149 for an example of how to use the PMC-100 software library configuration C header file in your application.

F.2.1 Release directory structure and file overview

The file structure that is delivered as follows:

```
<core>/logical/
|_<core>
|       |_pmc_files
|              |_<core>_pmc100_render_memory_info.pl
|              |_core_pmc100_mem_description.h (rendered)
|              |_<core>_pmc100_mem_description<unique>.h (rendered)
|_testbench/execution_tb
|       |_tests
|              |_pmc100_ecc.c
|              |_pmc100_all.c
|              |_pmc100_checktest.c
|              |_<core>_pmc100_wrapper.h
```

Note

Execution testbench files which are required for fault simulation are not shown.

The following table describes the IP core software library configuration and execution testbench file descriptions.

Table F-1 Software library file overview

File name	Description
`<core>_pmc100_render_memory_info.pl`	This generates the `<core>_pmc100_mem_description.h` file. This script is used by your IP core's render script. Therefore, it is not intended for you to use directly.
`core_pmc100_mem_description.h`	This is a C header file that configures the PMC-100 software library. It contains information that describes the memory configuration of the IP core. This is for the self-use model and uses generic file and object names. It is intended for use by an STL. This file is rendered by the `<core>_pmc100_render_memory_info.pl` script.
`<core>_pmc100_mem_description<unique>.h`	This is a C header file that configures the PMC-100 software library. It contains information that describes the memory configuration of the IP core. This is for the external-use model and is intended for use by a safety island processor that controls multiple PMC-100 MBIST controllers. This file is rendered by the `<core>_pmc100_render_memory_info.pl` script.
`pmc100_all.c`	This is an execution TB test that demonstrates the use of all the library functions on all memories in an IP core. **Note** This test is not intended to be used in fault simulation.
`pmc100_ecc.c`	This is an execution TB test that tests all the ECC logic in an IP core. **Note** This test is intended to be used in fault simulation.
`pmc100_checktest.c`	This is an execution TB test that just checks the ID code of the PMC-100 and therefore runs quickly.
`<core>_pmc100_wrapper.h`	This is an execution testbench header file that contains wrapper functions for the PMC-100 software library functions that include setup for the SBIST controller, which is needed for fault simulation.

Warning

Do not modify the `<core>_pmc100_render_memory_info.pl`, `core_pmc100_mem_description.h`, and `<core>_pmc100_mem_description<unique>.h` files.

The following diagram shows how `<core>_pmc100_mem_description.h` is generated by an IP core main render script. It also shows an example of the structure of an execution testbench test that uses the PMC-100 software library.

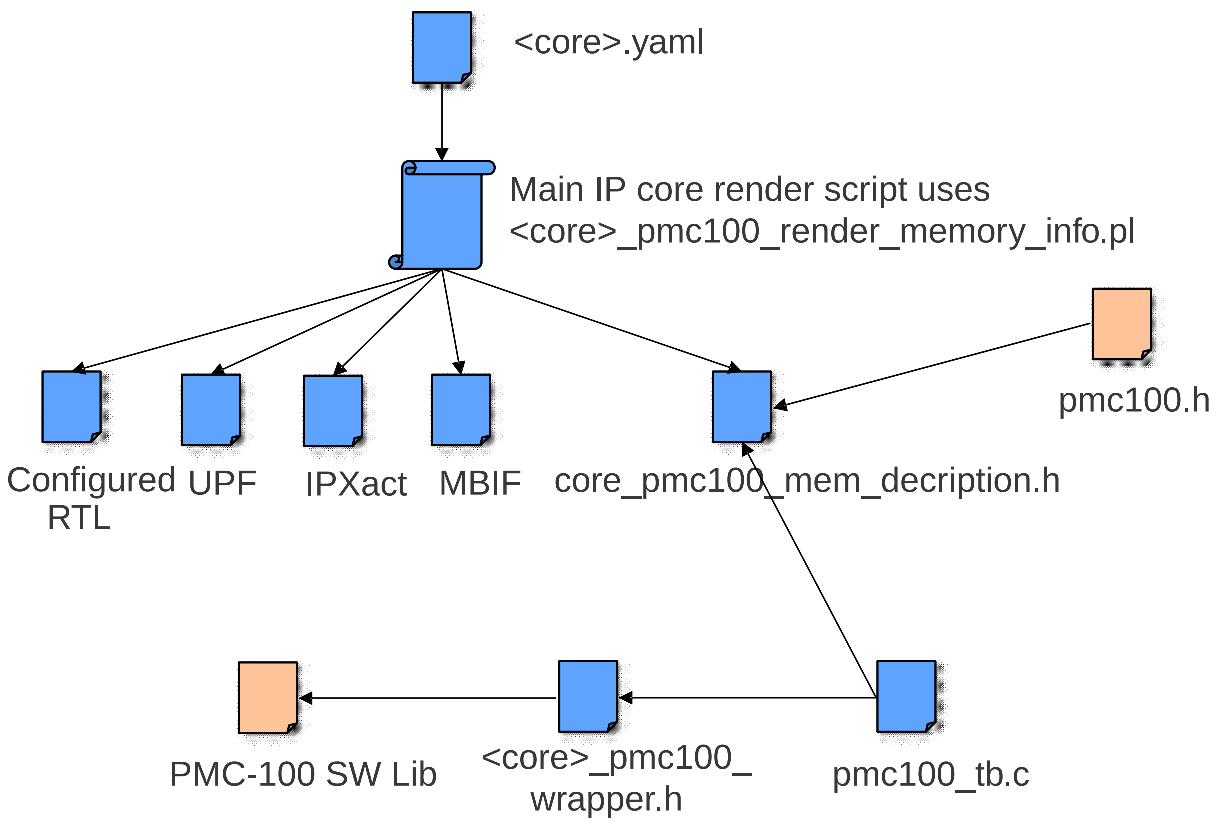

<core>.yaml
Main IP core render script uses <core>_pmc100_render_memory_info.pl
pmc100.h
Configured RTL
UPF
IPXact
MBIF
core_pmc100_mem_decription.h
PMC-100 SW Lib
<core>_pmc100_ wrapper.h
pmc100_tb.c

Figure F-1 IP core memory description file render flow and include file relationships

F.2.2 PMC-100 software library source code file structure

The source code file structure that the software library uses is as follows:

```
|_shared
    |_pmc100
        |_logical
        |_shared
        |_software
            |_api
            | |_pmc100_api1.c
            | |_pmc100_api2.c
            | |_pmc100_api1.h
            | |_pmc100_api2.h
            | |_pmc100_functions_common.c
            | |_pmc100_functions_common.h
            | |_pmc100_functions1.c
            | |_pmc100_functions1.h
            | |_pmc100_functions2.c
            | |_pmc100_functions2.h
            | |_pmc100.h
            |_driver
            | |_pmc100_drv.c
            | |_pmc100_drv.h
            |_test
            | |_main_threadx.c
```

The following table describes the software library source code files.

