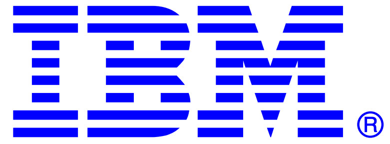


Power ISA™ Transactional Memory

December 12, 2012

The specifications in this RFC publication are preliminary and subject to change without notice. Periodic changes to this publication may be incorporated in new additions or supplements to this publication. This publication is provided "AS IS" and IBM Corporation makes no warranty of any kind, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

IBM® and POWER® are trademarks of IBM Corp., registered in many jurisdictions worldwide.

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication or disclosure is subject to restrictions set fourth in GSA ADP Schedule Contract with IBM Corporation.

© Copyright International Business Machines Corporation, 1994, 2012. All rights reserved.

RFC02183: Transactional Memory

Significance: Major

Status: New

Date: December 12, 2012
Target Version: 2.07
Source Version: 2.06
Books and sections affected:
Book 1:
- Section 1.4.5 Categories
- Section 2.2 Instruction Execution Order
- Section 2.3.1 Condition Register
- Section 3.2.2 Fixed-Point Exception Register
- Section 3.3.18 Move To/From System Register Instructions

Book 2:
- Section 1.1 Definitions
- Section 1.7.3.1 Reservations
- Section 1.7+ Transactions [Category: Transactional Memory]
- Section 4.1 Parameters Useful to Application Programs
- Section 4.3.2 Data Cache Instructions
- Section 4.4.3 Memory Barrier Instructions
- Chapter 7+ Transactional Memory Facility [Category: Transactional Memory]
- Section B.3+ Transactional Lock Elision [Category: Transactional Memory]

Book 3-S:
- Section 2.6 Processor Compatibility Register
- Section 3.2.1 Machine State Register
- Section 3.2.1+ State Transitions Associated with the Transactional Memory Facility [Category: Transactional Memory]
- Section 3.3.1 System Linkage Instructions
- Section 3.3.2 Power-Saving Mode Instructions
- Section 4.4.2+ Transactional Memory Instructions [Category: Transactional Memory]
- Section 4.4.3 Move To/From System Register Instructions
- Section 5.7.8: Reference and Change RecordingTo convey that transactional accesses are permitted to modify R/C/TS bits before the transaction commits, and even if the transaction fails, make the following changes in Figure 25:
- Section 5.9.3.1 SLB Management Instructions
- Section 5.9.3.3 TLB Management Instructions
- Section 5.10.1 Page Table Updates
- Sectino 6.4.3 Interrupt Processing
- Section 6.5 Interrupt Definitions
- Section 6.5.9 Program Interrupt
- Section 6.5.14 Trace Interrupt [Category: Trace]
- Section 6.5.21+ Facility Unavailable Interrupt
- Section 6.7 Exception Ordering
- Section 6.7 Exception Ordering
- Section 6.8 Interrupt Priorities

Appendices:
- Appendix C. Platform Support Requirements
- Appendix D. Complete SPR List
- Appendices G-J Opcode Map and Lists

Summary:

Transactional memory is a shared-memory synchronization construct allowing an application to perform a sequence of storage accesses that appear to occur atomically with respect to other threads.

Motivation

As systems include increasing levels of multiprocessing, and power constraints gate the performance improvements obtainable from frequency scaling, further performance gains are dependent on finding and exploiting parallelism. The described transactional memory architecture is motivated by several software usage cases, each involving optimistic execution as a means of exposing parallelism.

1. **Optimistic execution of lock-based programs**. Through speculative execution, the transactional facility will be used to execute lock-based critical sections without acquiring a lock, providing the benefits of fine-grain locking to applications whose locking protocols have not been carefully tuned for performance.
2. **Transactional programming in high-level languages**. The transactional programming model is an evolving industry-wide standard, offering programmer productivity advantages relative to lock-based shared memory programs. The Power ISA extensions described here will ensure that Power systems are competitive in support for this programming model.
3. **Checkpoint/Rollback usage.** This architecture will be used by some applications for its ability to

PRELIMINARY

checkpoint and restore architectural state, independent of its atomic storage access guarantees.

.

Changes to the Books

Book 1:

Section 1.4.5 Categories

Add “Transactional Memory”, abbreviation “TM”, notes “Full hardware Transactional Memory support” to the table after Trace.

Section 2.2 Instruction Execution Order

Insert a new third bullet in the first bulleted list, adding transaction failure to the list of things that can cause instruction fetching to be redirected.

------------------------- Begin text -------------------------------

In general, instructions appear to execute sequentially, in the order in which they appear in storage. The exceptions to this rule are listed below.

- *Branch* instructions for which the branch is taken cause execution to continue at the target address specified by the Branch instruction.
- *Trap* instructions for which the trap conditions are satisfied, and *System Call* instructions, cause the appropriate system handler to be invoked.
- Transaction failure will eventually cause the transaction's failure handler, implied by the ***tbegin.*** instruction, to be invoked. See the programming note following the ***tbegin.*** description in Section 8.5.
- Exceptions can cause the system error handler to be invoked, as described in Section 1.10, “Exceptions” on page 23.
- Returning from a system service program, system trap handler, or system error handler causes execution to continue at a specified address.

--------------------------- End text -------------------------------

Section 2.3.1 Condition Register

Extend the current 6th bullet to include tcheck. (NOTE THAT OTHER RFCs ALSO CHANGE THIS LIST.)

------------------------- Begin text -------------------------------

The bits in the Condition Register are grouped into eight 4-bit fields, named CR Field 0 (CR0), ..., CR Field 7 (CR7), which are set in one of the following ways.

- Specified fields of the CR can be set by a move to the CR from a GPR (***mtcrf***, ***mtocrf***).
- A specified field of the CR can be set by a move to the CR from another CR field (***mcrf***), from $XER_{32:35}$ (***mcrxr***), or from the FPSCR (***mcrfs***).
- CR Field 0 can be set as the implicit result of a fixed-point instruction.
- CR Field 1 can be set as the implicit result of a floating-point instruction.
- CR Field 6 can be set as the implicit result of a vector instruction.
- A specified CR field can be set as the result of a *Compare* instruction or of a ***tcheck*** instruction (see Book II).
- CR Field 1 can be set as the implicit result of a decimal floating-point instruction.

--------------------------- End text -------------------------------

Insert the following text before the store conditional paragraph, describing how TM instructions set the CR.

------------------------- Begin text -------------------------------

With the exception of ***tcheck*** , the Transactional Memory instructions set $CR0_{0:2}$ indicating the state of the facility prior to instruction execution, or transaction failure. A complete description of the meaning of these bits is given in the instruction descriptions in <insert crossref to TM description (Book II, Section 8.5?)>. These bits are interpreted as follows:

CR0	Description
000 || 0	Transaction state of Non-transactional prior to instruction
010 || 0	Transaction state of Transactional prior to instruction
001 || 0	Transaction state of Suspended prior to instruction
101 || 0	Transaction failure

The ***tcheck*** instruction similarly sets bits 1 and 2 of CR field BF to indicate the transaction state, and additionally sets bit 0 to TDOOMED, as defined in <insert cross ref to TM description (Section 8.2.1?)>.

CR field BF	Description
TDOOMED || 00 || 0	Transaction state of Non-transactional prior to instruction
TDOOMED || 10 || 0	Transaction state of Transactional prior to instruction
TDOOMED || 01 || 0	Transaction state of Suspended prior to instruction

Programming Note

Setting of bit 3 of the specified CR field to zero by ***tcheck*** and of field $CR0_3$ to zero by other TM instructions is intended to preserve these bits for future function. Software should not depend on the bits being zero.

--------------------------- End text -------------------------------

PRELIMINARY

Section 3.2.2 Fixed-Point Exception Register

Modify the description of FXCC to include tcheck.

-------------------------- Begin text --------------------------------

---------------------------- End text --------------------------------

Section 3.3.18 Move To/From System Register Instructions

Add TFHAR, TFIAR, TEXASR, and TEXASRU to the mtspr table.

Near the end of the mtspr description, add a statement that a move to a TM SPR in other than Non-transactional state causes a TM Bad Thing type Program interrupt.

-------------------------- Begin text --------------------------------

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
8	00000 01000	LR
9	00000 01001	CTR
13	00000 01101	AMR[5]
128	00100 00000	TFHAR[5]
129	00100 00001	TFIAR[5]
130	00100 00010	TEXASR[5]
131	00100 00011	TEXASRU[5]
256	01000 00000	VRSAVE
512	10000 00000	SPEFSCR[2]
896	11100 00000	PPR[3]
898	11100 00010	PPR32[4]

1 Note that the order of the two 5-bit halves of the SPR number is reversed.
2 Category: SPE.
3 Category: Server; see Book III-S.
4 Category: Phased-In. See Section 3.1 of Book II.
5 Category: Transactional Memory. See <crossref to Bk2 Ch 7+>.

If execution of this instruction is attempted specifying an SPR number that is not shown above, or an SPR number that is shown above but is in a category that is not supported by the implementation, one of the following occurs.

- If spr_0 = 0, the illegal instruction error handler is invoked.
- If spr_0 = 1, the system privileged instruction error handler is invoked.

If an attempt is made to execute ***mtspr*** specifying a TM SPR in other than Non-transactional state, a TM Bad Thing type Program interrupt is generated.

A complete description of this instruction can be found in Book III.

Special Registers Altered:
See above

---------------------------- End text --------------------------------

Zero the output for mfspr specifying the TFIAR when executed from a privilege level lower than that in which the TFIAR was set. Add TFHAR, TFIAR, TEXASR, and TEXASRU to the mfspr table.

