Freescale Semiconductor
Application Note

Document Number: AN3532
Rev. 0, 11/2007

Error Correction and Error Handling on PowerQUICC™ III Processors

by *Paul Genua*
NCSG
Freescale Semiconductor, Inc.
Austin, TX

In an effort to enable the design of highly reliable systems, the PowerQUICC™ III family provides error correction and checking mechanism on internal memories as well as the external DDR SDRAM data bus. This application note provides an introduction to the error correcting code (ECC) technology as well as an aid to initialization and error recovery on Freescale's PowerQUICC III family of processors.

Contents

1 Introduction

As embedded memory density increases and memory cell voltage decreases on microprocessors, it becomes possible that the state of a memory cell is subject to change by soft errors, such as changes in a memory state due to external factors like package decay, external system noise, and cosmic radiation.

The common sources of soft errors are low-energy alpha particles, high-energy cosmic particles, and thermal neutrons. From these common sources, neutrons are the most troublesome because they can penetrate most of the manmade material. The soft errors are random and not repeatable, and do not cause permanent damage to a device.

freescale semiconductor

To overcome the problems from soft errors, memory vendors are introducing new technologies to make the devices less susceptible to the soft errors. In addition, soft errors are being though of at the system-level and board-level design. The PowerQUICC™ III processor includes ECC, which is capable of correcting single-bit errors and detecting two-bit errors. The proper implementation of ECC with attention to susceptibility of soft errors ensures more robust embedded design.

2 Error Correcting Code (ECC) Theory

2.1 Parity

The parity memory helps protect data, but the protection is minimal. The intent of parity memory is to provide single-bit error detection capability. The parity checking works through the generation of a parity bit for the data to be written into memory. In the parity checking scheme, odd or even parity is generated through applying the XOR function to all bits of a data-word. The parity bit is then stored in memory along with the original data-word. When the data-word is read from the memory, the parity bit is re-computed and compared to the retrieved parity bit. If the re-computed parity bit differs from the retrieved parity bit, a parity error is generated and reported to the system.

The parity checking is relatively simple and inexpensive to implement. However, parity checking can only detect odd-bit errors and does not have the ability to correct errors once detected. Due to the speed and ease of implementation, parity checking is still used on L1 cache memories as well as L2 cache tags within the PowerQUICC™ III family of processors.

2.2 Error Correcting Codes

ECC is implemented in a system by adding additional the error correction bits that are known as syndrome bits. These bits are added to the user data bits. The bits are calculated in a systematic way, and contain additional information about the data-word that allows detection of dual bit errors or correction of single-bit errors.

ECC algorithms that are also known as Hamming codes are mathematically determined. All valid single error correction, double bit error detection (SECDED) ECC codewords differ from each other by at least 4 bits (known as a Hamming distance of 4). The distance of 4 bits ensures that a single-bit error and a two-bit error can be distinguished from each other and that any given valid ECC codeword can sustain a one-bit error and can unambiguously recover the original valid ECC codeword. However, two-bit errors cannot be corrected because multiple valid SECDED codewords can result in the same error vector with different two-bit errors.

Figure 1 shows the process of encoding an 8-bit data vector into an SECDED ECC codeword.

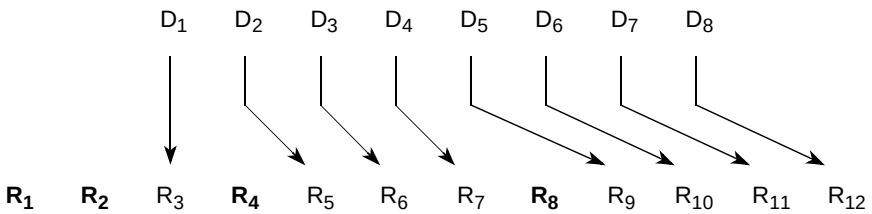

Figure 1. Encoding an 8-bit Data Vector into an SECDED ECC Codeword

In Figure 1, D[1 : 8] represents an 8-bit data-word, and R[1 : 12] is the combined data and ECC codeword. R1, R2, R4, and R8 together are considered as the ECC syndrome bits.

The basic idea is the use of recursive applications of an XOR function in order to generate parity protection via multiple check bits spread out throughout the error check bits. In this example, the encoding phase of the SECDED algorithm is implemented by distributing the input data vector throughout the ECC codeword. The bits in the input data vector are labeled as D_j, where the index j goes from 1 to N. Later, the data vector is distributed into the SECDED ECC word, R_m, where the index m goes from 1 to $N + (\log_2 N) + 1$, and m is not an integer power of 2.

In the bit vector R, all bit positions R_m for which m is a power of two are reserved for the check bits. In this example, each data bit in bit position, R_m, where m is not a power of 2 is protected by multiple check bits. The idea is that the data bit is protected by the check bits for which indices sum up to the index of the data bit. In the example, R_7 is protected by R_1, R_2, and R_4 while R_{10} is protected by R_2 and R_8.