Table F-2 Software library source code file overview

File name	Description
`pmc100_api1.[ch]`	Contains declaration and implementation of API group 1
`pmc100_api2.[ch]`	Contains declaration and implementation of API group 2

Table F-2 Software library source code file overview (continued)

File name	Description
`pmc100_functions1.[ch]`	Contains declaration and implementation of helper function for API group 1
`pmc100_functions2.[ch]`	Contains declaration and implementation of helper function for API group 2
`pmc100_functions_common.[ch]`	Contains declaration and implementation of common helper function for API group 1 and 2
`pmc100.h`	Contains the definition of library data structures
`pmc100_drv.[ch]`	Example platform independent reference interrupt driven driver and example of PMC-100 library API usage
`main_threadx.c`	Reference example of using PMC-100 driver in ThreadX OS

Normally, the `pmc100` directory should be placed in the same directory that your core. The library is broken down into two groups of functions in `pmc100_api1` and `pmc100_api2`, respectively. You can use `pmc100_api1` on its own but `pmc100_api2` must be used together with `pmc100_api1`.

In working with these files, consider the following recommendations:
1. You must not modify the files in the api directory.
2. The files in the driver and test directories are examples and therefore, you may modify and copy them as required.
3. The API is defined by `pmc100.h`, `pmc100_api1.h`, and `pmc100_api2.h`. Your software must only reference these files from the library. The other functions and data types in the library are considered private and therefore, must not be used directly.
4. The files in the driver and test directories are provided as examples and can be used as required.

F.3 PMC-100 software library data structures

This section describes the PMC-100 software library memory, parameter, and context structures.

F.3.1 pmc100.h data structures

Below are the data structures defined in the `software/api/pmc100.h` header file.

IP core memory information structure

This structure holds a memory information for each type of memory within a configuration of the IP core; for example, instruction tag, data tag, instruction cache, and data cache.

These memories are the same as the memories specified in the production MBIST documentation provided with your IP core. This data structure contains additional information about the ECC logic associated with each memory.

All fields are read-only except `ccw` and `cfgr`, which you can modify from your application software. This may be useful when using the SRAM tests `PMC100_ShortBurst_1port`, `PMC100_ShortBurst_2port`, and `PMC100_MarchCm`.

```
/* Definition of the memory structure */
typedef struct {
  const uint32_t options;              // Memory specific options
  const uint32_t mcr;                  // Pipeline depth and cycles per operation
  const uint32_t haddr;                // High address
  const uint32_t laddr;                // Low address
  const uint32_t addrw;                // Address width
  uint32_t ccw;                        // RAM column address width. User modifiable.
  const uint32_t banks_number;         // Number of RAM banks memory consists of
  const uint32_t bank_width;           // Width of bank value -1
  const uint32_t valid_bits;           // RAM data field width (not including ECC bits)
  const uint32_t addr_protected_bits; // Number of address bits protected by ECC scheme
  const uint32_t dm_ecc[8];            // Data mask with ECC fields
  const uint32_t dm_noecc[8];          // Data mask without ECC fields
  const uint32_t xm[8];                // XOR mask
  const uint32_t ecc_num_units;        // The number of ECC units, ecc_ar elements
  const uint32_t ecc_ar[4];            // AR register value for each ECC unit
  uint32_t cfgr;                       // CFGR register value, MBISTOLCFG output. User
modifiable.
} Pmc100MemInfo_type;
```

Where memory specific options are each represented by a single bit, allowing multiple options to be set together, as follows:

- Error correction present – bit[0]
- Cache corkscrew mode required – bit[1]
- Reserved bits[31:2]

Parameter structure

This structure holds various parameters that are used by the PMC-100 software library. All the fields in this structure are read-only, except `PMC100_BASE`, which you may modify in your application software if you are accessing the PMC-100 via an interface that is external to your processor. Therefore, the system determines the base address of each PMC-100 your software accesses.

```
/*****************************************************************************/
/* PMC100 parameters structure                                               */
/*                                                                           */
/* This structure should be populated with CPU specific parameters generated  */
/* during rendering process. This data structure should be the part of PMC    */
/* API library context, thus the parameters would be visible to the APIs.     */
/*****************************************************************************/
 typedef struct {
   uint32_t PMC100_BASE; /* PMC100 base address. User modifiable */
   const    uint32_t REVAND;
   const    uint32_t REVISION;
   const    uint32_t DEVID_MBWIDTH; /* MBIST byte enable width */
   const    uint32_t DEVID_MERWIDTH; /* MBIST register and signal width */
   const    uint32_t DEVID_MARWIDTH; /* MBIST array width */
   const    uint32_t DEVID_MDWIDTH; /* MBIST data width */
   const    uint32_t DEVID_MAWIDTH; /* MBIST address width */
   const    uint32_t DEVID_RCOWIDTH; /* RAM cycles of operation field width */
   const    uint32_t DEVID_AOWIDTH; /* AOR register and AUXOUT signal width */
   const    uint32_t DEVID_AIWIDTH; /* AIR register and AUXIN signal width */
   const    uint32_t DEVID_PDWIDTH; /* Pipeline depth field width */
   const    uint32_t DEVID_PROGSIZE; /* Program size */
   const    uint32_t DEVID_MCWIDTH; /* MBIST configuration width */
   const    uint32_t DEVID;
   const    uint32_t DEVID1;
   const    uint32_t DEVID2;
   const    uint32_t NUM_OF_DATA_WORDS;
   const    uint32_t BANK_SEL_WIDTH;
   const    uint32_t NUM_OF_MEMORIES;
   const    uint32_t CTRL_RW_MASK;
   const    uint32_t CFGR_RW_MASK;
   const    uint32_t MCR_RW_MASK;
   const    uint32_t AR_RW_MASK;
   const    uint32_t BER_RW_MASK;
   const    uint32_t PCR_RW_MASK;
   const    uint32_t HIGHADDR_RW_MASK;
   const    uint32_t LOWADDR_RW_MASK;
   const    uint32_t CADDR_RW_MASK;
   const    uint32_t RADDR_RW_MASK;
   const    uint32_t AIR_RW_MASK;
   const    uint32_t AOR_RW_MASK;
   const    uint32_t MER_RW_MASK;
   const    uint32_t LCR_RW_MASK;
   const    uint32_t LSCR_RW_MASK;
   const    uint32_t TCCR_RW_MASK;
} Pmc100Params_type;
```