-------------------------- Begin text --------------------------------

Move From Special Purpose Register XFX-form

mfspr RT,SPR

31	RT	spr	339	/
0	6	11	21	31

```
n ← spr5:9 || spr0:4
if n = 129 then see Book III-S
else
  if length(SPR(n)) = 64 then
     RT ← SPR(n)
  else
     RT ← 320 || SPR(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in the table below. If the SPR field contains 129, the instruction references the Transaction Failure Instruction Address Register (TFIAR)<TM> and the result is dependent on the privilege with which it is executed. See Book III-S. Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

decimal	SPR[1] $spr_{5:9}$ $spr_{0:4}$	Register Name
1	00000 00001	XER
8	00000 01000	LR
9	00000 01001	CTR
13	00000 01101	AMR[8]
128	00100 00000	TFHAR[8]
129	00100 00001	TFIAR[8]
130	00100 00010	TEXASR[8]
131	00100 00011	TEXASRU[8]
136	00100 01000	CTRL
256	01000 00000	VRSAVE
259	01000 00011	SPRG3
260	01000 00100	SPRG4[2]

1 Note that the order of the two 5-bit halves of the SPR number is reversed.
2 Category: Embedded.
3 See Chapter 5 of Book II.
4 Category: SPE.
5 Category: Alternate Time Base.
6 Category: Server; see Book III-S.
7 Category: Phased-In. See Section 3.1 of Book II.
8 Category: Transactional Memory. See <crossref to Bk2 Ch 7+>.

PRELIMINARY

decimal	SPR[1] spr$_{5:9}$ spr$_{0:4}$	Register Name
261	01000 00101	SPRG5[2]
262	01000 00110	SPRG6[2]
263	01000 00111	SPRG7[2]
268	01000 01100	TB[3]
269	01000 01101	TBU[3]
512	10000 00000	SPEFSCR[4]
526	10000 01110	ATB[3,5]
527	10000 01111	ATBU[3,5]
896	11100 00000	PPR[6]
898	11100 00010	PPR32[7]

1 Note that the order of the two 5-bit halves of the SPR number is reversed.
2 Category: Embedded.
3 See Chapter 5 of Book II.
4 Category: SPE.
5 Category: Alternate Time Base.
6 Category: Server; see Book III-S.
7 Category: Phased-In. See Section 3.1 of Book II.
8 Category: Transactional Memory. See <crossref to Bk2 Ch 7+>.

---------------------------- End text --------------------------------

Book 2:

Section 1.1 Definitions

Add transaction failure to the list of causes of deviation from the sequential execution model.

-------------------------- Begin text --------------------------------

- **program order**
  The execution of instructions in the order required by the sequential execution model. (See Section 2.2 of Book I.) A ***dcbz*** instruction that modifies storage which contains instructions has the same effect with respect to the sequential execution model as a *Store* instruction as described there. An additional exception to the sequential execution model beyond those described in Book I is caused by transaction failure (see <crossref to description of transaction failure handling>).

---------------------------- End text --------------------------------

Add the definition of aggregate store as a new bullet at the end of the section.

-------------------------- Begin text --------------------------------

- **aggregate store**
  The set of stores caused by a successful transaction, which are performed as an atomic unit.

---------------------------- End text --------------------------------

Section 1.7.3.1 Reservations

Rather than continue to maintain the classification of the causes of reservation loss in the second sentence of the third paragraph, replace that sentence as follows.

-------------------------- Begin text --------------------------------

A processor has at most one reservation at any time. A reservation is established by executing a ***lbarx***, ***lharx***, ***lwarx,*** or ***ldarx*** instruction, as described in item 1 below, and is lost or may be lost, depending on the item, if any of the following occur. Items 1-8 apply only if the relevant access is performed. (For example, an access that would ordinarily be caused by an instruction might not be performed if the instruction causes the system error handler to be invoked.)

---------------------------- End text --------------------------------

Add the following as a new third item in the list of causes of reservation loss.

-------------------------- Begin text --------------------------------

4. <TM> Any of the following occurs on the processor holding the reservation.
   a. The transaction state changes (from Non-transactional, Transactional, or Suspended state to one of the other two states; see Section 8.2, “Transactional Memory Facility States”), except in the following cases
      - If the change is from Transactional state to Suspended state, the reservation is not lost.
      - If the change is from Suspended state to Transactional state, the reservation is not lost if it was established in Transactional state.
      - If the change is caused by a ***treclaim.*** or ***trechkpt.*** instruction, whether the reservation is lost is undefined.
   b. The transaction nesting depth (see Section 8.4, “Transactional Memory Facility Registers”) changes; whether the reservation is lost is undefined. (This item applies only if the processor is in Transactional state both before and after the change.)
   c. The processor is in Suspended state and executes a *Store Conditional* instruction (***stbcx.***, ***sthcx.***, ***stwcx.***, or ***stdcx.***) or a ***waitrsv*** instruction; the reservation is lost if it was established in Transactional state. In this case the *Store Conditional* instruction’s store is not performed, and the ***waitrsv*** does not wait. (For *Store Conditional*, the reservation is also lost if it was established in Suspended state; see item 2.)

---------------------------- End text --------------------------------

Section 1.7+ Transactions [Category: Transactional Memory]

Add the following section before the section on instruction storage.

PRELIMINARY

------------------------- Begin text -------------------------------

A transaction is a group of instructions that collectively have unique storage access behavior intended to facilitate parallel programming. (It is possible to nest transactions within one another. The description in this chapter will ignore nesting because it does not have a significant impact on the properties of the memory model. Nesting and its consequences will be described elsewhere.) Sequences of instructions that are part of the transaction may be interleaved with sequences of Suspended state instructions that are not part of the transaction. A transaction is said to *succeed* or to *fail*, and failure may happen before all of the instructions in the transaction have completed. If the transaction fails, it is as if the instructions that are part of the transaction were never executed. If the transaction succeeds, it appears to execute as an atomic unit as viewed by other processors and mechanisms. (Although the transaction appears to execute atomically, some knowledge of the inner workings will be necessary to avoid apparent paradoxes in the rest of the model. These details are described below.) The execution of Suspended state sequences have the same effect that the sequence would have in the absence of a transaction, independent of the success or failure of the transaction, including accessing storage according to the weakly consistent storage model or SAO, based on storage attributes. Upon failure, normal execution continues at the failure handler. Except for the rollback of the effects of transactional instructions upon transaction failure, as viewed by the executing thread, the interleaved sequences of Transactional and Suspended state instructions appear to execute according to the sequential execution model. See <cross ref to bk2 ch7+> for more details. The unique attributes of the storage model for transactions are described below.

Transaction processing does not support the rollback of operations on the reservation mechanism. To prevent this possibility, a reservation is lost as a result of a state change from Transactional to Non-transactional or Non-transactional to Transactional. It is possible to successfully complete an atomic update in Transactional state, though such a sequence would have no benefit. It is also possible to complete an atomic update in Suspended state, or straddling an interval in Suspended state if Suspended state is entered via an interrupt or ***tsuspend.*** and exited via ***tresume.***, , ***rfid***, ***hrfid***, or ***mtmsrd***. However, an atomic update will not succeed if only one of the *Load and Reserve* / *Store Conditional* instruction pair is executed in Suspended state.

Programming Note

Note that if a *Store Conditional* instruction within a transaction does not store, it may still be possible for the transaction to succeed. Software must not depend on the two operations having the same outcome. For example, software must not use success of an enclosing transaction as a replacement for checking the condition code from a transactional *Store Conditional* instruction.

Programming Note

Accessing storage locations in Suspended state that have been accessed transactionally has the potential to create apparent storage paradoxes. Consider, for example, a case where variable X has intial value zero, is updated transactionally to one, is read in Suspended state, subsequently the transaction fails, and variable X is read again. In the absence of external conflicts, the observed sequence of values will be zero, one, zero: old, new, old.

Performing an atomic update on X in Suspended state may be even more confusing. Suppose the atomic sequence increments X, but that the only way to have X=1 is via the transactional store that occurs before entering Suspended state. The store conditional, if it succeeds, will store X=2 and in so doing, kill the transaction. But with the transaction having failed, X was never equal to one.

The flexibility of the Suspended state programming model can create unintuitive results. It must be used with care.

Successful transactions are serialized in some order, and no processor or mechanism is able to observe the accesses caused by any subset of these transactions as occurring in an order that conflicts with this order. Specifically, let processor i execute transactions 0, 1,…, j, j+1, …, where only successful transactions are numbered, and the numbering reflects program order. Let T_{ij} be transaction j on processor i. Then there is an ordering of the T_{ij} such that no processor or mechanism is able to observe the accesses caused by the transactions T_{ij} in an order that conflicts with this ordering. Note that Suspended state storage accesses are not included in the serialization property.

Programming Note

The ordering of the T_{ij} for a given i is consistent with program order for processor i.

Because of the difference between a transaction's instantaneous appearance and the finite time required to execute it in an implementation, it is exposed to changes in memory management state in a way that is not true for individual accesses. A change to the translation or protection state that would prevent any access

PRELIMINARY

from taking place at any time during its processing for the transaction compromises the integrity of the transaction. Any such change must either be prevented or must cause the transaction to fail. The architecture will automatically fail a transaction if the memory management state change is accomplished using ***tlbie***. An implementation may overdetect such conflicts between the ***tlbie*** and the transaction footprint. (Overdetection may result from the technique used to detect the conflict. A bloom filter may be used, as an example. Subsequent references to translation invalidation conflicts implicitly include any cases of spurious overdetection.) Changes made in some other manner must be managed by software, for example by explicitly aborting any affected transactions. Examples of instructions that require software management are ***tlbiel***, ***slbie***, ***slbia***, and ***tlbia***.

The atomic nature of a transaction, together with the cumulative memory barrier created by the transaction and the memory barriers created by ***tbegin.*** and ***tend.*** described below, has the potential to eliminate the need for explicit memory barriers within the transaction, and before and after the transaction as well. However, since there may be a desire to preserve existing algorithms while exploiting transactions, the interaction of memory barriers and transactions is defined. In the presence of transactions, storage access ordering is the same as if no transactions are present, with the following exceptions. Memory barriers that are created while the transaction is running (other than the integrated cumulative barrier of the transaction described below), data dependencies, and SAO do not order transactional stores. Instead, transactional stores are grouped together into an *aggregate store*, which is performed as an atomic unit with respect to other processors and mechanisms when the transaction succeeds, after all the transactional loads have been performed. With this store behavior, the appearance of transactional atomicity is created in a manner similarly to that for a *Load and Reserve* / *Store Conditional* pair. Success of the transaction is conditional on the storage locations specified by the loads not having been stored into by a more recent Suspended state store or by any store by another processor or mechanism since the load was performed. (There are additional conditions for the success of transactions.)

The *tbegin.* instruction creates a memory barrier that immediately precedes the transaction and orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair a_ib_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism. Set A contains all data accesses caused by instructions preceding the ***tbegin.*** that are neither Write Through Required nor Caching Inhibited. Set B contains all data accesses caused by instructions following the ***tbegin.***, including Suspended state accesses, that are neither Write Through Required nor Caching Inhibited.

A successful transaction has an integrated memory barrier behavior. When a processor (P1) executes a ***tend.*** instruction and ***tend.*** processing determines that the transaction will succeed, a memory barrier is created, which orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair a_ib_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism. Set A contains all non-transactional data accesses by other processors and mechanisms that have been performed with respect to P1 before the memory barrier is created and are neither Write Through Required nor Caching Inhibited. Set B contains the aggregate store and all non-transactional data accesses by other processors and mechanisms that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in set B. Note that the cumulative barrier does not order Suspended state storage accesses interleaved with the transaction.