In this example, check bit R_1 protects R_3, R_5, R_7, R_9, and R_{11} through the use of the exclusive-or function. Therefore, R_1 is created by the exclusive-or function for R_3, R_5, R_7, R_9, and R_{11}. If any one bit in the protected set changes value, the stored value of R_1 differs from the re-computation of the check bit, R_1.

Figure 2 shows a method to aid in the generation of check bits.

R_1	=	R_3	⊕	R_5	⊕	R_7	⊕	R_9	⊕	R_{11}
R_{0001}	=	R_{0011}	⊕	R_{0101}	⊕	R_{0111}	⊕	R_{1001}	⊕	R_{1011}
R_2	=	R_3	⊕	R_6	⊕	R_7	⊕	R_{10}	⊕	R_{11}
R_{0010}	=	R_{0011}	⊕	R_{0110}	⊕	R_{0111}	⊕	R_{1010}	⊕	R_{1011}

Figure 2. Method to Aid in the Generation of the Check Bits

If all bit indices in binary format are rewritten, and the check bit, R_1 (R_{0001} in binary) along with its protected set of R_3, R_5, R_7, R_9 is examined, it is observed that both the check bit and all protected bits contain a '1' in the least significant bit positions (bit 0) of their respective indices. This observation holds true for all check bits and can aid in generating the remaining bits. R_2 (R_{0010} in binary) contains a '1' in the bit 1 position of its index. R_2 can therefore be generated using the exclusive-or operation on all the data bits R_k with a '1' in bit position 1 of their respective indices.

R_0 is used as a parity check for the entire input data vector. The purpose of R_0 is to provide a quick sanity check to ensure that the error that had occurred was not a double-bit error. In the case of a single-bit error, both the ECC syndrome as well as R_0 reports that an error had occurred. In the case of a two-bit error, R_0 returns the same parity as the original codeword, whereas the ECC syndrome bits return a bit position that is non-zero. The difference between these redundant error reporting mechanisms enables to distinguish between a single-bit error and a two-bit error. In the examples, even parity scheme is used for the generation of R_0, and odd parity scheme can also be implemented.

Figure 3 shows the complete encoding process for an SECDED codeword, which includes the original 8-bits of data interspersed with the syndrome bits.

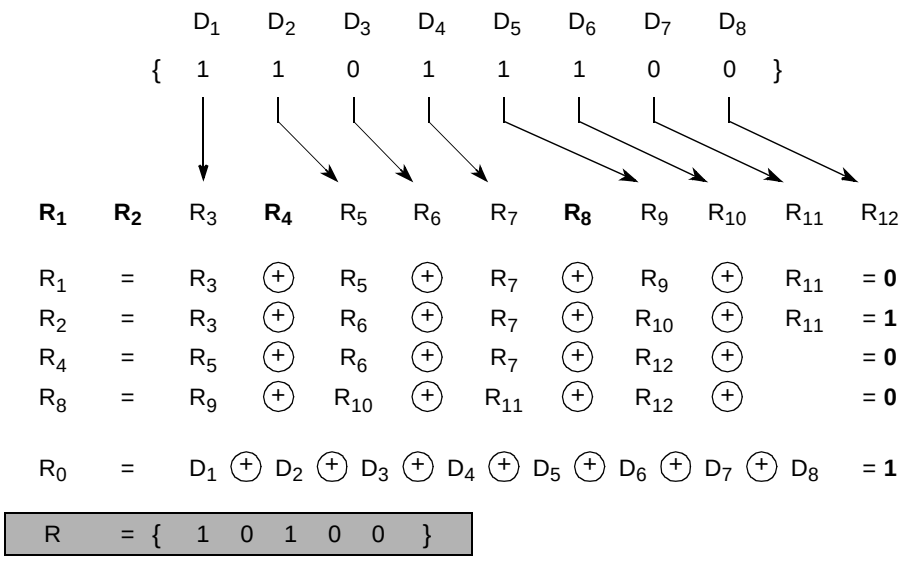

Figure 3. Encoding Process for an SECDED Codeword

The following steps describe the encoding process for an SECDED codeword:

1. Distribute the data vector into the ECC word.
2. Calculate the individual check bits.
3. Calculate R_0 as a parity check over the original data vector, resulting in an eventual ECC word of R={ 1 0 1 0 0 }. For example data vector of D = { 1 1 0 1 1 1 0 0 }.

Figure 4 shows the verification process for an example codeword that includes a single-bit error at index R_7.