Register address offset structure

This structure holds the address offsets of the PMC-100 registers. This is required if you write your own functions to program PMC-100 directly. It is defined in the `software/api/pmc100.h` header file. This file also defines macros for setting the `PMC100_CTRL` bits and provides the `PMC-100 CoreSight ID` register values.

```
/******************************************************************************/
/*                      PMC100 register address offset structure              */
/******************************************************************************/
typedef struct
{
 PMC100__IO  uint32_t CTRL;           /* Offset 0x000     (RW) Control
Register                 */
 PMC100__IO  uint32_t MCR;            /* Offset 0x004     (RW) Memory Control
Register         */
 PMC100__IO  uint32_t BER;            /* Offset 0x008     (RW) Byte Enable
Register             */
 PMC100__IO  uint32_t PCR;            /* Offset 0x00C     (RW) Program Control
Register         */
 PMC100__I   uint32_t RPR;            /* Offset 0x010     (RO) Read Pipeline
Register           */
 PMC100__IO  uint32_t HIGHADDR;       /* Offset 0x014     (RW) High Address
Register            */
 PMC100__IO  uint32_t CADDR;          /* Offset 0x018     (RW) Column Address
Register          */
 PMC100__IO  uint32_t RADDR;          /* Offset 0x01C     (RW) Row Address
Register             */
 PMC100__IO  uint32_t AIR;            /* Offset 0x020     (RW) Auxiliary Input
Register         */
 PMC100__IO  uint32_t AOR;            /* Offset 0x024     (RW) Auxiliary Output
Register        */
 PMC100__IO  uint32_t MER;            /* Offset 0x028     (RW) MBISTOLERR Input
Register        */
 PMC100__IO  uint32_t LSPR;           /* Offset 0x02C     (RW) Loop Start
Register              */
 PMC100__IO  uint32_t LCR;            /* Offset 0x030     (RW) Loop Counter
Register            */
 PMC100__IO  uint32_t AR;             /* Offset 0x034     (RW) Array
Register                   */
 PMC100__IO  uint32_t CFGR;           /* Offset 0x038     (RW) MBISTOLCFG Output
Register       */
 PMC100__IO  uint32_t TCCR;           /* Offset 0x03C     (RW) Test Continue Counter
Register   */
 PMC100__IO  uint32_t LOWADDR;        /* Offset 0x040     (RW) Low Address
Register             */
 PMC100__IO  uint32_t LSCR;           /* Offset 0x044     (RW) Loop Suspend Counter
Register    */
 PMC100__I   uint32_t RESERVED0[14];  /* Offset
0x048-07C                                        */
 PMC100__IO  uint32_t X[8];           /* Offset 0x080-09C (RW) Data Register
Xx                 */
 PMC100__I   uint32_t RESERVED1[24];  /* Offset
0x0A0-0FC                                     */
 PMC100__IO  uint32_t Y[8];           /* Offset 0x100-11C (RW) Data Register
Yx                 */
 PMC100__I   uint32_t RESERVED2[24];  /* Offset
0x120-17C                                     */
 PMC100__IO  uint32_t DM[8];          /* Offset 0x180-19C (RW) Data Mask Register
DMx            */
 PMC100__I   uint32_t RESERVED3[24];  /* Offset
0x1A0-1FC                                     */
 PMC100__IO  uint32_t XM[8];          /* Offset 0x200-21C (RW) XOR mask Register
XMx             */
 PMC100__I   uint32_t RESERVED4[24];  /* Offset
0x220-27C                                     */
 PMC100__I   uint32_t RESERVED5[32];  /* Offset
0x280-2FC                                     */
 PMC100__IO  uint32_t P[32];          /* Offset 0x300-37C (RW) Program Register
Px               */
 PMC100__I   uint32_t RESERVED6[736]; /* Offset 0x380-
EFC                                     */
 PMC100__I   uint32_t ITCTRL;         /* Offset 0xF00     (RO) Integration Mode Control
Register*/
 PMC100__I   uint32_t RESERVED7[39];  /* Offset 0xF04-
F9C                                     */
 PMC100__IO  uint32_t CLAIMSET;       /* Offset 0xFA0     (RW) Claim tag set
Register           */
 PMC100 __IO  uint32_t CLAIMCLR;       /* Offset 0xFA4     (RW) Claim Tag Clear
Register         */
 PMC100__I   uint32_t DEVAFF0;        /* Offset 0xFA8     (RO) Device Affinity 0
Register       */
 PMC100__I   uint32_t DEVAFF1;        /* Offset 0xFAC     (RO) Device Affinity 1
Register       */
 PMC100__O   uint32_t LAR;            /* Offset 0xFB0     (WO) Lock Access
Register             */
 PMC100__I   uint32_t LSR;            /* Offset 0xFB4     (RO) Lock Status
Register             */
 PMC100__I   uint32_t AUTHSTATUS;     /* Offset 0xFB8     (RO) Authentication Status
Register   */
 PMC100__I   uint32_t DEVARCH;        /* Offset 0xFBC     (RO) Device Architecture
Register     */
 PMC100__I   uint32_t DEVID2;         /* Offset 0xFC0     (RO) Device ID
Register               */
 PMC100__I   uint32_t DEVID1;         /* Offset 0xFC4     (RO) Device ID
Register               */
 PMC100__I   uint32_t DEVID;          /* Offset 0xFC8     (RO) Device ID
Register               */
 PMC100__I   uint32_t DEVTYPE;        /* Offset 0xFCC     (RO) Device Type
Register             */
 PMC100__I   uint32_t PIDR4;          /* Offset 0xFD0     (RO) Peripheral ID
Register           */
 PMC100__I   uint32_t PIDR5;          /* Offset 0xFD4     (RO) Peripheral ID
Register           */
 PMC100__I   uint32_t PIDR6;          /* Offset 0xFD8     (RO) Peripheral ID
Register           */
 PMC100__I   uint32_t PIDR7;          /* Offset 0xFDC     (RO) Peripheral ID
Register           */
 PMC100__I   uint32_t PIDR0;          /* Offset 0xFE0     (RO) Peripheral ID
Register           */
 PMC100__I   uint32_t PIDR1;          /* Offset 0xFE4     (RO) Peripheral ID
Register           */
 PMC100__I   uint32_t PIDR2;          /* Offset 0xFE8     (RO) Peripheral ID
Register           */
 PMC100__I   uint32_t PIDR3;          /* Offset 0xFEC     (RO) Peripheral ID
Register           */
 PMC100__I   uint32_t CIDR0;          /* Offset 0xFF0     (RO) Component ID
Register            */
 PMC100__I   uint32_t CIDR1;          /* Offset 0xFF4     (RO) Component ID
Register            */
 PMC100__I   uint32_t CIDR2;          /* Offset 0xFF8     (RO) Component ID
Register            */
 PMC100__I   uint32_t CIDR3;          /* Offset 0xFFC     (RO) Component ID
Register            */
} Pmc100_type;
```

Context structure

This structure defines the context of the library. The pointer to this library context is the first parameer of most the API functions.

```
/* PMC context */
typedef struct
```

```
{
   /* The array of structures which contains
    * all the IP Core specific memory and memory
    * protection logic configuration information required
    * by the tests.
    */
   Pmc100MemInfo_type *mem_array;

   /* This structure holds the PMC-100 instance parameter values and other
    * related values used by PMC-100 library.  All the fields in this
    * structure are read only.
    */
   Pmc100Params_type *params;

   /* PMC100_MER mask. Internal variable used by library.
    * Initialized by library functions.
    */
   uint32_t mer_error;
} Pmc100Context_type;
```

The Library assumes that the `mem_array` field contains the pointer to the first element of the memory structure array that holds a memory information for each type of memory generated during the rendering process. The Library assumes that the `params` field contains the pointer to thePMC-100 specific parameters for the IP core that contains it.