A *tend.* instruction that ends a successful transaction creates a memory barrier that immediately follows the transaction and orders storage accesses pairwise, as follows. Let A and B be sets of storage accesses as defined below. For each pair a_ib_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor or mechanism. Set A contains all data accesses caused by instructions preceding the ***tend.,*** including Suspended state accesses, that are neither Write Through Required nor Caching Inhibited. Set B contains all data accesses caused by instructions following the ***tend.*** that are neither Write Through Required nor Caching Inhibited.

Section 1.7+.1 Rollback-Only Transactions

A Rollback-Only Transaction (ROT) is a sequence of instructions that is executed, or not, as a unit. The purpose of the ROT is to enable bulk speculation of instructions with minimum overhead. It leverages the rollback mechanism that is invoked as part of transaction failure handling, but has reduced overhead in that it does not have the full atomic nature of the transaction and its synchronization and serialization properties. The absence of a (normal) transaction's atomic quality means that a ROT must not be used to manipulate shared data.

PRELIMINARY

More specifically, a ROT differs from a normal transaction as follows.

- ROTs are not serialized.
- There are no barriers created by ***tbegin.*** and ***tend.***
- A ROT has no integrated cumulative barrier.
- There is no monitoring of storage locations specified by loads for modification by other processors and mechanisms between the performing of the loads and the completion of the ROT.
- The stores that are included in the ROT need not appear to be performed as an aggregate store. (Implementations are likely to provide an aggregate store appearance, but the correctness of the program must not depend on the aggregate store appearance.)

---------------------------- End text --------------------------------

Section 4.1 Parameters Useful to Application Programs

Add item 13 to the list: “Maximum transaction level”.

Section 4.3.2 Data Cache Instructions

Add a programming note at the end of the dcbst description warning against treating it like dcbf.

-------------------------- Begin text --------------------------------

---------------------------- End text --------------------------------

Section 4.4.3 Memory Barrier Instructions

Add another pointer to the introductory material, pointing to the description of transaction behavior in Bk2Ch1.

-------------------------- Begin text --------------------------------

The *Memory Barrier* instructions can be used to control the order in which storage accesses are performed. See <crossref to new sec 1.7+, Bk2> for a description of how the *Memory Barrier* instructions interact with transactions. Additional information about these instructions and about related aspects of storage management can be found in Book III.

---------------------------- End text --------------------------------

Chapter 7+ Transactional Memory Facility [Category: Transactional Memory]

Add a chapter describing TM at the end of Book 2.

-------------------------- Begin text --------------------------------

8.1 Transactional Memory Facility Overview

This chapter describes the registers and instructions that make up the transactional memory (TM) facility. Transactional memory is a shared-memory synchronization construct allowing an application to perform a sequence of storage accesses that appear to occur atomically with respect to other threads.

A set of instructions, special-purpose registers, and state bits in the MSR (see Book III) are used to control a transactional facility that is associated with each hardware thread. A ***tbegin.*** instruction is used to initiate transactional execution, and a ***tend.*** instruction is used to terminate transactional execution. Loads and stores that occur between the ***tbegin.*** and ***tend.*** instruction appear to occur atomically. An implementation may prematurely terminate transactional execution for a variety of reasons, rolling back all transactional storage updates that have been made by the thread since the ***tbegin.*** was executed, and rolling back the contents of a subset of the thread’s book I registers to their contents before the ***tbegin.*** was executed. In the event of such premature termination, control is transferred to a software failure handler associated with the transaction, which may then retry the transaction or choose an alternate path depending on the cause of transaction failure. A transaction can be explicitly aborted via a set of conditional abort instructions and an unconditional abort instruction, ***tabort.***. A ***tsr.*** instruction is used to suspend or resume transactional execution, while allowing the transaction to remain active.

Programming Note

A ***tbegin.*** should always be followed immediately by a ***beq*** as the first instruction of the failure handler, that branches to the main body of the failure handler. The failure handler should always either retry the transaction or use non-transactional code to perform the same operation. (The number of retries should be limited to avoid the possibility of an infinite loop. The limit could be based on the perceived permanence / transience of the failure.) A failure handler policy which includes trying a different transaction before returning to the one that failed may fail to make forward progress.

Programming Note

The architecture does not include a “fairness guarantee” or a “forward progress” guarantee for transactions. If two processors repeatedly conflict with one another in an attempt to complete a transaction, one of the two may always succeed while the other may always fail. If two processors repeatedly conflict with one another in an attempt to complete a transaction, both may always fail, depending on the details of the transaction. This is different from the behavior of a typical locking routine, in which one or the other of the competitors will generally get the lock.

PRELIMINARY

Transactions performed using this facility are “strongly atomic”, meaning that they appear atomic with respect to both transactional and non-transactional accesses performed by other threads. Transactions are isolated from reads and writes performed by other threads; i.e. transactional reads and writes will not appear to be interleaved with the reads and writes of other threads.

Nesting of transactions is supported using a form of nesting called *flattened nesting*, in which transactions that are initiated during transactional execution are subsumed by the pre-existing transaction. Consequently, the effects of a nested transaction do not become visible until the outer transaction commits, and if a nested transaction fails, the entire set of transactions (outer as well as nested) is rolled back, and control is transferred to the outer transaction’s failure handler. The barriers created by ***tbegin.*** and ***tend.*** and the integrated cumulative barrier that are described in <crossref to ch1 transaction description> are only created for outer transactions and not any transactions nested within them.

References to *Store* instructions, and stores, include ***dcbz*** and the storage accesses that it causes.

Rollback-Only Transactions

Rollback-Only Transactions (ROTs) differ from nornal transactions in that they are speculative but not atomic. They are initated by a unique variant of ***tbegin.*** They may be nested with other ROTs or with normal transactions. When a normal transaction is nested within a ROT, the behavior from the normal ***tbegin.*** until the end of the outer transaction is characteristic of a normal transaction. Although subject to failure from storage conflicts, the typical cause of ROT failure is via a *Tabort* variant that is executed after the program detects an error in its (software) speculation. Except where specifically differentiated or where differences follow from specific differentiation, the following description applies to ROTs as well as normal transactions.

8.1.1 Definitions

Commit: A transaction is said to *commit* when it successfully completes execution. When a transaction is committed, its transactional accesses become irrevocable, and are made visible to other threads. A transaction completes by either commiting or failing.

Speculative registers: The set of registers that are saved when a transaction is initiated, and restored upon transaction failure, is a subset of the architected register state, consisting of the General Purpose Registers, Floating-Point Registers, Vector Registers, Vector-Scalar Registers, and the following Special Registers and fields: CR fields other than CR0, LR, CTR, FPSCR, AMR, PPR, VRSAVE, VSCR, DSCR, and TAR. This subset is referred to as the *speculative register state.* The speculative register state includes all problem-state writable registers with the exception of CR0, , the performance monitor registers, and the Transactional Memory registers. With the exception updates of CR0 and the Transactional Memory registers, explicit updates of registers that are not included in the set of speculative registers are disallowed in Transactional state (i.e. will cause the transaction to fail), but are permitted in Suspended state. Suspended state modifications of these registers will not be rolled back in the event of transaction failure. (Modifications of Transactional Memory registers are only permitted in Non-transactional state. Attempts to modify Transactional Memory registers in other than Non-transactional state will cause a TM Bad Thing type Program interrupt.)

> **Programming Note**
>
> CR0 and the Transactional Memory registers (TFHAR, TEXASR, TFIAR) are not saved, or restored when the transaction fails, because they are modified as a side effect of transaction failure (so restoring them would lose information needed by the failure handler). The performance monitor registers are not saved or restored because saving and restoring them would add significant implementation complexity and is not needed by software. Also, these registers can be modified asynchronously by the processor, so restoring them when the transaction fails could cause loss of information.

Transactional accesses: Data accesses that are caused by an instruction that is executed when the thread is in the Transactional state (see Section 8.2) are said to be *transactional*, or to have been *performed transactionally.* The set of accesses caused by a committed normal transaction is performed as if it were a single atomic access. That is, it is always performed in its entirety with no visible fragmentation. The sets performed by normal transactions are thus serialized: each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors. Until a transaction commits, its set of transactional accesses is provisional, and will be discarded should the transaction fail.

Non-transactional accesses: Storage accesses performed in the existing Power® storage model are said to be *non-transactional*. In contrast to transactional storage accesses, there is no provision of atomicity across multiple non-transactional accesses. Non-transactional storage updates are not discarded in the event of a transaction failure.

Outer transaction: A transaction that is initiated from the Non-transactional state is said to be an *outer* transaction. A ***tbegin.*** instruction that initiates an outer transaction is sometimes referred to as an *outer* ***tbegin.***. Similarly, a ***tend.*** instruction with A=0 that ends

PRELIMINARY

an outer transaction is sometimes referred to as an outer ***tend.***.

Nested Transaction: A transaction that is initiated while already executing a transaction is said to be *nested* within the pre-existing transaction. The set of active nested transactions forms a stack growing from the outer transaction.

Failure: A transaction failure is an exceptional condition causing the set of transactional storage updates to be discarded, and speculative registers to be reverted to their pre-transactional values.

Failure handler: A failure handler is a software component responsible for handling transaction failure. On transaction failure, hardware redirects control to the failure handler associated with the outer transaction.

Conflict: A transactional memory access is said to conflict with another transactional or non-transactional access if both accesses reference the same storage block, and at least one of them is a store. If two transactions make conflicting accesses, at least one of them will fail. If a transaction fails as a result of a conflict with a store, the store may have been executed by another processor or may have been executed in Suspended state by the processor with the failing transaction. For a ROT, no conflict is caused if the ROT performs a load and another program performs a non-transactional store to the same block.

A transactional memory access is said to conflict with a ***tlbie*** if the storage location being accessed is in the page the translation for which is being invalidated by the tlbie. For a ROT, no conflict is caused if the access is a load.

A Suspended state cache control instruction is said to cause a conflict if it would cause the destruction of a transactional update or if it would make a transactional update visible to another thread.

8.2 Transactional Memory Facility States

The transactional memory facility supports several modes of operation, referred to in this document as the *transaction state*. These states control the behavior of storage accesses made during the transaction and the handling of transaction failure. Changes to transaction state affect all transactions currently using the transactional facility on the affected thread: the outer transaction as well as any nested transactions, should they exist.

Non-transactional: The default, initial state of execution; no transaction is executing. The transactional facility is available for the initiation of a new transaction.

Transactional: This state is initiated by the execution of a ***tbegin.*** instruction in the Non-transactional state. Storage accesses (data accesses) caused by instructions executed in the Transactional state are performed transactionally. Other storage accesses associated with instructions executed in the Transactional state (instruction fetches, implicit accesses)are performed non-transactionally. In the event of transaction failure, failure is recorded as defined in Section 8.3.2, and control is transferred to the failure handler as described in Section 8.3.3.

Suspended: The Suspended execution state is explicitly entered with the execution of a ***tsuspend.*** form of ***tsr.*** instruction during a transaction, the execution of a ***trechkpt.*** instruction from non-transactional state, or as a side-effect of interrupt while in the Transactional state. Storage accesses and accesses to SPRs that are not part of the speculative register state are performed non-transactionally; they will be performed independently of the outcome of the transaction. The initiation of a new transaction is prevented in this state. In the event of transaction failure, failure recording is performed as defined in Section 8.3.2, but failure handling is usually deferred until transactional execution is resumed (see Section 8.3.3 for details).