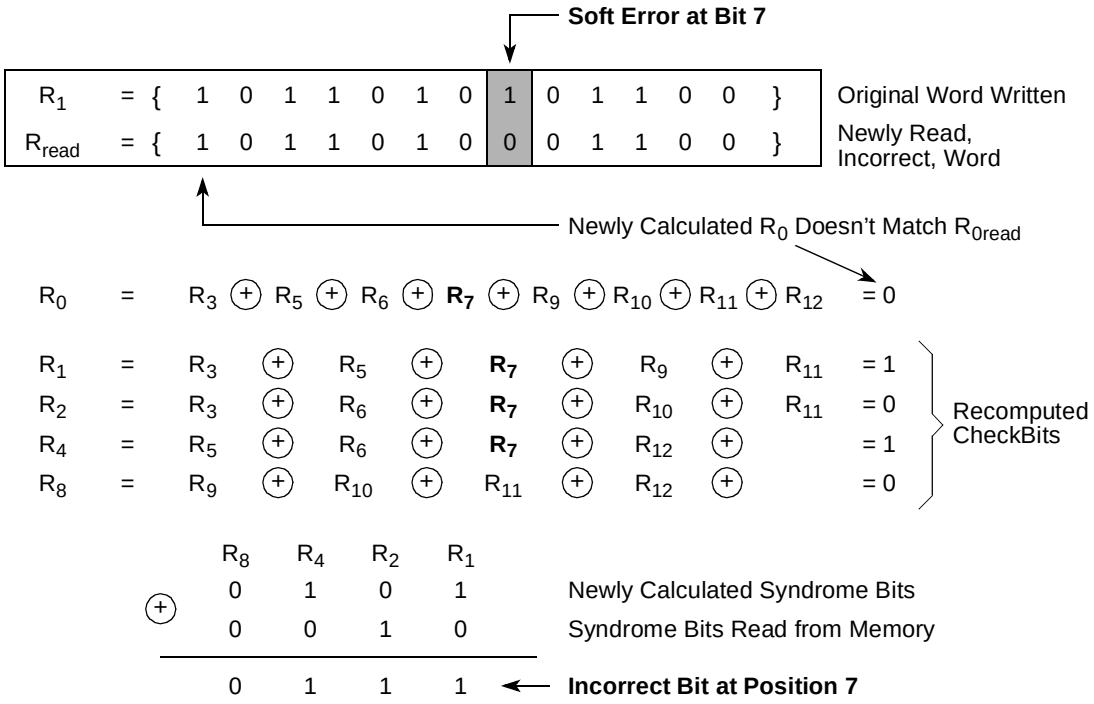

Figure 4. Verification Process for an Example Codeword

For the purposes of this example, an assumption is made that a soft error caused R_7 to flip from a '1' to a '0'. After recalculating R_0, it is observed that the result does not match the R_0 read into the memory controller. On further recalculation of the check bits R_1, R_2, R_4, and R_8, it is observed that three of the four check bits have changed. As both check bits and parity check, R_0, have changed, it is concluded that it is a single-bit error. If check bits are different but the parity bit remained same, it can be concluded that a two-bit error has occurred.

Finally, if the newly computed check bits are compared with the check bits read through an exclusive OR function, the exact bit position of the offending bit can be found.

The strength of SECDEC ECC is that is it able to detect and correct single-bit errors with minimal overhead in additional memory. The data bus (width = 64 bits, N = 64) of PowerQUICC™ III processor, the SECDED ECC algorithm requires 8 check bits that is the same amount of additional bits as using an 8 : 1 parity checking scheme when applied to a 64 bit wide data vector. The delay logic of SECDED is proportional to $\log_2 N$, and in the case of the PowerQUICC III processor, an additional cycle of read latency is incurred.

3 Error Detection and Handling on Internal Memories

To address the issue of soft errors, Freescale provides multiple levels of error detection and correction on the PowerQUICC™ III family of processors. The smaller L1 caches, L2 address tag arrays, and local bus

are parity protected. The L2 cache data array as well as the external DDR memory bus is SECDED ECC protected. In the following sections, each form of error detection and correction is addressed in details.

The chances of a soft error in the L1 caches or L2 tag array are small (less than 1 % probability per year). With parity checking enabled, this rare error can be detected in the parity-protected memory when the erroneous cache line is accessed, triggering a machine check exception. Depending on the operating system, the machine check handler responds by halting the application, alerting the kernel, or even resetting the hardware. To avoid the need for such a catastrophic system fault or interruption in service, a detected parity error can be managed by intelligent parity detection handlers that can also be useful in handling a higher soft error rate caused by a higher amount of cosmic radiation.

Although soft errors cannot be eliminated, the chances that an error causes a system fault can be reduced. A soft error detected by parity can be mitigated by invalidating the erroneous cache line and refetching the correct instruction or data from memory. The memory areas considered are as follows:

- L1 instruction cache parity error handling
- L1 data cache parity error handling
- L2 tag parity error or multi-bit ECC error handling
- L2 data cache

Invalidating and refetching generally works for reads. However, writes to the L1 data cache or L2 cache that result in modified data and a parity error are problematic and may cause a system fault that must be handled by the operating system or even a system reboot to avoid data corruption.