Below is example code that shows how to create the PMC-100 Library context in your application.

```
#include "pmc100_api1.h"
#include "pmc100_api2.h"
#include "core_pmc100_mem_description.h"

int main(void)
{
        . . .
        . . .
        /* PMC-100 Library context */
        Pmc100Context_type pmc100_ctx;

        /* init pmc100 lib context
        * the context data instances are defined in auto-generated file
        * core_pmc100_mem_description.h
        */
        pmc100_ctx.mem_arrray = (Pmc100MemInfo_type*) &core_mem_pmc100[0];
        pmc100_ctx.params = (Pmc100Params_type*) &core_params_pmc100;
        . . .
        . . .

}
```

F.3.2 Example core_pmc100_mem_description.h file

Below is an example of the `core_pmc100_mem_description.h` file from the Arm *Cortex-M55* processor.

Note

1. The `core_mem_pmc100` and `core_params_pmc100` variables are declared as const, allowing them to be accessed from read only memory. Some fields in these types are not defined as const but the declarations override this. Pointers to these variables must be typecast when used, see section *Context structure* on page Appx-F-149 for an example of how to do this. In the self-use model, it is not normally necessary for an application to modify the contents of these variables but if this is required then the `<core>_pmc100_mem_description<unique>.h` file should be used.
2. The `core_ram_enum` enumerated type contains a list of identifiers, one for each memory within an IP core.
3. All memory identifiers will always be present in this enum regardless of if a memory is present in a particular configuration of an IP core or not.

4. The identifiers for memories that are not present within a particular configuration of an IP core are assigned a value of -1.
5. The order and constant values assigned to each memory identifier may vary depending on the memory configuration of an IP core.

```
#include "pmc100.h"

// Enumeration that can be used to select the required memory from the Pmc100MemInfo_type
array.
enum core_ram_enum {ITAG=0, IDATA=1, DTAG=2, DDATA=3, ITCM=4, DTCM=5} ram_type;

/* Declaration of an array of memories structures. Each structure contains all the IP Core
specific memory and memory protection logic configuration information required by the tests.
*/
const Pmc100MemInfo_type core_mem_pmc100[6] = {
    /**********
     * ITAG   *
     **********/
    {
        0x0, /* ITAG_OPTIONS */
        131, /* ITAG_MCR */
        1023, /* ITAG_HADDR */
        0, /* ITAG_LADDR */
        10, /* ITAG_ADDRW */
        0, /* ITAG_CCW */
        2, /* ITAG_BANKS */
        1, /* ITAG_BANKSW */
        25, /* ITAG_VALID_BITS */
        9, /* ITAG_ADDR_PROTECTION */
        /* dm_ecc array */
        {
            0x0ffffff8, /* ITAG_DM0_ECC */
            0, /* ITAG_DM1_ECC */
            0, /* ITAG_DM2_ECC */
            0,
            0,
            0,
            0,
            0
        },
        /* dm_noecc array */
        {
            0x003ffff8, /* ITAG_DM0_NOECC */
            0, /* ITAG_DM1_NOECC */
            0, /* ITAG_DM2_NOECC */
            0,
            0,
            0,
            0,
            0
        },
        /* xm array */
        {
            8, /* ITAG_XM0 */
            0, /* ITAG_XM1 */
            0, /* ITAG_XM2 */
            0,
            0,
            0,
            0,
            0
        },
        1, /* ITAG_ECC_NUM_UNITS */
        /* ecc_ar array */
        {
            4, /* ITAG_ECC_AR */
            0,
            0,
            0
        },
        0 /* ITAG_CFGR */
    },
    /**********
     * IDATA  *
     **********/
    {
        0x2, /* IDATA_OPTIONS */
        131, /* IDATA_MCR */
        4095, /* IDATA_HADDR*/
        0, /* IDATA_LADDR */
        12, /* IDATA_ADDRW */
        0, /* IDATA_CCW */
        2, /* IDATA_BANKS */
        1, /* IDATA_BANKSW */
        38, /* IDATA_VALID_BITS */
        12, /* IDATA_ADDR_PROTECTION */
        /* dm_ecc array */
…
…
    }
};


/******************************************************************************/
/* PMC100 parameters structure                                                */
/******************************************************************************/
const Pmc100Params_type core_params_pmc100 = {
    0xE0046000, /* YAMIN PMC100 base register address */
    0x0, /* REVAND */
    0x0, /* REVISION */
    10, /* DEVID_MBWIDTH */
    5, /* DEVID_MERWIDTH */
    5, /* DEVID_MARWIDTH */
…
…
    0x1f, /* MER_RW_MASK */
    0x80ff0000, /* LCR_RW_MASK */
    0x80ff0000, /* LSCR_RW_MASK */
    0xffff0000 /* TCCR_RW_MASK */
};
```

F.3.3 Example <core>_pmc100_mem_description<unique>.h file

This shows an example of the `<core>_pmc100_mem_description<unique>.h` file for the Yamin processor, called `yamin_pmc100_mem_description<UNIQUE>.h`. The `<UNIQUE>` string is provided by the `UNIQUE` value in the `yamin.yaml` file. This `UNIQUE` string is also used in the variable names in the header file shown below. The header file and variable declarations have unique names, allowing them to be used in applications that access the PMC-100 in multiple processors. This is known as the external-use model.