Until failure occurs, Load instructions that access storage locations that were transactionally written by the same thread will return the transactionally written data. After failure is detected, but before failure handling is performed, such loads may return either the transactionally written data, or the current non-transactional contents of the accessed location. The ***tcheck*** instruction can be used to determine whether any previous such loads may have returned non-transactional contents.

Suspended state Store instructions that access cache blocks that have been accessed transactionally (due to load or store) by the same thread may cause the transaction to fail.

Programming Note

The intent of the Suspended execution state is to temporarily escape from transactional handling when transactional semantics are undesirable. Examples of such cases include storage updates that should be retained in the event of transactional failure, which is useful for debugging, interthread communication, the access of Caching-Inhibited storage, and the handling of interrupts. In the event of transaction failure during the Suspended execution state, failure handling is deferred until transactional execution is resumed, allowing the block of Suspended state code to complete its activities.

PRELIMINARY

Programming Note

During Suspended state execution, accessing cache blocks that have been transactionally accessed by the same thread prior to entering Suspended state requires special care, because failure may occur due to uncontrollable events such as interactions with other threads or the operating system. Up until a transaction fails, loads from transactionally modified cache blocks will return the transactionally modified data. However once the transaction fails, the loads may return either the transactionally updated version of storage, or the most recent non-transactional version. Stores to transactionally modified blocks may or may not cause the thread's transaction to fail.

Table 1 enumerates the set of Transactional Memory instructions and events that can cause changes to the transaction state. Transaction states are abbreviated N (Non-transactional), T (Transactional), and S (Suspended). (Interrupts, and the , ***rfid***, ***hrfid***, and ***mtmsrd*** instructions, can also cause changes to the transaction state; see Book III.)

Programming Note

tbegin. in Suspended state merely updates CR0. When ***tbegin.*** is followed by ***beq***, this will result in a transfer to the failure handler. Nothing more severe (e.g. an interrupt) is required.

Instr/ event State	***tbegin.***	***tend.***	Abort caused by ***tabort.*** and conditional ***tabort.*** variants	***tsuspend.***	***tresume.***	**Failure**	**treclaim.**	**trechkpt.**
N[7]	T	N[3]	N[3]	N[3]	N[3]	Not applicable	N[8]	S[9]
T	T	N, if outer transaction or A=1 form; otherwise T	N[4,5]	S	T	N[4,5]	N[4]	S[8]
S	S[1]	S[8]	S[4]	S[3]	T[6]	S[4]	N[4]	S[8]

Notes

1. **CR0 updated indicating transactional initiation was unsuccessful, due to a pre-existing transaction occupying the transactional facility.**
3. **Execution of these operations does not affect transaction state, allowing for the instructions to be used in software modules called from Non-transactional, Transactional, and Suspended paths.**
4. **If failure recording has not previously occurred, failure recording is performed as defined in Section 8.3.2.**
5. **Failure handling is performed as defined in Section 8.3.3.**
6. **If failure has occurred during Suspended execution, failure handling will be performed sometime after the execution of *tresume*, and no later than the set of events listed in Section 8.3.3.**
7. **Any attempt to execute a TM instruction when MSR_{TM}=0 causes a Transactional Memory type of Facility Unavailable interrupt. Any attempt to execute a TM instruction in other than hypervisor state when $HFSCR_{TM}$=0 causes a Transactional Memory type of Hypervisor Facility Unavailable interrupt. The contents of this row assumes that Transactional Memory is enabled.**
8. **Generate TM Bad Thing type Program interrupt.**
9. **If $TEXASR_{FS}$=0, generate a TM Bad Thing type Program interrupt.**

Table 1: Transaction state transitions caused by TM instructions and transaction failure

8.2.1 The TDOOMED bit

The status of an active transaction is summarized by a transaction doomed bit (TDOOMED) that resides in an implementation-dependent location. When 0, it indicates that the active transaction is valid, meaning that it remains possible for the transaction to commit successfully, if failure does not occur before committing. When 1 it indicates that transaction failure has already occurred for the transaction.

The TDOOMED bit is set to 0 upon the successful initiation of an outer transaction by ***tbegin.***. It is set to 1 when failure occurs or as a result of executing ***trechkpt.***. When failure occurs, TDOOMED is set to 1 before any other effects of the transaction failure (recording the failure in TEXASR, rollback of transactional stores, over-writing of the transactionally accessed locations by a conflicting store, etc.) are visible to software executing on the processor that executed the transaction. In Non-transactional state, the value of TDOOMED is undefined.

8.3 Transaction Failure

8.3.1 Causes of Transaction Failure

A transaction failure is said to be “externally-induced” if the failure is caused by a thread other than the transactional thread. Likewise, a transaction failure is said to be “self-induced” if the failure is caused by the transactional thread itself.

For self-induced failure as a result of attempting to execute an instruction that is forbidden in the Transactional state, a Privileged Instruction type of Program Interrupt takes precedence over transaction failure. (For example, an attempt to execute ***stdcix*** in Transactional state and problem state will result in a Privileged Instruction type of Program interrupt.) Transaction failure takes precedence over all other interrupt types. The relevant instructions are listed in the third bullet of the second set of bullets below and the first bullet in the third set of bullets below.

In general, a ROT will not fail in the following scenarios when the failure is specified as a conflict on a transactional access and the access is a load.

Transactions will fail for the following externally-induced causes

- Conflict with transactional access by another thread
- Conflict with non-transactional access by another thread
- Conflict with a translation invalidation caused by a ***tlbie*** performed by another thread

Transactions will fail for the following self-induced causes

- Abort caused by the execution of ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.*** or ***treclaim.*** instruction.
- Transaction level overflow, defined as an attempt to execute ***tbegin.*** when the transaction level is already at its maximum value
- Footprint overflow, defined as an attempt to perform a storage access in Transactional state which exceeds the capacity for tracking transactional accesses.
- Execution of the following instructions while in the Transactional state: ***doze, sleep, nap, rvwinkle, icbi, , dcbf, dcbi, dcbst, [h]rfid, , mtmsr[d], mtsle, mtsr, mtsrin, msgsnd, msgsndp, msgclr, msgclrp, slbie, slbia, slbmte, slbfee,*** and ***tlbie[l].*** (These instructions are considered to be *disallowed* in Transactional state.) The disallowed instruction is not executed; failure handling occurs before it has been executed.

> **Programming Note**
>
> Note that execution of a Power Saving instruction in Suspended state causes a TM Bad Thing type Program interrupt.

- Execution, while in Transactional state, of ***mtspr*** specifying an SPR that is not part of the speculative register state and is not a Transactional Memory SPR. The ***mtspr*** is not executed; failure handling occurs before it has been executed. (Modification of XER_{FXCC} and CR_{CR0} are allowed, but the changes will not be rolled back in the event of transaction failure.)
- Conflict caused by a Suspended state store to a block that was previously accessed transactionally.
- Conflict caused by a Suspended state ***tlbie*** that specifies a translation that was previously used transactionally. (This case will be recorded as a translation invalidation conflict because it may be hard to differentiate from a conflict caused by a ***tlbie*** performed by another thread and because it is highly likely to be a transient failure.)

For each of the following potential causes, the transaction will fail if the absence of failure would compromise transaction semantics; otherwise, whether the transaction fails is undefined.

- Execution of the following instructions while in the Transactional state: ***eciwx, ecowx, lbzcix, ldcix, lhzcix, lwzcix, stbcix, stdcix, sthcix, stwci***. The disallowed instruction is not executed; failure handling occurs before it has been executed. (These instructions are considered to be disallowed in Transactional state if they cause transaction failure in Transactional state.) Execution of these instructions in the Suspended state is allowed and does not cause transaction failure.
- Execution of the following instructions in the Transactional state: ***wait***, ***waitasec***. The disallowed instruction is not executed; failure handling occurs before it has been executed. (These instructions are considered to be disallowed in a transaction if they cause transaction failure.)
- Execution of the following instructions in the Suspended state: ***wait***, ***waitasec***. The disallowed instruction is treated as a no-op; failure recording occurs. (These instructions are considered to be disallowed in a transaction if they cause transaction failure.)
- Access of a disallowed type while in the Transactional state: Caching Inhibited, Write Through Required, and Memory Coherence not Required for data access; Caching-Inhibited for instruction fetch. The disallowed access is not performed; failure handling occurs such that the instruction that would cause (or be associated with, for instruction fetch) the disallowed access type appears not to have been executed. Accesses of this type in the Suspended state are allowed and do not cause transaction failure.

PRELIMINARY

- Instruction fetch from a block that was previously written transactionally (reported as a unique cause that includes both self-induced and externally-induced instances)
- ***dcbf***, ***dcbi***, or ***icbi*** specifying a block that was previously accessed transactionally, in either of the following cases.

  Programming Note

  Note that ***dcbf*** with L=3 should never compromise transactional semantics, but it is still permitted to cause transaction failure in Suspended state and it is disallowed in Transactional state.

  - the instruction (***dcbf***, ***dcbi***, or ***icbi***) is executed in Suspended state on the processor executing the transaction (self-induced conflict)
  - the instruction is executed by another processor (externally-induced conflict)
- ***dcbst*** specifying a block that was previously written transactionally, in either of the following cases.
  - ***dcbst*** is executed in Suspended state on the processor executing the transaction (self-induced conflict)
  - ***dcbst*** is executed by another processor (externally-induced conflict)
- Cache eviction of a block that was previously accessed transactionally

Programming Note

WARNING: Software should not depend for its correct execution on the behavior (whether or not the relevant transaction fails) of the cases described in the preceding set of bullets. The behavior is likely to vary from design to design. Such a dependence would impact the software's portability without any tangible advantage.

Programming Note

Because the atomic nature of a transaction implies an apparent delay of its component accesses until they can be performed in unison, the use of cache control instructions to manage cache residency and/or the performing of storage accesses may have unexpected consequences. Although they may not cause transaction failure directly, their use in a transaction is strongly discouraged.

If an instruction or event does not cause transaction failure, it behaves as defined in the architecture.

The set of failure causes and events are further classified as *precise* and *imprecise* failure causes. All externally induced events are imprecise, and all self-induced events are precise with the exception of the following cases:

- Self-induced conflicts caused by instruction fetch
- Self-induced conflicts caused by footprint overflow
- Self-induced conflicts in Suspended state, caused by a store to a block that was previously accessed transactionally, or a ***dcbf***, ***dcbi***, or ***icbi*** specifying a block that was previously accessed transactionally, or a ***dcbst*** specifying a block that was previously written transactionally, or a ***tlbie*** specifying a translation that was previously used transactionally.

When failure recording and handling occur (as defined in Section 8.3.2 and 8.3.3) for a precise failure, they will occur precisely according to the sequential execution model, adhering to the following rules:

1. Effects of the failure occur such that all instructions preceding the instruction causing the failure appear to have completed with respect to the executing thread.
2. The instruction causing the failure may appear not to have begun execution (except for causing the failure), or may have completed, depending on the failure cause.
3. Architecturally, no subsequent instruction has begun execution.

Failure handling for imprecise failure types is guaranteed to occur no later than the execution of ***tend.*** with A=1 or $TEXASR_{TL}$ =1. Failure recording for imprecise failure types is guaranteed to occur no later than failure handling. Any operation that can cause imprecise failure if performed in-order can also cause imprecise failure if performed out-of-order.

Programming Note

Because instruction fetch from a transactionally written block may result in failure, it is recommended that transactionally accessed data and transactionally accessed instructions not be co-located within a single block.

Programming Note

The architecture does not detect and cause transaction failure for translation invalidations to transactionally accessed pages or segments, when the translation invalidation is caused by instructions other than ***tlbie*** (i.e. ***slbie, slbia, tlbiel, tlbia***). Consequently, software is responsible for aborting transactions in circumstances where such local translation invalidations may affect a local transaction.

PRELIMINARY

8.3.2 Recording of Transaction Failure

When transaction failure occurs, information about the cause and circumstances of failure are recorded in SPRs associated with the transactional facility. Failure recording is performed a single time per transaction that fails, controlled by the state of the TEXASR failure summary (FS) bit; when 0, FS indicates that failure recording has not already been performed, and is therefore permissible.

The following RTL function specifies the actions taken during the recording of transaction failure:

```
TMRecordFailure(FailureCause)
                  #FailureCause is 32-bit cause
code
if TEXASR_FS = 0
   if failure IA known then
      TFIAR <- CIA
      TEXASR_37 <- 1
   else
      TFIAR <- approximate instruction address
      TEXASR_37 <- 0
   TEXASR_0:31 <- FailureCause

   if MSR_TS=0b01 then TEXASR_Suspended <- 1
   TEXASR_PR <- MSR_PR
   TEXASR_HV <- MSR_HV
   TFIAR_HV PR <- MSR_HV PR
  TEXASR_FS <- 1
  TDOOMED <- 1
```

When failure recording occurs, the TEXASR and TFIAR SPRs are set indicating the source of failure. When possible, TFIAR is set to the effective address of the instruction that caused the failure, and $TEXASR_{37}$ is set to 1 indicating that the contents of TFIAR are exact. When the instruction address is not known exactly, an approximate value is placed in TFIAR and $TEXASR_{37}$ is set to 0. TEXASR bits 0:31 are set indicating the cause of the failure, and the $TEXASR_{Suspended}$, $TEXASR_{HV}$, $TEXASR_{PR}$, $TFIAR_{HV}$, and $TFIAR_{PR}$ bits are set indicating the machine state in which the failure was recorded. $TEXASR_{TL}$ is unchanged. The TDOOMED bit is set to 1.

Programming Note

TFIAR is intended for use in the debugging of transactional programs by identifying the source of transaction failure. Because TFIAR may not always be set exactly, software should test $TEXASR_{37}$ before use; if zero, the contents of TFIAR are an approximation.

8.3.3 Handling of Transaction Failure

After detection of failure, the timing of failure handling is dependent on the state of the transactional facility.

In Transactional state, failure handling may occur immediately, but an implementation is free to delay handling until one of the following failure handling synchronizing events occurs in Transactional state.

- An abort caused by the execution of a ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, or ***tabortwci.*** instruction.
- An attempt, in Transactional state, to execute a disallowed instruction, perform an access of a disallowed type, or execute an *mtspr* instruction that specifies an SPR that is not part of the speculative register state and is not a Transactional Memory SPR.
- An attempt to commit a transaction, caused by the execution of ***tend.*** with A = 1 or when $TEXASR_{TL}$ = 1.
- The execution of a ***treclaim.*** instruction.

If the failure is caused by an event in the preceding list, failure handling occurs immediately. (If the failure is caused by ***treclaim.***, CR0 is not set to indicate failure and the transaction's failure handler is not invoked.)

When failure handling occurs, speculative registers are reverted to their pre-transactional values, all transactional updates to storage are discarded if they have not previously been discarded, and any resources occupied by the transaction are discarded. CR0 is set to 0b101 || 0. The transaction state is set to Non-transactional, and control flow is redirected to the instruction address stored in TFHAR.

The following RTL function specifies the actions taken during the handling of transaction failure:

```
TMHandleFailure()
   If speculative storage updates have not previ-
ously been discarded
       Discard speculative storage updates
   Revert speculative registers to pre-transac-
tional values
   Discard all resources related to current trans-
action
  MSR_TS <- 0b00                #Non-transactional
  NIA <- TFHAR
  CR0 <- 0b101 || 0
```

Upon failure detected in Suspended state from causes other than the execution of a ***treclaim.***, ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, or ***tabortwci.*** instruction, failure recording occurs as described in Section 8.3.2, but failure handling is deferred until the transaction is resumed. Once resumed, failure handling will occur no later than the set of failure handling synchronizing events listed above. However, speculative storage updates may be discarded at any time between

PRELIMINARY

the time that failure is detected, up until the occurrence of one of the failure handling synchronizing events listed above. Upon failure in Suspended state caused by ***treclaim.***, failure recording, discarding of speculative storage updates, and reverting of the speculative registers to pre-transactional values are immediate (but CR0 is not set to indicate failure and the transaction's failure handler is not invoked). Upon failure in Suspended state caused by a conditional or unconditional *Abort* instruction, failure recording and discarding of speculative storage updates are immediate. The remainder of failure handling occurs immediately after the transaction is resumed (e.g., the next instruction executed after the ***tresume*** (or ***rfid***, etc.) is the instruction at the address contained in TFHAR).

Programming Note

A *Load* instruction executed immediately after ***treclaim.*** or a conditional or unconditional *Abort* instruction is guaranteed not to load a speculative storage update.

Engineering Note

In order to preserve the appearance of precise failures, failure handling should occur no later than the following events that would allow such delay in failure handling to be detected for the precise causes of failure

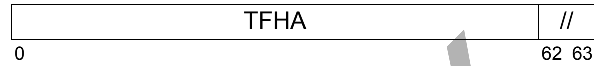
- An abort caused by the execution of ***tabort., tabortdc., tabortdci., tabortwc., tabortwci.***
- The execution of ***treclaim.***
- An execution of ***tsr., tend.***, , ***rfid***, ***hrfid***, or ***mtmsrd***, that change Transactional state
- Interrupts

8.4 Transactional Memory Facility Registers

The architecture is augmented with three Special Purpose Registers in support of transactional memory. TFHAR stores the effective address of the software failure handler used in the event of transaction failure. TFIAR is used to inform software of the exact location of the transaction failure, when possible. TEXASR contains a transaction level indicating the nesting depth of an active transaction, as well as an indicator of the cause of transaction failure and some machine state when the transaction failed. These registers can be accessed only when MSR_{TM}=1 and either $HFSCR_{TM}$=1 or the processor is in hypervisor state, (see Book III-S), and can be written only when also in Non-transactional state.

8.4.1 Transaction Failure Handler Address Register (TFHAR)

The Transaction Failure Handler Address Register is a 64-bit SPR that records the effective address of a software failure handler used in the event of transaction failure. Bits 62:63 are reserved.



This register is written with the NIA for the ***tbegin.*** as a side-effect of the execution of an outer ***tbegin.*** instruction (***tbegin.*** executed in the Non-transactional state).

8.4.2 Transaction EXception And Summary Register (TEXASR)

The Transaction EXception And Summary Register is a 64-bit register, containing a transaction level ($TEXASR_{TL}$) and summary information for use by transaction failure handlers. Bits 0:31 are called the *failure cause* in the instruction descriptions.

Bit(s Description

0:6 ***Failure Code***
The Failure Code is copied from the ***tabort.*** or ***treclaim.*** source operand. When set, TFIAR is exact.

7 ***Failure Persistent***
The failure is likely to recur on each execution of the transaction. This bit is a hint. It is set to 1 for causes in bits 8:11, copied from the ***tabort.*** or ***treclaim.*** source operand when RA is nonzero, and set to 0 for all other failure causes.

Programming Note

The Failure Persistent bit may be viewed as an eighth bit in the failure code in that both fields are supplied by the least significant byte of RA and software may use all eight to differentiate among the cases for which it performs an abort or reclaim. However, software is expected to organize its cases so that bit 7 predicts the persistence of the case.

8 ***Disallowed***
The instruction, SPR, or access type is not permitted. When set, TFIAR is exact. <cross-ref to 1.3.1 list of inst>

PRELIMINARY

Programming Note

An instruction fetch to storage that is Caching Inhibited, while nominally disallowed, will be reported as Implementation-specific (bit 15). This choice was made because it seems like a relatively unlikely programming error, and there is a significant chance that data from an external conflict (store by another thread) could indirectly cause a wild branch to storage that is Caching Inhibited.

9 ***Nesting Overflow***
The maximum transaction level was exceeded. When set, TFIAR is exact.

10 ***Footprint Overflow***
The tracking limit for transactional storage accesses was exceeded. When set, TFIAR is an approximation.

Programming Note

Note that transactional footprint tracking resources may be shared by multiple programs executing concurrently. Depending on the circumstances, this failure cause may or may not be persistent.

11 ***Self-Induced Conflict***
A self-induced conflict occurred in Suspended state, due to one of the following: a store to a block that was previously accessed transactionally; a ***dcbf***, ***dcbi,*** or ***icbi*** specifying a block that was previously accessed transactionally; a ***dcbst*** specifying a block that was previously written transactionally; or a ***tlbie*** that specifies a translation that was previously used transactionally. When set, TFIAR may be exact.

12 ***Non-Transactional Conflict***
A conflict occurred with a non-transactional access by another processor. When set, TFIAR is an approximation.

13 ***Transaction Conflict***
A conflict occurred with another transaction. When set, TFIAR may be exact.

14 ***Translation Invalidation Conflict***
A conflict occurred with a TLB invalidation. When set, TFIAR is an approximation.

15 ***Implementation-specific***
An implementation-specific condition caused the transaction to fail. Such conditions are transient and the value in the TFIAR may be exact.

16 ***Instruction Fetch Conflict***
An instruction fetch (by this or another thread) was performed from a block that was previously written transactionally. Such conditions are transient and the value in the TFIAR may be exact.

17-30 Reserved for future failure causes

31 ***Abort***
An abort was caused by the execution of a ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.*** or ***treclaim.*** instruction. When due to ***tabort.*** or ***treclaim.***, bits in $TEXASR_{0:7}$ are user-supplied. When set, TFIAR is exact.

32 ***Suspended***
When set to 1, the failure was recorded in Suspended state. When set to 0, the failure was recorded in Transactional state.

33 Reserved

34:35 ***Privilege***
The thread was in this privilege state (HV||PR) when the failure was recorded.

Programming Note

When looking at the state of the bits above, note that the transaction may fail in one state but the failure may be recorded in another if the failure cause is imprecise. An access that causes footprint overflow and is immediately followed by a ***tsuspend.*** is an example of when such a scenario is likely.

36 ***Failure Summary (FS)***
Set to 1 when a failure has been detected and failure recording has been performed.

37 ***TFIAR Exact***
Set to 1 when the value in the TFIAR is exact. Otherwise the value in the TFIAR is approximate.

38 ***ROT***
Set to 1 when a ROT is initiated. Set to zero when a non-ROT ***tbegin.*** is executed.

39 Reserved

40:51 Reserved

52:63 ***Transaction Level (TL)***
Transaction level (nesting depth + 1) for the active transaction, if any; otherwise 0 if the most recently executed transaction completed successfully, or the transaction level at which the most recently executed transaction failed if the most recently executed transaction did not complete successfully.

PRELIMINARY

Programming Note

A value of 1 corresponds to an outer transaction. A value greater than 1 corresponds to a nested transaction.

The transaction level in $TEXASR_{TL}$ contains an unsigned integer indicating whether the current transaction is an outer transaction, or is nested, and if nested, its depth. The maximum transaction level supported by a given implementation is of the form $2^t - 1$. The value of *t* corresponding to the smallest maximum is 4; the value of *t* corresponding to the largest maximum is 12. This value is tied to the “Maximum transaction level” parameter useful for application programmers, as specified in Section 4.1. The high-order 12-*t* bits of $TEXASR_{TL}$ are treated as reserved.

Transaction failure information is contained in $TEXASR_{0:37}$. The fields of TEXASR are initialized upon the successful initiation of a transaction from the Non-transactional state, by setting $TEXASR_{TL}$ to 1, indicating an outer transaction, and all other fields to 0.

When transaction failure is recorded, the failure summary bit $TEXASR_{FS}$ is set to 1, indicating that failure has been detected for the active transaction and that failure recording has been performed. $TEXASR_{0:31}$ are set indicating the source of the failure. Exactly one of bits 8 through 31 will be set indicating the instruction or event that caused failure. In the event of failure due to the execution of a ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.*** or ***treclaim.*** instruction, $TEXASR_{31}$ is set to 1, and, for ***tabort.*** and ***treclaim.***, a software defined failure code is copied from a register operand to $TEXASR_{0:7}$. $TEXASR_{Suspended}$ indicates whether the transaction was in the Suspended state at the time that failure occurred. The value of MSR_{HV} and MSR_{PR} at the time that failure occurs are copied to $TEXASR_{34}$ and $TEXASR_{35}$, respectively. In some circumstances, the failure causing instruction address in TFIAR may not be exact. In such circumstances, $TEXASR_{37}$ is set to 0 indicating that the contents of TFIAR are not exact; otherwise $TEXASR_{37}$ is set to 1.

Programming Note

The transaction level contained in $TEXASR_{TL}$ should be interpreted by software as follows:

When in the Transactional or Suspended state, this field contains an unsigned integer representing the transaction level of the active transaction, with 1 indicating an outer transaction, and a number greater than 1 indicating a nested transaction. The nesting depth of the active transaction is $TEXASR_{TL} – 1$.

When in the Non-transactional state, $TEXASR_{TL}$ contains 0 if the last transaction committed successfully, otherwise it contains the transaction level at which the most recent transaction failed.

Programming Note

The Privilege bits in TEXASR represent the state of the machine at the point when failure occurs. This information may be used by problem-state software to determine whether an unexpected hypervisor or operating system interaction was responsible for transaction failure. This information may be useful to operating systems or hypervisors when restoring register state for failure handling after the transactional facility was reclaimed, to determine which of the operating system or the hypervisor has retained the pre-transactional version of the speculative registers.

Engineering Note

If multiple failure causes occur at the same time, hardware must choose one cause to report. In general, it will be most beneficial to report a persistent cause when both types occur simultaneously.

8.4.3 Transaction Failure Instruction Address Register (TFIAR)

The Transaction Failure Instruction Address Register is a 64-bit SPR that is set to the exact effective address of the instruction causing the failure, when possible. Bits 62:63 contain $MSR_{HV\ PR}$ at the time of the failure..

TFIA	HV	PR
0	62	63

In certain cases, the exact address may not be available, and therefore TFIAR will be an approximation. An approximate value will point to an instruction near the instruction that was executing at the time of the failure. TFIAR accuracy is recorded in an Exact bit residing in $TEXASR_{37}$.

8.5 Transactional Facility Instructions

The *Transactional Memory* instructions may only be executed when MSR_{TM}=1 and either $HFSCR_{TM}$=1 or the processor is in hypervisor state (see Book III-S).

Similar to the *Floating-Point Status and Control Register* instructions, modifications of transaction state caused by the execution of *Transactional Memory* instructions or by failure handling synchronize the effects of exception-causing floating-point instructions executed by a given processor. Executing a Transactinal Memory instruction, or invocation of the failure handler, ensures that all floating-point instructions previously initiated by the given processor have completed before the transaction state is modified, and that no subsequent floating-point instructions are initiated

PRELIMINARY

by the given processor until the transaction state has been modified. In particular:

- All exceptions that will be caused by the previously initiated instructions are recorded in the FPSCR before the transaction state is modified.
- All invocations of the system floating-point enabled exception error handler that will be caused by the previously initiated instructions have occurred before the transaction state is modified.
- No subsequent floating-point instruction that alters the settings of any FPSCR bits is initiated until the transaction state has been modified.

(Floating-point Storage Access instructions are not affected.)

Transaction Begin *X-form*

tbegin. R

31	A	//	R	///	///	654	1
0	6	7	10	11	16	21	31