3.1 L1 Instruction Cache Parity

The e500 core supports L1 instruction cache parity. As the L1 instruction cache does not contain modified data, an instruction cache parity error can be recoverable. Parity generation is always performed, but parity checking can be enabled or disabled using the L1CSR1[ICPE] bit. By default, this bit is disabled at power-up. If ICPE is set, parity is checked per instruction read. There is a single parity bit per 4-byte instruction. If a parity error is detected, the device generates a machine check interrupt. If MSR[ME] is set, the machine check interrupt handler is invoked, and if MSR[ME] is cleared, the device enters the checkstop state.

If both MSR[ME] and L1CSR1[ICPE] are set, the detection of the L1 instruction cache parity error generates a machine check interrupt and the device places the physical address of the erroneous instruction in the MCSRR0 register. The recovery mechanism for this error is to invalidate the cache line of the erroneous instruction in the L1 instruction cache and resume operation by executing the **rfmci** instruction. However, the recovery code must be guaranteed not to generate an L1 cache parity error. When a machine check interrupt is taken, the device clears the MSR[ME] bit. In this state, any subsequent machine check interrupts cause the device to enter the checkstop state. Therefore, if the machine check handler recovery code generates an L1 cache parity error, the device enters the checkstop state.

To recover from the L1 instruction cache parity error (ICPERR) the following requirements must be met:

- The machine check parity recovery code and stack are in a cache-inhibited region
- The cache-inhibited region is guaranteed not to take an exception for DSO, ISI, Alignment, or a TLB miss. One of the sixteen Level 2 TLB1 entries is dedicated for this space.

When a machine check exception occurs, perform the following steps:

1. Save a general purpose register (GPR) to a software use special purpose register (SPRG) dedicated to the machine check handler.
2. In the GPR, load a pointer to the machine check stack in a cache inhibited region.
3. Save any necessary scratch GPRs to the stack.

Now, perform the following steps using only the scratch GPRs:

1. If MCSRR0[0 : 15] != IVPR[0 : 15], go to step 2. Otherwise, go to step 6.
2. Execute the **icbi** instruction to the address in MCSRR0.
3. Restore the scratch GPRs from the stack.
4. Restore the GPR from the dedicated machine and check the SPRG.
5. Execute the **rfmci** instruction to resume operation.
6. For each IVORn, if MCSRR0[16 : 27] == IVORn[16 : 27] go to step 7; otherwise go to step 8.
7. This is a CPU30 errata error condition. Perform the following steps to attempt a recover:
   a) Execute the **icbi** instruction to an address in SRR0.
   b) Restore the scratch GPRs from the stack.
   c) Restore the GPR from the dedicated machine and check the SPRG.
   d) Execute the **rfi** instruction.

   NOTE

   If the registers are corrupted because of a CPU30 error, the recovery method explained in step 7 is not applicable. For more information, see Section 3.1.1, “CPU30 Erratum.”
8. If this is not a CPU30 errata error condition, go to step 2.

3.1.1 CPU30 Erratum

On the e500 core (revision 1.0 and revision 2.0), the L1 instruction cache parity errors can be reported incorrectly when an instruction that is corrupted by a parity error appears to cause an interrupt or to write the SPEFSCR. This erratum not only can report an incorrect value in MSRR0, but also can affect other architectural registers, such as the SPEFSCR and ESR. When this problem occurs, MSRR0 points to the first instruction of an interrupt handler (instead of pointing to the instruction that caused the interrupt) and SRR0 points to the corrupted instruction. To correct this problem, perform the following steps:

1. Place the first instruction of all the exception handlers in a cache-inhibited region.
2. When an instruction cache parity error is detected (MCSR[ICPERR] is set when a machine check exception occurs), check to see whether the MCSRR0 is pointing to the first instruction of any exception handler.
3. If the MCSRR0 is not pointing to the first instruction of an exception handler, invalidate the instruction cache entry to which the MCSRR0 is pointing and execute the **rfmci** instruction.
4. If the MCSRR0 is pointing to the first instruction of an exception handler and SRR0 is pointing to the application code, terminate the application.

5. If the MCSRR0 is pointing to the first instruction of an exception handler and SRR0 is pointing to the kernel code, initiate a reset sequence.