Note

1. The `<core><UNIQUE>_mem_pmc100` and `<core><UNIQUE>_params_pmc100` variables have fields that may be modified by an application and therefore must be placed in read/write memory.
2. The `<core><UNIQUE>_ram_enum` enumerated type contains a list of identifiers, one for each memory within an IP core.
3. All memory identifiers will always be present in this enum regardless of if a memory is present in a particular configuration of an IP core or not.
4. The identifiers for memories that are not present within a particular configuration of an IP core are assigned a value of -1.
5. The order and constant values assigned to each memory identifier may vary depending on the memory configuration of an IP core.

```
#include "pmc100.h"

// Enumeration that can be used to select the required memory from the Pmc100MemInfo_type
array.
enum yamin<UNIQUE>_ram_enum {ITAG=0, IDATA=1, DTAG=2, DDATA=3, ITCM=4, DTCM=5}} ram_type;

/* Declaration of an array of memories structures. Each structure contains all the IP Core
specific memory and memory protection logic configuration information required by the tests.
*/
Pmc100MemInfo_type yamin<UNIQUE>_mem_pmc100[6] = {
    /**********
     * ITAG   *
     **********/
    {
        0x0, /* ITAG_OPTIONS */
        131, /* ITAG_MCR */
        1023, /* ITAG_HADDR */
        0, /* ITAG_LADDR */
        10, /* ITAG_ADDRW */
        0, /* ITAG_CCW */
        2, /* ITAG_BANKS */
        1, /* ITAG_BANKSW */
        25, /* ITAG_VALID_BITS */
        9, /* ITAG_ADDR_PROTECTION */
        /* dm_ecc array */
        {
            0x0ffffff8, /* ITAG_DM0_ECC */
            0, /* ITAG_DM1_ECC */
            0, /* ITAG_DM2_ECC */
            0,
            0,
            0,
            0,
            0
        },
        /* dm_noecc array */
        {
            0x003ffff8, /* ITAG_DM0_NOECC */
            0, /* ITAG_DM1_NOECC */
            0, /* ITAG_DM2_NOECC */
            0,
            0,
            0,
            0,
            0
        },
        /* xm array */
        {
            8, /* ITAG_XM0 */
            0, /* ITAG_XM1 */
            0, /* ITAG_XM2 */
            0,
            0,
            0,
            0,
            0
        },
        1, /* ITAG_ECC_NUM_UNITS */
        /* ecc_ar array */
        {
            4, /* ITAG_ECC_AR */
            0,
            0,
            0
        },
        0 /* ITAG_CFGR */
    },
    /**********
     * IDATA  *
     **********/
    {
        0x2, /* IDATA_OPTIONS */
        131, /* IDATA_MCR */
        4095, /* IDATA_HADDR*/
        0, /* IDATA_LADDR */
        12, /* IDATA_ADDRW */
        0, /* IDATA_CCW */
        2, /* IDATA_BANKS */
        1, /* IDATA_BANKSW */
        38, /* IDATA_VALID_BITS */
        12, /* IDATA_ADDR_PROTECTION */
        /* dm_ecc array */
…
…
    }
};


/*******************************************************************************/
/* PMC100 parameters structure                                                 */
/*******************************************************************************/
Pmc100Params_type yamin<UNIQUE>_params_pmc100 = {
    0xE0046000, /* YAMIN PMC100 base register address */
    0x0, /* REVAND */
    0x0, /* REVISION */
    10, /* DEVID_MBWIDTH */
    5, /* DEVID_MERWIDTH */
    5, /* DEVID_MARWIDTH */
…
…
    0x1f, /* MER_RW_MASK */
    0x80ff0000, /* LCR_RW_MASK */
    0x80ff0000, /* LSCR_RW_MASK */
    0xffff0000 /* TCCR_RW_MASK */
};
```

F.4 PMC-100 software library function parameters

The followng table describes the PMC-100 software library `suspend` and `dont_save_restore` function parameters.

Table F-3 Software library function parameters

Parameter	Description
`suspend_tccr`, `suspend_tc`	Most of the library functions have a `suspend_tccr` and `suspend_tc` parameter passed to them. These parameters can each enable the suspend mode for a test. PMC-100 suspend mode, causes PMC-100 to automatically suspend execution part way through a test and resume later. The resume event can either be triggered by the PMC100_TCCR counter or by the external TC input signal. This feature allows a test to be broken down in to a series of short busts that can each be as small as 20 clock cycles and the gap between bursts is either controlled by the system or application SW: • `suspend_tccr` enables suspend mode using the PMC100_TCCR.TCC counter. When `suspend_tccr` is non-zero, its value is loaded into PMC100_TCCR.TCCI field, which initializes the PMC100_TCCR.TCC counter. • `suspend_tc` enables suspend mode using the TC input signal. • Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If this occurs, then the function will immediately return with the -1 value. • If both `suspend_tccr` and `suspend_tc` are zero PMC-100 execution will not be suspended, and the test will be executed until it completes or fails. In this case the function poles for the test to end or fail. • If either `suspend_tccr` and `suspend_tc` is non-zero, PMC-100 execution will suspend after one or more loops of the test algorithm, depending on the `loops_before_suspension` parameter value. The functions will immediately return 0 if only one of the suspend parameters is non-zero, which must be ignored. Interrupt handlers must be used in this case to process the test end and test fail interrupts generated by PMC-100. • When `suspend_tccr` is non-zero, PMC-100 execution will be suspended for the following number of clock cycles: `suspend_tccr - the number of cycles it takes to execute the algorithm before it suspends` The processor may be stalled while PMC-100 is executing a test algorithm and so it is recommended that the `suspend_tccr` value is chosen to be at least 100 times larger than the number of cycles it takes to execute the algorithm before it suspends. A similar recommendation is made for the gap between TC pulses when `suspend_tc` is non-zero.
`dont_save_restore`	This parameter controls the memory save and restore behaviour of an algorithm. The `dont_save_restore` parameter may be set to 0 or 1 and the meaning of each value is as follows: **0** In most cases, the `dont_save_restore` parameter must be set to 0. This causes the algorithm to save and restore the memory contents during testing. The function does not corrupt the memory. **1** You can set the `dont_save_restore` parameter to 1. This prevents the algorithm from saving and restoring the memory contents during testing. The function corrupts the memory. This saves up to six operations in each loop of a test algorithm. Therefore, it runs significantly faster. This mode can be used in cases where you do not need to preserve the memory contents. For example, when testing a cache, if its contents have been flushed to main memory, then it will be invalidated after testing is complete. **Note** You must disable the memory in this case and if it is a cache then lookups must also be disabled. If this is not possible in a particular IP core, then `dont_save_restore` set to 1 cannot be used with the memory under test.

F.5 PMC-100 software library functions

This section describes the PMC-100 software library functions.

F.5.1 PMC100_SW_Version

This function returns a 24-bit value for the PMC-100 library version.