```
ROT  <- R
CR0  <- 0 || MSR_TS || 0

if MSR_TS = 0b00 then                 #Non-transactional
   TEXASR <- 0x000000000 || 0b00 || ROT || 0b0 ||
0x000001
   TFHAR <- CIA + 4
   TDOOMED <- 0
   MSR_TS <- 0b10
   save registers to speculative register check-
point
   if not ROT then
     enforce_barrier(mbll)
     enforce_barrier(mbls)
     enforce_barrier(mbsl)
     enforce_barrier(mbss)
else if MSR_TS = 0b10 then            #Transactional
   if TEXASR_TL=TL_max then
      cause <- 0x00400000
      TMRecordFailure(cause)
      TMHandleFailure()
   else
      TEXASR_TL <- TEXASR_TL + 1
      if (TEXASR_ROT=1) & (not ROT)
         enforce_barrier(mbll)
         enforce_barrier(mbls)
         enforce_barrier(mbsl)
         enforce_barrier(mbss)
         TEXASR_ROT <- 0
```

The ***tbegin.*** instruction initiates execution of a transaction, either an outer transaction or a nested transaction, as described below.

An outer transaction is initiated when ***tbegin.*** is executed in the Non-transactional state. If R=0, a barrier is inserted equivalent to that produced by a ***sync*** instruction with E=0b1111. (See <crossref to sync description>.) TEXASR and TFHAR are initialized, and the TDOOMED bit is set to 0. A nested transaction is initiated when ***tbegin.*** is executed in the Transactional state unless the transaction level is already at its maximum value, in which case failure recording is performed with a failure cause of 0x00400000 and failure handling is performed. When initiating a nested transaction, the transaction level held in $TEXASR_{TL}$ is incremented by 1, and if $TEXASR_{ROT}$ =1 but R=0, a barrier is inserted equivalent to that produced by a ***sync*** instruction with E=0b1111 and $TEXASR_{ROT}$ is turned off. The effects of a nested transaction will not be visible until the outer transaction commits, and in the event of failure, speculative registers are reverted to the pre-transactional value of the outer transaction. Initiation of a transaction is unsuccessful when in the Suspended state.

When successfully initiated, transactional execution continues until the transaction is terminated using a ***tend.***, ***tabort.***, ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, ***tabortwci.***, or ***treclaim.*** instruction, suspended using a ***tsr*** instruction, or failure occurs. Upon transaction failure while in the Transactional state, transaction failure recording and failure handling are performed as defined in Section 8.3. Upon transaction failure while in the Suspended state, failure recording is performed as defined in Section 8.3.2, but failure handling is usually deferred.

CR0 is set as follows.

CR0	*Description*
000 || 0	Transaction initiation successful, unnested (Transaction state of Non-transactional prior to ***tbegin.***)
010 || 0	Transaction initiation successful, nested (Transaction state of Transactional prior to ***tbegin.***)
001 || 0	Transaction initiation unsuccessful, (Transaction state of Suspended prior to ***tbegin.***)

Other than the setting of CR0, ***tbegin.*** in the Suspended state is treated as a no-op.

The use of the A field is implementation specific.

Special Registers Altered

CR0 TEXASR TFHAR TS

PRELIMINARY

Programming Note

When a transaction is successfully initiated, and failure subsequently occurs, control flow will be redirected to the instruction following the ***tbegin.*** instruction. When failure handling occurs, as described in Section 8.3.3, CR0 is set to 0b101 || 0. Consequently, instructions following ***tbegin.*** should also expect this value as an indication of transaction failure. Most applications will follow ***tbegin.*** with a conditional branch predicated on $CR0_2$; code at this target is responsible for handling the transaction failure.

Transaction End X-form

tend. A

31	A	//	/	///	///	686	1
0	6	7	10	11	16	21	31

```
CR0  <- 0b0 || MSR_TS || 0

if MSR_TS = 0b10 then            #Transactional
   if A = 1 | TEXASR_TL = 1 then
      if (TDOOMED) then
         TMHandleFailure()
      else
         if not TEXASR_ROT
           insert integrated cumulative barrier
         Commit transaction
         TEXASR_TL <- 0
          Discard all resources related to current
transaction
          MSR_TS <- 0b00          #Non-transactional
          if not TEXASR_ROT
           enforce_barrier(mbll)
           enforce_barrier(mbls)
           enforce_barrier(mbsl)
           enforce_barrier(mbss)
   else TEXASR_TL <- TEXASR_TL - 1  # nested
```

The A=0 variant of ***tend.*** supports nested transactions, in which the transaction is committed only if the execution of ***tend.*** completes an outer transaction. Execution of this variant by a nested transaction ($TEXASR_{TL}$ > 1) causes $TEXASR_{TL}$ to be decremented by 1. The A=1 variant of ***tend.*** unconditionally completes the current outer transaction and all nested transactions.

When the ***tend.*** instruction completes an outer transaction, transaction commit is predicated on the TDOOMED bit. If TDOOMED is 1, failure handling occurs as defined in Section 8.3.3. If TDOOMED is 0, the transaction is committed, and $TEXASR_{TL}$ is set to 0. In both cases, the transaction state is set to Non-transactional.

When the ***tend.*** instruction commits a transaction, it atomically commits its writes to storage. If $TEXASR_{ROT}$=0, the integrated cumulative barrier is inserted prior to the creation of the aggregate store, and a barrier is inserted equivalent to that produced by a sync instruction with E=0b1111 after the aggregate store. (See <crossref to sync description>.) If the transaction has failed prior to the execution of ***tend.*** no storage updates are performed and no barrier is inserted. In either case (success or failure), all resources associated with the transaction are discarded.

If the transaction succeeds, Condition Register field 0 is set to 0 || MSR_{TS} || 0.If the transaction fails, CR0 is set to 0b101 || 0.

Other than the setting of CR0, ***tend.*** in Non-transactional state is treated as a no-op. If an attempt is made to execute ***tend.*** in Suspended state, a TM Bad Thing type Program interrupt occurs.

Special Registers Altered
CR0 TEXASR TS

Extended Mnemonics

Extended mnemonics for transaction end.

Extended:	***Equivalent To:***
tend.	***tend.*** 0
tendall.	***tend.*** 1

Programming Note

When an outer ***tend.*** or a ***tend.*** with A=1 is executed in the Transactional state, the CR0 value 0b101 || 0 will never be visible to the instruction that immediately follows ***tend.***, because in the event of failure the failure handler will have been invoked not later than the completion of the ***tend.*** instruction.

Transaction Abort X-form

tabort. RA

31	///	RA	///	910	1
0	6	11	16	21	31

```
CR0 <-  0 || MSR_TS || 0

if MSR_TS = 0b10 | MSR_TS = 0b01 then
#Transactional, or Suspended
   if RA = 0 then cause <-0x00000001
   else          cause <- GPR(RA)_56:63 || 0x000001
   if MSR_TS= 0b01 & TEXASR_FS = 0 then  #Suspended
      Discard speculative storage updates
   TMRecordFailure(cause)
   if MSR_TS = 0b10 then                  #Transactional
      TMHandleFailure()
```

PRELIMINARY

The ***tabort.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. When in the Transactional state or the Suspended state the ***tabort.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 8.3.2. If RA is 0, the failure cause is set to 0x00000001, otherwise it is set to $GPR(RA)_{56:63}$ || 0x000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 8.3.3 (this includes discarding the set of transactional storage updates).

If the transaction state is Suspended, the set of transactional storage updates is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabort.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered
CR0 TEXASR TFIAR TS

Transaction Abort Word Conditional X-form

tabortwc. TO,RA,RB

31	TO	RA	RB	782	1
0	6	11	16	21	31

```
a <- EXTS((RA)32:63)
b <- EXTS((RB)32:63)
abort <- 0

CR0 <- 0 || MSRTS || 0

if (a < b)  & TO0 then abort <- 1
if (a > b)  & TO1 then abort <- 1
if (a = b)  & TO2 then abort <- 1
if (a u< b) & TO3 then abort <- 1
if (a >u b) & TO4 then abort <- 1

if abort & (MSRTS = 0b10 | MSRTS = 0b01) then
            #Transactional or Suspended
   cause <- 0x00000001
   if MSRTS= 0b01 & TEXASRFS = 0 then  #Suspended
      Discard speculative storage updates
   TMRecordFailure(cause)
   if MSRTS = 0b10 then               #Transactional
      TMHandleFailure()
```

The ***tabortwc.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register $RA_{32:63}$ are compared with the contents of register $RB_{32:63}$. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended, then the ***tabortwc.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 8.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 8.3.3 (this includes discarding the set of transactional storage updates).

If the transaction state is Suspended, the set of transactional storage updates is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortwc.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered
CR0 TEXASR TFIAR TS

Transaction Abort Word Conditional Immediate X-form

tabortwci. TO,RA,SI

31	TO	RA	SI	846	1
0	6	11	16	21	31