The ICPERR recovery workaround discussed in this application note assumes that an unrecoverable CPU30 error is highly unlikely. As the erratum states, architectural registers can be updated because of a CPU30 error, making the error unrecoverable. The following sequence must occur for a CPU30 error to be detected and be unrecoverable. Note that each step in this scenario is less likely to occur than its previous step:

1. An ICPERR must be detected. The parity error must be on the 4-byte instruction and not on the parity bit itself.
2. The opcode that is corrupted with a parity error causes an interrupt other than the machine check interrupt due to ICPERR. For example, the resulting opcode can be in illegal instruction that generates a program interrupt.
3. The opcode that is corrupted with a parity error causes an interrupt, and the resulting opcode is one that updates architectural registers that cannot be restored.

Therefore, the recovery for the CPU30 error used in this application note is the simplest approach, which is to invalidate the instruction cache line pointed to by SRR0 and then execute the **rfi** instruction.

3.2 L1 Data Cache Parity

The e500 core supports L1 data cache parity. As the L1 data cache can contain modified data, recovery from a data cache parity error may or may not be possible, depending on whether the data has been modified. Parity generation is always performed on a data store, but parity checking can be enabled or disabled using the L1CSR0[CPE] bit. By default, this bit is disabled at power-up. If CPE is enabled, parity is checked on a per byte basis. There is a single parity bit for every one byte of data. If a parity error is detected on a load, the device generates a machine check interrupt. If MSR[ME] is set, the machine check interrupt handler is invoked; if MSR[ME] is cleared, the device enters the checkstop state.

When both MSR[ME] and L1CSR0[CPE] are set, the detection of an L1 data cache parity error generates a machine check interrupt. If the machine check interrupt is due to a data cache parity error (DCPERR), the address of the failing load instruction is placed in the MCSRR0 register (the MCAR register is not meaningful for this error). If the machine check interrupt is due to a data cache push parity error (DCP_PERR), the address on the cache line with the error is placed in the MCAR register (the MCSRR0 register is not meaningful for this error). The recoverability for this error is determined depending on if the parity error is on modified data. If it is determined that the data was not modified, the recovery mechanism for this error is to invalidate the data cache line and resume operation by executing the **rfmci** instruction. However, the recovery code must be guaranteed not to generate an L1 cache parity error. When a machine check interrupt is taken, the device clears the MSR[ME] bit. In this state, any subsequent machine check interrupts cause the device to enter the checkstop state. Therefore, if the machine check handler recovery code generates an L1cache parity error, the device enters the checkstop state.

Recovery from an L1 data cache parity error (DCPERR) is possible only when the parity error is on a non-modified line. To determine whether a line has been modified, first verify that all the following requirements are met:

- The machine check parity recovery code and stack are in a cache-inhibited region.

- The cache-inhibited region is guaranteed not to take an exception for DSI, ISI, Alignment, and a TLB miss. One of the sixteen Level 2 TLB1 entries is dedicated for this space.
- The machine check handler is re-entrant to a 2-deep level. The handler can take a second machine check while it is in the machine check and save enough state to recover.

When a machine check exception is generated, perform the following steps:

1. Save a GPR to an SPRG dedicated to the machine check handler.
2. In the GPR, load a pointer to the machine check stack in a cache inhibited region.
3. Save any necessary scratch GPRs and the dedicated SPRG to the stack.
4. Save a re-entrant flag to the stack to indicate how deep the handler is nested.
5. Set the MSR[ME] bit to re-enable the machine check interrupt handler.
6. Using only the scratch GPRs, flush the L1 data cache based on the following conditions:
   a) If a DCP_PERR occurs during the flush, the system cannot recover from the loss of data. The machine check handler may halt the application, alert the kernel, or cause a reset of the hardware. If desired, the handler can use the re-entrant flag to determine the registers that are required to restore from the stack to recover the previous machine check state.
   b) If no DCP_PERR occurs, the original DCPERR is for an unmodified line and is invalidated by the flush. Continue to step 7.
7. Restore scratch GPRs from the stack.
8. Restore GPR from dedicated machine and check the SPRG.
9. Execute the **rfmci** instruction to resume operation.

The notion of acceptable levels of data loss versus performance loss is a system application-dependent decision. Implementation of the techniques outlined in the current section is left up to the user. In this application note, the workaround method shown is L1 data cache disabled.

3.2.1 Early Write Back Technique

The amount of modified data in the L1 data cache should be minimized to minimize the potential for unrecoverable loss of data because of an L1 data parity error. The minimization can be done by periodically flushing the contents of the L1 data cache to memory. During each flush, a portion of the L1 data cache can be flushed, for example, one set at a time.

3.2.2 L1 Data Cache Push Parity Error

A cache parity error can be detected when modified data is cast out or pushed back to main memory. It is not possible to recover from a L1 data cache push parity error (DCP_PERR). Only modified L1 data cache lines are pushed because of a snoop or cache line victimization due to reallocation. A DCP_PERR causes a loss of data, which is unrecoverable. In Linux, the user process must be halted or, if it is a kernel process, a kernel alert must be initiated.