Parameters

`void`

Return value

<24-bit version value>
major: bits[23:16], minor1: bits[15:8], minor2: bits[7:0]

Syntax

```
int PMC100_SW_Version(void)
```

F.5.2 PMC100_Check_Device_ID

This function checks that the PMC-100 CoreSight ID registers match the expected values. This is used to check that the test is communicating with the expected PMC version.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

Return values

1
Pass

0
Fail

Syntax

```
int PMC100_Check_Device_ID(Pmc100Context_type *ctx)
```

F.5.3 PMC100_PMC_Selftest

This function executes a basic PMC-100 self-test.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

`Pmc100MemInfo_type *mem`
Pointer to the structure that stores all the information for a particular memory type

`uint32_t ecc_ar_idx`

Index in mem->ecc_ar[] of ECC unit to test

`uint32_t dont_save_restore`

1 - Don’t save and restore SRAM contents during test

0 - Save and restore SRAM contents during test

Return values

1
Pass

0
Fail

Syntax

```
int PMC100_PMC_Selftest(Pmc100Context_type *ctx,
                        Pmc100MemInfo_type *mem,
                        utin32_t ecc_ar_idx,
                        uint32_t dont_save_restore)
```

F.5.4 PMC100_Address_LatentFault

Programs PMC-100 to carryout the Address Latent Fault ECC logic test algorithm.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

`Pmc100MemInfo_type *mem`
Pointer to the structure which stores all the information for a particular memory type

`uint32_t suspend_tccr`
Enables suspend mode using the TCCR register value and used as the TCCR.TCCI value

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t suspend_tc`
Enables suspend mode using the TC input signal

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t loops_before_suspension`

Number of loops to perform before suspending - 1

`uint32_t double_error`

0	For single bit error
1	For double bit error
2	For double bit error shifted by 1
3	For double bit error shifted by 2
4	For double bit error shifted by 3

`uint32_t ecc_ar_idx`

Index in mem->ecc_ar[] of ECC unit to test

`uint32_t dont_save_restore`

1 Do not save and restore SRAM contents during test

0 Save and restore SRAM contents during test

`uint32_t bank_start`
The bank start number
The value range is 0 to `mem.banks_number-1`.

`uint32_t bank_end`
The bank end number
The value must be less than or equal to `bank_start`.
The value range is 0 to `mem.banks_number-1`.

Return values

1
Pass

0
Fail

-1
Invalid parameter value

Syntax

```
int PMC100_Address_LatentFault(Pmc100Context_type *ctx,
                               Pmc100MemInfo_type *mem,
                               uint32_t suspend_tccr,
                               uint32_t suspend_tc,
                               uint32_t loops_before_suspension,
                               uint32_t double_error,
                               uint32_t ecc_ar_idx,
                               uint32_t dont_save_restore,
                               uint32_t bank_start,
                               uint32_t bank_end)
```

F.5.5 PMC100_Address_SinglePointFault

This function programs PMC-100 to carry out the Address Single Point Fault ECC logic test algorithm.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

`Pmc100MemInfo_type *mem`
Pointer to the structure which stores all the information for a particular memory type

`uint32_t suspend_tccr`
Enables suspend mode using the TCCR register value and used as the TCCR.TCCI value

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t suspend_tc`

Enables suspend mode using the TC input signal

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t loops_before_suspension`

Number of loops to perform before suspending - 1

`uint32_t double_error`

0	For single bit error
1	For double bit error
2	For double bit error shifted by 1
3	For double bit error shifted by 2
4	For double bit error shifted by 3

`uint32_t ecc_ar_idx`

Index in mem->ecc_ar[] of ECC unit to test

`uint32_t dont_save_restore`

1	Do not save and restore SRAM contents during test
0	Save and restore SRAM contents during test

`uint32_t bank_start`

The bank start number
The value range is 0 to `mem.banks_number-1`.

`uint32_t bank_end`

The bank end number
The value must be less than or equal to `bank_start`.
The value range is 0 to `mem.banks_number-1`.

Return values

1

Pass

0

Fail

-1

Invalid parameter value

Syntax

```
int PMC100_Address_SinglePointFault(Pmc100Context_type *ctx,
                Pmc100MemInfo_type *mem,
                uint32_t suspend_tccr,
                uint32_t suspend_tc,
                uint32_t loops_before_suspension,
                uint32_t double_error,
                uint32_t ecc_ar_idx,
                uint32_t dont_save_restore,
                uint32_t bank_start,
                uint32_t bank_end)
```

F.5.6 PMC100_Data_LatentFault

This fucntion programs PMC-100 to carry out the Data Latent Fault ECC logic test algorithm.

Parameters

`Pmc100Context_type *ctx`

Pointer to the library context

`Pmc100MemInfo_type *mem`

Pointer to the structure which stores all the information for a particular memory type

`uint32_t suspend_tccr`

Enables suspend mode using the TCCR register value and used as the TCCR.TCCI value

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t suspend_tc`

Enables suspend mode using the TC input signal

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t loops_before_suspension`

Number of loops to perform before suspending - 1

`uint32_t double_error`

0 For single bit error
1 For double bit error
2 For double bit error shifted by 1
3 For double bit error shifted by 2
4 For double bit error shifted by 3

`uint32_t ecc_ar_idx`

Index in mem->ecc_ar[] of ECC unit to test

`uint32_t dont_save_restore`

1 Do not save and restore SRAM contents during test
0 Save and restore SRAM contents during test

`uint32_t bank_start`

The bank start number
The value range is 0 to `mem.banks_number-1`.

`uint32_t bank_end`

The bank end number
The value must be less than or equal to `bank_start`.
The value range is 0 to `mem.banks_number-1`.

Return values

1
Pass

0
Fail

-1
Invalid parameter value

Syntax

```
int PMC100_Data_LatentFault(Pmc100Context_type *ctx,
                Pmc100MemInfo_type *mem,
                uint32_t suspend_tccr,
                uint32_t suspend_tc,
                uint32_t loops_before_suspension,
                uint32_t double_error,
                uint32_t ecc_ar_idx,
                uint32_t dont_save_restore,
                uint32_t bank_start,
                uint32_t bank_end)
```

F.5.7 PMC100_Data_SinglePointFault

This function programs PMC-100 to carry out the Data Single Point Fault ECC logic test algorithm.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

`Pmc100MemInfo_type *mem`
Pointer to the structure which stores all the information for a particular memory type

`uint32_t suspend_tccr`
Enables suspend mode using the TCCR register value and used as the TCCR.TCCI value

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t suspend_tc`
Enables suspend mode using the TC input signal

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t loops_before_suspension`

Number of loops to perform before suspending - 1

`uint32_t double_error`

0	For single bit error
1	For double bit error
2	For double bit error shifted by 1

3 For double bit error shifted by 2

4 For double bit error shifted by 3

`uint32_t ecc_ar_idx`

Index in mem->ecc_ar[] of ECC unit to test

`uint32_t dont_save_restore`

1 Do not save and restore SRAM contents during test

0 Save and restore SRAM contents during test

`uint32_t bank_start`

The bank start number
The value range is 0 to `mem.banks_number-1`.

`uint32_t bank_end`

The bank end number
The value must be less than or equal to `bank_start`.
The value range is 0 to `mem.banks_number-1`.