3.2.3 Write-Through Mode

L1 data cache push parity errors can be avoided altogether by putting all cacheable data space in write-through mode. In write-through mode, data is never in the modified state in the cache. All data written to the cache is simultaneously written to memory to maintain coherency. If a data cache parity error (DCPERR) occurs, it is always recoverable by simply invalidating the cache line. Write-through mode is enabled on a per page basis through TLBs. If a system moves from write-back mode to write-through mode, system software must account for the fact that the **lwarx**, **stwcx**, **dcba**, and **dcbz** instructions generate exceptions if they are executed to a write-through space.

3.3 L2 Cache Errors

The L2 cache provides ECC protection on its data array and parity protection on its tag array. The ECC protection automatically detects and corrects single-bit errors. The single-bit error threshold feature should be used to detect when a threshold of single-bit errors is reached. When the threshold is reached, software should invalidate the L2 tags to clear out all single-bit errors. This section discusses the recovery from L2 tag parity errors and L2 multi-bit ECC errors that can be signaled in two ways:

- With the machine check interrupt by setting MSR[ME] = 1 and HID1[RFXE] = 1
- With MPIC

The better practice is to signal both errors using one of the two methods, but not both. In this application note, the machine check interrupt is required. When L2 returns a cache line for a hit and an L2 error is detected, the L2 also asserts the core_fault_in signal to the e500 core. If HID1[RFXE] = 1, the core responds either by generating a machine check if MSR[ME] = 1 or by entering the checkstop state if MSR[ME] = 0.

3.3.1 L2 Tag Parity Error

The L2 tag parity protection is guaranteed to detect an odd number of bit flips in the L2 tag array. An L2 tag parity error can be detected only on a valid L2 tag. An L2 tag parity error is signaled only when there is an L2 hit into a valid way of a set that also has another valid way with a tag parity error.

3.3.2 L2 Multi-Bit ECC

The L2 ECC protection is guaranteed to detect a double-bit error in the L2 data array, although double-bit errors are unlikely to occur. Detection of errors with more than two bits is not guaranteed, but these errors are even less likely to occur. An L2 multi-bit ECC error can be detected in a cacheline only when there is a hit to a valid L2 tag for that cacheline.

3.3.3 L2 Tag Parity and Multi-Bit ECC Error Recovery

An L2 error can be an L2 tag parity error or an L2 multi-bit ECC error. When L2 hits in a set and detects an error, it forwards the hit data to the core, along with the assertion of the core_fault_in signal. The core generates the machine check exception as soon as it detects the assertion of core_fault_in.

If the L1 instruction cache requested the hit data with which core_fault_in was asserted, the device takes a machine check interrupt and the cacheline updates the L1 instruction cache. As the L1 cache can be

corrupted by an L2 error, flash invalidate both the L1 instruction cache and the L2 cache to recover from an L2 error. The L1 instruction cache cannot contain modified data and the L2 cache is write-through by design, so the flash invalidate of either cache affects performance but no modified data is lost.

If the L1 data cache requested the hit data with which core_fault_in was asserted, the device takes a machine check interrupt, the cacheline is removed, and the L1 data cache is not updated. When an L2 error is signaled when the L1 data cache requests the hit data, it is not clear whether the L1 data cache request is due to a load or store operation.

A store operation that misses in the L1 data cache and hits in the L2 cache merges with the hit data when the L2 returns the cacheline to the core. If an L2 error is detected and core_fault_in is asserted, the merged cacheline is removed and the L1D cache is not updated. When the data space is marked cacheable and write-back, a data loss occurs if there is an L1D miss and an L2 hit with core_fault_in assertion due to L2 error. This data loss scenario can also occur if the L1D is disabled. As it cannot be determined whether the operation is due to a load or store, an assumption is made that every occurrence of this scenario is a possible loss of data. The workarounds for this loss of data scenario are as follows:

- L2 instruction-only mode guarantees that no stores hit in the L2, so the loss of data scenario never occurs. L2IO mode incurs a performance penalty because no data is cached in the L2.
- Write-through mode on the cacheable data space guarantees that all modified data is stored to external memory. The error scenario occurs and is detected, but there is no loss of data. Write-through mode incurs a performance penalty because all stores must propagate to external memory.

3.4 Unprotected Memory

The L1 tag/status array, the MMU arrays, the branch target cache, and various compiled SRAMs around the microprocessor are not protected by ECC or parity because they are small in relation to the cache arrays. Also, they are often larger cells with more stored charge and therefore less vulnerable to soft errors. Nevertheless, the results of a soft error are unpredictable. For example, an error in the MMU could result in some kind of TLB fault. An error in the branch target cache could result in a program error.

The existence of unprotected memory and the difficulties of dealing with parity detection on modified data lead us to wonder why all memory is not protected by ECC or better protected by parity. One reason is that the delays associated with ECC or parity generation and detection affect the performance of some of the most time-critical functions in a microprocessor. Also, storing parity or ECC bits increases the size and cost of the die. In each successive process and each architecture generation, there are trade-offs between the amount of soft error protection required by customer applications and the cost in price and performance. Today, the largest arrays, which are more likely to experience soft errors, are the only arrays that are judged to require error correction or detection.

4 External DDR SDRAM ECC

The PowerQUICC™ III DDR utilizes a 64 bit data word, with 8 ECC syndrome bits, for a total of 72 bits of data. The examples used in Section 2.2, “Error Correcting Codes” are simplified cases of SECDED ECC logic applied to 8-bit data words.

The syndrome bits for the DDR ECC are assigned to data bits as listed in Table 1.

Table 1. DDR SDRAM ECC Syndrome Encoding

Data Bit	Syndrome Bit							
	0	1	2	3	4	5	6	7
0	•	•						•
1	•		•					•
2	•			•				•
3	•				•			•
4	•	•				•		
5	•		•			•		
6	•			•		•		
7	•				•	•		
8	•	•					•	
9	•		•				•	
10	•			•			•	
11	•				•		•	
12	•	•				•	•	•
13	•		•			•	•	•
14	•			•		•	•	•
15	•				•	•	•	•
16		•	•					•
17		•		•				•
18		•			•			•
19	•	•			•			
20		•	•			•		
21		•		•		•		
22		•			•	•		
23	•	•			•	•		•
24		•	•				•	
25		•		•			•	
26		•			•		•	
27	•	•			•		•	•
28		•	•			•	•	•
29		•		•		•	•	•

Data Bit	Syndrome Bit							
	0	1	2	3	4	5	6	7
32			•	•				•
33			•		•			•
34	•		•		•			
35		•	•		•			
36			•	•		•		
37			•		•	•		
38	•		•		•	•		•
39		•	•		•	•		•
40			•	•			•	
41			•		•		•	
42	•		•		•		•	•
43		•	•		•		•	•
44			•	•		•	•	•
45			•		•	•	•	•
46	•		•		•	•	•	
47		•	•		•	•	•	
48		•				•	•	
49			•			•	•	
50				•		•	•	
51	•					•	•	
52		•				•		•
53			•			•		•
54				•		•		•
55	•					•		•
56		•					•	•
57			•				•	•
58				•			•	•
59	•						•	•
60				•	•		•	
61	•			•	•		•	•

Table 1. DDR SDRAM ECC Syndrome Encoding (continued)

Data Bit	Syndrome Bit									Data Bit	Syndrome Bit							
	0	1	2	3	4	5	6	7			0	1	2	3	4	5	6	7
30		•			•	•	•	•		62		•		•	•		•	•
31	•	•			•	•	•			63			•	•	•		•	•

As listed in Table 1, syndrome bits are not assigned exactly as per the example shown in Section 2.2, "Error Correcting Codes." Instead, the parity bit, R0, is spread out amongst the syndrome bits for different 16-bit words (that is, R1 is parity for D[16 : 31]). Therefore, from Table 1, it can be seen that R3, for example, is equal to the XOR of D2, D6, D10, D14, D17, D21, D25, D29, D32, D36, D40, D44, D50, D54, D58, D60, D61, D62, and D63. Using Table 1, it is possible to generate, by hand, ECC syndrome bits for a given data-word. For example, for the 64-bit data-word, 0x0123_4567_0123_4567, the syndrome bits are 0x4B.

5 Initializing ECC on PowerQUICC™ III Processor

DDR SDRAM ECC is activated on PowerQUICC III processor by setting DDR_SDRAM_CFG[ECC_EN] = 1. Once this bit is set, single error detection, correction, and multiple bit error detection is active, though errors are reported to software. At this time, one clock cycle is added to the read path by the memory controller in order to check ECC and correct single-bit errors. (ECC generation does not add a cycle to the write path).

Before enabling ECC error reporting one must first program the entire DDR RAM with an initial value. The initial value need not be 0x0, and need not be the same value for the entire memory array. If the cache is enabled at this time, then one must do a cache flush so as to ensure that the initialization data makes it to the external memory. This initial value programming is necessary so that the data in the ECC RAM corresponds to the data in the SDRAM word and the ECC does not detect errors due to an initial mismatch. Note that ECC error reporting is enabled by default, so it must be disabled before the first write to DDR space, or else all write generate errors.

The PowerQUICC III Hip7 and PowerQUICC III lite (MPC8560, MPC8540, MPC8555, and MPC8541) processor that supports DDR1 memory, the writing of an initial value to DDR space can be facilitated through the use of the DMA engine on the PowerQUICC III processor. This minimizes the load on the CPU, and speed up a potentially long process.