Return values

1

Pass

0

Fail

-1

Invalid parameter

Syntax

```
int PMC100_Data_SinglePointFault(Pmc100Context_type *ctx,
                Pmc100MemInfo_type *mem,
                uint32_t suspend_tccr,
                uint32_t suspend_tc,
                uint32_t loops_before_suspension,
                uint32_t double_error,
                uint32_t ecc_ar_idx,
                uint32_t dont_save_restore,
                uint32_t bank_start,
                uint32_t bank_end)
```

F.5.8 PMC100_Set_Reg2zero

This function sets all the writable PMC-100 registers to zero.

Parameters

`Pmc100Context_type *ctx`

Pointer to the library context

Return values

1

Pass

0

Fail

Syntax

```
int PMC100_Set_Reg2zero(Pmc100Context_type *ctx)
```

F.5.9 PMC100_Error_Injection

This function programs PMC-100 to inject an error into a memory location.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

`Pmc100MemInfo_type *mem`
Pointer to the structure which stores all the information for a particular memory type

`uint32_t address`

Address of location to inject an error

The range is `mem.laddr` to `mem.haddr`.

`uint32_t err_mask[]`

Mask that indicates which bits to inject an error

Return values

1
Pass

0
Fail

-1
Invalid parameter

Syntax

```
int PMC100_Error_Injection(Pmc100Context_type *ctx,
                Pmc100MemInfo_type *mem,
                uint32_t  address,
                uint32_t  err_mask[])
```

F.5.10 PMC100_Write_Read_Allregs

This function reads and writes to all the PMC-100 registers. It is intended for use in the PMC-100 self-test.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

Return values

1
Pass

0
Fail

Syntax

```
int PMC100_Write_Read_Allregs(Pmc100Context_type *ctx)
```

F.5.11 PMC100_PMC_CheckTestResult

This function checks that the Test Fail Flag is not set. If an error occurs, this function prints information about the error.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

Return values

1
Pass

0
Fail

Syntax

```
int PMC100_CheckTestResult(Pmc100Context_type *ctx)
```

F.5.12 PMC100_ShortBurst_1port

This function programs PMC-100 to carry out the Short Burst software transparent single-ported SRAM test.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

`Pmc100MemInfo_type *mem`
Pointer to the structure which stores all the information for a particular memory type

`uint32_t suspend_tccr`
Enables suspend mode using the TCCR register value and used as the TCCR.TCCI value

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t suspend_tc`
Enables suspend mode using the TC input signal

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t loops_before_suspension`

Number of loops to perform before suspending - 1

`uint32_t dont_save_restore`

1 Do not save and restore SRAM contents during test
0 Save and restore SRAM contents during test

`uint32_t addrcd`

Address change direction

PMC100_CTRL.ADDRCD value

Affects the next PMC100_RADDR and PMC100_CADDR address register values as follows:

0

PMC100_CADDR is changed first.

All PMC100_CADDR values are accessed before the PMC100_RADDR is changed.

1

PMC100_RADDR is changed first.

All PMC100_RADDR values are accessed before the PMC100_CADDR is changed.

`uint32_t addr_end`

End address of SRAM region tested

The range is `mem.laddr` to `mem.haddr`.

`uint32_t addr_start`

Start address of SRAM region tested

The value must be less or equal to `addr_end`.

The range is `mem.laddr` to `mem.haddr`.

Return values

1

Pass

0

Fail

-1

Invalid parameter value

Syntax

```
int PMC100_ShortBurst_1port(Pmc100Context_type *ctx,
                            Pmc100MemInfo_type *mem,
                            uint32_t suspend_tccr,
                            uint32_t suspend_tc,
                            uint32_t loops_before_suspension,
                            uint32_t dont_save_restore,
                            uint32_t addrcd,
                            uint32_t addr_end,
                            uint32_t addr_start)
```

F.5.13 PMC100_ShortBurst_2ports

This function programs PMC-100 to carry out the Short Burst software transparent two-ported SRAM test.

Parameters

`Pmc100Context_type *ctx`

Pointer to the library context

`Pmc100MemInfo_type *mem`

Pointer to the structure which stores all the information for a particular memory type

`uint32_t suspend_tccr`

Enables suspend mode using the TCCR register value and used as the TCCR.TCCI value

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t suspend_tc`

Enables suspend mode using the TC input signal

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t loops_before_suspension`

Number of loops to perform before suspending - 1

`uint32_t dont_save_restore`

1 Do not save and restore SRAM contents during test

0 Save and restore SRAM contents during test

`uint32_t addrcd`

Address change direction

PMC100_CTRL.ADDRCD value

Affects the next PMC100_RADDR and PMC100_CADDR address register values as follows:

0

PMC100_CADDR is changed first.

All PMC100_CADDR values are accessed before the PMC100_RADDR is changed.

1

PMC100_RADDR is changed first.

All PMC100_RADDR values are accessed before the PMC100_CADDR is changed.

`uint32_t addr_end`

End address of SRAM region tested

The range is `mem.laddr` to `mem.haddr`.

`uint32_t addr_start`

Start address of SRAM region tested

The value must be less or equal to `addr_end`.

The range is `mem.laddr` to `mem.haddr`.

Return values

1

Pass

0
Fail

-1
Invalid parameter value

Syntax

```
int PMC100_ShortBurst_2ports(Pmc100Context_type *ctx,
                             Pmc100MemInfo_type *mem,
                             uint32_t suspend_tccr,
                             uint32_t suspend_tc,
                             uint32_t loops_before_suspension,
                             uint32_t dont_save_restore,
                             uint32_t addrcd,
                             uint32_t addr_end,
                             uint32_t addr_start)
```

F.5.14 PMC100_MarchCm

This function programs PMC-100 to carry out the March C- production MBIST SRAM test.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

`Pmc100MemInfo_type *mem`
Pointer to the structure which stores all the information for a particular memory type

`uint32_t suspend_tccr`
Enables suspend mode using the TCCR register value and used as the TCCR.TCCI value

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t suspend_tc`
Enables suspend mode using the TC input signal

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t loops_before_suspension`

Number of loops to perform before suspending - 1

`uint32_t dont_save_restore`

1 Do not save and restore SRAM contents during test
0 Save and restore SRAM contents during test

`uint32_t addrcd`

Address change direction

PMC100_CTRL.ADDRCD value

Affects the next PMC100_RADDR and PMC100_CADDR address register values as follows:

0

PMC100_CADDR is changed first.

All PMC100_CADDR values are accessed before the PMC100_RADDR is changed.

1

PMC100_RADDR is changed first.

All PMC100_RADDR values are accessed before the PMC100_CADDR is changed.

`uint32_t addr_end`

End address of SRAM region tested

The range is `mem.laddr` to `mem.haddr`.

`uint32_t addr_start`

Start address of SRAM region tested

The value must be less or equal to `addr_end`.

The range is `mem.laddr` to `mem.haddr`.

Return values

1

Pass

0

Fail

-1

Invalid parameter value

Syntax

```
int PMC100_MarchCm(Pmc100Context_type *ctx,
                Pmc100MemInfo_type *mem,
                uint32_t suspend_tccr,
                uint32_t suspend_tc,
                uint32_t loops_before_suspension,
                uint32_t dont_save_restore,
                uint32_t addrcd,
                uint32_t addr_end,
                uint32_t addr_start)
```

F.5.15 PMC100_Raw_Mem_Dump

This function programs PMC-100 to dump the contents of a memory. Read data from the full width of the MBIST data bus, including ECC fields, is copied to memory buffer `*p`. For details of the MBIST read data format for each memory, see the integration documentation for your IP core. The contents of the buffer can then be read by a debugger. Alternatively, the contents of an embedded memory can be dumped directly by a debugger without needing to copy to an intermediate memory. In this case, the function will have to be copied and modified to add debugger access mechanisms to PMC-100 registers.

Parameters

`Pmc100Context_type *ctx`

Pointer to the library context

`Pmc100MemInfo_type *mem`

Pointer to the structure which stores all the information for a particular memory type

`uint32_t *p`

Pointer to the buffer to write the content of memory

`uint32_t len`

Length of buffer in words = `(end_addr+1- start_addr) * NUM_OF_DATA_WORDS`

`uint32_t addr_end`

End address of SRAM region tested

The range is `mem.laddr` to `mem.haddr`.

`uint32_t addr_start`

Start address of SRAM region tested

The value must be less or equal to `addr_end`.

The range is `mem.laddr` to `mem.haddr`.

Return values

1

Pass

0

Fail

-1

Invalid parameter value

Syntax

```
int PMC100_Raw_Mem_Dump(Pmc100Context_type *ctx,
                Pmc100MemInfo_type *mem,
                uint32_t *p,
                uint32_t len,
                uint32_t addr_end,
                uint32_t addr_start)
```

F.5.16 PMC100_Memory_Init

Programs PMC-100 to initialize the contents of a memory 0. Writes are performed through the ECC generation logic and the ECC fields are also initialized correctly.

Parameters

`Pmc100Context_type *ctx`

Pointer to the library context

`Pmc100MemInfo_type *mem`

Pointer to the structure which stores all the information for a particular memory type

`uint32_t suspend_tccr`

Enables suspend mode using the TCCR register value and used as the TCCR.TCCI value

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t suspend_tc`

Enables suspend mode using the TC input signal

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t loops_before_suspension`

Number of loops to perform before suspending - 1

`uint32_t addr_end`

End address of SRAM region tested

The range is `mem.laddr` to `mem.haddr`.

`uint32_t addr_start`

Start address of SRAM region tested

The value must be less or equal to `addr_end`.

The range is `mem.laddr` to `mem.haddr`.

Return values

1

Pass

0

Fail

-1

Invalid parameter value

Syntax

```
int PMC100_Memory_Init(Pmc100Context_type *ctx,
                Pmc100MemInfo_type *mem,
                uint32_t suspend,
                uint32_t loops_before_suspension,
                uint32_t addr_end,
                uint32_t addr_start)
```

F.5.17 PMC100_Mem_Scrub

This function programs PMC-100 to carry out the memory scrubbing algorithm.

Parameters

`Pmc100Context_type *ctx`

Pointer to the library context

`Pmc100MemInfo_type *mem`

Pointer to the structure which stores all the information for a particular memory type

`uint32_t suspend_tccr`

Enables suspend mode using the TCCR register value and used as the TCCR.TCCI value

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t suspend_tc`

Enables suspend mode using the TC input signal

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t loops_before_suspension`

Number of loops to perform before suspending - 1

`uint32_t addr_end`

End address of SRAM region tested

The range is `mem.laddr` to `mem.haddr`.

`uint32_t addr_start`

Start address of SRAM region tested

The value must be less or equal to `addr_end`.

The range is `mem.laddr` to `mem.haddr`.

Return values

1

Pass

0

Fail

-1

Invalid parameter value

Syntax

```
int PMC100_Mem_Scrub(Pmc100Context_type *ctx,
                Pmc100MemInfo_type *mem,
                uint32_t suspend,
                uint32_t loops_before_suspension,
                uint32_t addr_end,
                uint32_t addr_start)
```

F.5.18 PMC100_Mem_Fatal_Scrub

This function programs PMC-100 to carry out the memory scrubbing algorithm with un-correctable error check. At the beginning of each loop, the algorithm checks if the current memory location contains an uncorrectable error and if it does, the test fails.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

`Pmc100MemInfo_type *mem`
Pointer to the structure which stores all the information for a particular memory type

`uint32_t suspend_tccr`
Enables suspend mode using the TCCR register value and used as the TCCR.TCCI value

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t suspend_tc`
Enables suspend mode using the TC input signal

Note

Both `suspend_tccr` and `suspend_tc` must not be non-zero at the same time. If both parameters are zero, suspend mode is disabled and the function uses poling to detect test end or test fail. If either parameter is non-zero, then suspend mode is enabled and interrupt will indicate whether the test ends or fails. The return value should be ignored in this case.

`uint32_t loops_before_suspension`

Number of loops to perform before suspending - 1

`uint32_t addr_end`

End address of SRAM region tested

The range is `mem.laddr` to `mem.haddr`.

`uint32_t addr_start`

Start address of SRAM region tested

The value must be less or equal to `addr_end`.

The range is `mem.laddr` to `mem.haddr`.

Return values

1
Pass

0
Fail

-1
Invalid parameter value

Syntax

```
int PMC100_Mem_Fatal_Scrub(Pmc100Context_type *ctx,
                Pmc100MemInfo_type *mem,
                uint32_t suspend,
                uint32_t loops_before_suspension,
                uint32_t addr_end,
                uint32_t addr_start)
```

F.5.19 PMC100_MemError_Reporting_Bus

This function programs PMC-100 to inject an error into a memory location and read it back through the ECC checking logic. The **MBISOLPREN** signal is set during the read that enabled the error to be reported on the IP core error reporting bus. The original value of the memory location is saved and restored.

Parameters

`Pmc100Context_type *ctx`
Pointer to the library context

`Pmc100MemInfo_type *mem`
Pointer to the structure which stores all the information for a particular memory type

`uint32_t double_error`

0	For single bit error
1	For double bit error

`uint32_t address`

Address of location to inject an error

Return values

1
Pass

0
Fail

-1
Invalid parameter value

Syntax

```
int PMC100_MemError_Reporting_Bus(Pmc100Context_type *ctx,
                Pmc100MemInfo_type *mem,
                uint32_t  double_error
                uint32_t  address )
```

Appendix G
Revisions

This appendix describes the technical changes between released issues of this book.

It contains the following section:

G.1 Revisions

The following table shows the technical changes between released issues of this book.

Table G-1 Issue 0000-01

Change	Location	Affected
First release for r0p0	-	-