The following is an example of DMA initialization using code taken from U-boot initialization:

```
void dma_init(void) {
volatile immap_t *immap = (immap_t *)CFG_IMMR;
volatile ccsr_dma_t *dma = &immap->im_dma;

dma->satr0 = 0x02c40000;
dma->datr0 = 0x02c40000;
asm("sync; isync; msync");
return;
}
uint dma_check(void) {
volatile immap_t *immap = (immap_t *)CFG_IMMR;
```

```
volatile ccsr_dma_t *dma = &immap->im_dma;
volatile uint status = dma->sr0;

/* While the channel is busy, spin */
while((status & 4) == 4) {
status = dma->sr0;
}

if (status != 0) {
printf ("DMA Error: status = %x\n", status);
}
return status;
}

int dma_xfer(void *dest, uint count, void *src) {
volatile immap_t *immap = (immap_t *)CFG_IMMR;
volatile ccsr_dma_t *dma = &immap->im_dma;

dma->dar0 = (uint) dest;
dma->sar0 = (uint) src;
dma->bcr0 = count;
dma->mr0 = 0xf000004;
asm("sync;isync;msync");
dma->mr0 = 0xf000005;
asm("sync;isync;msync");
return dma_check();
}
```

Later, the PowerQUICC III processors, including the MPC8548, support automatic initialization of DDR memory. To utilize this feature, software must provide a value to be loaded into DDR in the DDR_DATA_INIT register, and then initialize DDR by setting DDR_SDRAM_CFG_2[DINT].

Finally, after setting all memory to an initial value error reporting is then initialized through the ERR_DISABLE register. ERR_DISABLE[MBED] = 0 reports multiple bit errors and ERR_DISABLE[SBED] = 0 reports single-bit error reporting. Once reporting is turned on, the processor can be setup to generate interrupts based upon errors through the ERR_INT_ER register.

To summarize, the entire sequence for initialization of DDR ECC is as follows:

1. Enable ECC by setting DDR_SDRAM_CFG[ECC_EN] = 1.
2. Disable ECC error reporting (MBED = 1 and SBED = 1) in the ERR_DISABLE register.
3. Write dummy data to the entire DDR memory to initialize the ECC syndrome bits.
4. Enable ECC error reporting via ERR_DISABLE[MBED,SBED] = 00.

5.1 Debugging DDR ECC

In debugging DDR memory issues, it may be desirable to read back syndrome bits that are stored into memory. This is not a straightforward process, though it is possible through the following steps:

1. Enable ECC by following the procedure as described in Section 5, “Initializing ECC on PowerQUICC™ III Processor.”
2. Store data to memory.

A word to memory is written for the desired ECC. Assuming everything is working correctly, and ECC is enabled properly, the memory controller writes the corresponding syndrome bits to memory.

3. Disable ECC by setting DDR_SDRAM_CFG[ECC_EN] = 0.
4. Write a different word to memory at the same location.

   A different word is written to the same memory location as specified in step 2. As ECC is disabled, the memory controller does not touch the syndrome bits already existing in memory.
5. Enable ECC again by setting DDR_SDRAM_CFG[ECC_EN] = 1 and making sure error reporting is turned on.
6. Read memory location.

   It generates an ECC error as the word written in step 4 does not matches the syndrome bits written in step 2. At this point, it is possible to read back the value of the ECC syndrome bits stored in memory, corresponding to the data written in step 2 by reading the value of register, Capture_ECC.

5.2 Testing ECC

The PowerQUICC™ III DDR controller has the ability to inject errors into the ECC words and/or syndrome bits in an effort to enable testing and debugging of the ECC interface.

If errors are enabled via ECC_ERR_INJECT[EIEN], a bit in either DATA_ERR_INJECT_HI or DATA_ERR_INJECT_LO forces the corresponding bit in the data word to be inverted upon every 64-bit write to memory. Subsequent reads should then generate an ECC error. Instead of writing the correctly generated ECC syndrome bits, ECC_ERR_INJECT[EMB] enables the mirroring of the most significant byte of the data word into the ECC byte upon a write to memory. Additionally, ECC_ERR_INJECT[EEIM] enables the inverting of ECC syndrome bits upon writes to memory.

These three error injection modes enable the debugging of the ECC hardware itself, in the event of a board level issue involving ECC.

6 Conclusion

With ever increasing densities of on and off chip memory's, combined with the ongoing decrease in memory cell voltages, soft errors are becoming a greater problem that must be considered in any new system design.

New technologies are introduced with the PowerQUICC™ III processor, which help to make the processor less susceptible to soft errors than many previous generation processors. The combination of ECC and parity enable us to detect and correct errors on internal caches as well as external DDR SDRAM memories, which results in a more robust embedded design.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
+1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN3532
Rev. 0
11/2007

freescale™
semiconductor