```
a <- EXTS((RA)32:63)
abort <- 0

CR0 <- 0 || MSRTS || 0

if a < EXTS(SI)  & TO0 then abort <- 1
if a > EXTS(SI)  & TO1 then abort <- 1
if a = EXTS(SI)  & TO2 then abort <- 1
if a u< EXTS(SI) & TO3 then abort <- 1
if a >u EXTS(SI) & TO4 then abort <- 1

if abort & (MSRTS = 0b10 | MSRTS = 0b01) then
            #Transactional or Suspended
   cause <- 0x00000001
   if MSRTS= 0b01 & TEXASRFS = 0 then  #Suspended
      Discard speculative storage updates
   TMRecordFailure(cause)
   if MSRTS = 0b10 then               #Transactional
      TMHandleFailure()
```

The ***tabortwci.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register $RA_{32:63}$ are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended then the ***tabortwci.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 8.3.2, using the failure cause 0x00000001.

PRELIMINARY

If the transaction state is Transactional, failure handling is performed as defined in Section 8.3.3 (this includes discarding the set of transactional storage updates).

If the transaction state is Suspended, the set of transactional storage updates is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortwci.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered

CR0 TEXASR TFIAR TS

Transaction Abort Doubleword Conditional X-form

tabortdc. TO,RA,RB

31	TO	RA	RB	814	1
0	6	11	16	21	31

```
a <- ( RA )
b <- ( RB )
abort <- 0

CR0 <-  0 || MSR_TS || 0

if (a < b)  & TO_0 then abort <- 1
if (a > b)  & TO_1 then abort <- 1
if (a = b)  & TO_2 then abort <- 1
if (a u< b) & TO_3 then abort <- 1
if (a >u b) & TO_4 then abort <- 1

if abort & (MSR_TS = 0b10 | MSR_TS = 0b01) then
           #Transactional or Suspended
   cause <- 0x00000001
   if MSR_TS= 0b01 & TEXASR_FS = 0 then  #Suspended
      Discard speculative storage updates
   TMRecordFailure(cause)
   if MSR_TS = 0b10 then                #Transactional
      TMHandleFailure()
```

The ***tabortdc.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register RA are compared with the contents of register RB. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended, then the ***tabortdc.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 8.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 8.3.3 (this includes discarding the set of transactional storage updates).

If the transaction state is Suspended, the set of transactional storage updates is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortdc.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered

CR0 TEXASR TFIAR TS

Transaction Abort Doubleword Conditional Immediate X-form

tabortdci. TO,RA, SI

31	TO	RA	SI	878	1
0	6	11	16	21	31

```
a <- (RA)
abort <- 0

CR0 <-  0 || MSR_TS || 0

if a < EXTS(SI)  & TO_0 then abort <- 1
if a > EXTS(SI)  & TO_1 then abort <- 1
if a = EXTS(SI)  & TO_2 then abort <- 1
if a u< EXTS(SI) & TO_3 then abort <- 1
if a >u EXTS(SI) & TO_4 then abort <- 1

if abort & (MSR_TS = 0b10 | MSR_TS = 0b01) then
           #Transactional or Suspended
   cause <- 0x00000001
   if MSR_TS= 0b01 & TEXASR_FS = 0 then  #Suspended
      Discard speculative storage updates
   TMRecordFailure(cause)
   if MSR_TS = 0b10 then                #Transactional
      TMHandleFailure()
```

The ***tabortdci.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. The contents of register RA are compared with the sign-extended value of the SI field. If any bit in the TO field is set to 1 and its corresponding condition is met by the result of the comparison, and the transaction state is Transactional or Suspended then the ***tabortdci.*** instruction causes transaction failure, resulting in the following:

Failure recording is performed as defined in Section 8.3.2, using the failure cause 0x00000001.

If the transaction state is Transactional, failure handling is performed as defined in Section 8.3.3 (this includes discarding the set of transactional storage updates).

If the transaction state is Suspended, the set of transactional storage updates is discarded (if not already discarded for a pending failure), but failure handling is deferred.

Other than the setting of CR0, execution of ***tabortdci.*** in the Non-transactional state is treated as a no-op.

PRELIMINARY

Special Registers Altered
CR0 TEXASR TFIAR TS

Transaction Suspend or Resume X-form

tsr. L

31	///	L	///	///	750	1
0	6	10	11	16	21	31

```
CR0 <-  0 || MSR_TS || 0
if L = 0 then
   if MSR_TS = 0b10 then                #Transactional
      MSR_TS <- 0b01                     #Suspended
else
   if MSR_TS = 0b01                      #Suspended
      MSR_TS <- 0b10                    #Transactional
```

The ***tsr.*** instruction sets condition register field 0 to 0 || MSR_{TS} || 0. Based on the value of the L field, two variants of ***tsr.*** are used to change the transaction state.

If L = 0, and the transaction state is Transactional, the transaction state is set to Suspended.

If L = 1, and the transaction state is Suspended, the transaction state is set to Transactional.

Other than the setting of CR0, the execution of ***tsr.*** in the Non-transactional state is treated as a no-op.

Special Registers Altered
CR0 TS

Programming Note

When resuming a transaction that has encountered failure while in the Suspended state, failure handling is performed after the execution of ***tresume*** and no later than transaction completion due to the execution of one of the following instructions: an outer ***tend.*** or ***tend.*** with A=1, ***tabort.***, a ***tabortdc.***, ***tabortdci.***, ***tabortwc.***, or ***tabortwci.*** that causes the transaction to be aborted, or ***treclaim.***.

Extended Mnemonics

Extended mnemonics for Transaction Suspend or Resume.

Extended:	***Equivalent To:***
tsuspend.	***tsr***. 0
tresume.	***tsr***. 1

Transaction Check X-form

tcheck BF

31	BF	//	///	///	718	/
0	6	9	11	16	21	31

```
if MSR_TS = 0b10 | MSR_TS = 0b01 then   #transactional
                                         #or suspended
  for each load caused by an instruction following
  the outer tbegin and preceding this tcheck
    if (Load instruction was executed in T state
       with TEXASR_ROT=0 or accessing a location
       previously stored transactionally) |
       (Load instruction was executed in S state
        with TEXASR_ROT=0 and accessed a location
        previously accessed transactionally)|
       (Load instruction was executed in S state
        with TEXASR_ROT=1 and accessed a location
        previously stored transactionally)
    then wait until load has been performed with
    respect to all processors and mechanisms
CR field BF <-  TDOOMED || MSR_TS || 0
```

If the transaction state is Transactional or Suspended, the ***tcheck*** instruction ensures that all loads, that are caused by instructions that follow the outer ***tbegin.*** instruction and precede the ***tcheck*** instruction and satisfy one of the following properties, have been performed with respect to all processors and mechanisms.

- The load is caused by an instruction that was executed in Transactional state, either while $TEXASR_{ROT}$=0 or accessing a location previously stored transactionally.
- The load is caused by an instruction that was executed in Suspended state while $TEXASR_{ROT}$=0 and accesses a location that was accessed transactionally.
- The load is caused by an instruction that was executed in Suspended state while $TEXASR_{ROT}$=1 and accesses a location that was stored transactionally.

The ***tcheck*** instruction then copies the TDOOMED bit into bit 0 of CR field BF, copies MSR_{TS} to bits 1:2 of CR field BF, and sets bit 3 of CR field BF to 0.

Other than the setting of CR field BF, execution of ***tcheck*** in the Non-transactional state is treated as a no-op.

Special Registers Altered
CR field BF

PRELIMINARY

Programming Note

One use of the ***tcheck*** instruction in Suspended state is to determine whether preceding loads from transactionally modified locations have returned the data the transaction stored. (If the transaction has failed, some of the loads may have returned a more recent value that was stored by a conflicting store, or may have returned the pre-transaction contents of the location.).

Another use of ***tcheck*** in Suspended state is to determine whether the contents of storage, as seen in Suspended state, are consistent with the transaction succeeding -- e.g., whether no location that has been accessed transactionally (stored transactionally, for ROTs), and has been seen in Suspended state, has been subject of a conflict thus far. (A location is seen in Suspended state either by being loaded in Suspended state or by being loaded in Transactional state and the value (or a value derived therefrom) passed, in a register, into Suspended state.)

A use of ***tcheck*** in Transactional state is to determine whether the transaction still has the potential to succeed.

Note that ***tcheck*** provides an instantaneous check on the integrity of a subset of the accesses performed within a transaction. ***tcheck*** is not a failure synchronizing mechanism. Even if no accesses follow the ***tcheck***, there may still be latent failures that haven't been recorded, for example caused by accesses that ***tcheck*** does not wait for, by external conflicts that will happen in the future, or simply by time of flight to the failure detection mechanism for operations that have already been performed.

Programming Note

The ***tcheck*** instruction can return 1 in bit 0 of CR field BF before the failure has been recorded in TEXASR and TFIAR.

Programming Note

The ***tcheck*** instruction may cause pipeline synchronization. As a result, programs that use ***tcheck*** excessively may perform poorly.

---------------------------- End text --------------------------------

Section B.3+ Transactional Lock Elision [Category: Transactional Memory]

Add a new section after B.3 describing Transactional Lock Elision examples.

------------------------- Begin text --------------------------------

8.6 Transactional Lock Elision

This section illustrates the use of the Transactional Memory facility to implement transactional lock elision (TLE), in which lock-based critical sections are speculatively executed as a transaction without first acquiring a lock. This locking protocol is an alternative to the routines described above, yielding increased concurrency when the lock that guards a critical section is frequently unnecessary.

8.6.1 Enter Critical Section

The following example shows the entry point to a critical section using transactional lock elision. The entry code starts a transaction using the ***tbegin.*** instruction and checks whether the transaction was aborted or not. If not, it checks whether the lock is free or not. If the lock is found to be free, the thread proceeds to execute the critical section.

In this example it is assumed that the address of the lock is in GPR 3, and the value indicating that the lock is free is in GPR 4. The handling of cases of transaction abort and busy lock are described in subsequent examples.

```
tle_entry:
   tbegin.              #Start TLE transaction
   beq- tle_abort       #Handle TLE transaction abort
   lwz r6,0(r3)         #Read lock
   cmpw r6,r4           #Check if lock is free
   bne- busy_lock       #If not, handle lock busy case

critical_section1:
```

8.6.2 Handling Busy Lock

In the event that the lock is already held, by either another thread or the current thread, the transaction is aborted using the ***tabort*** instruction, using a software-defined code *TLE_BUSY_LOCK* indicating the cause of the abort. The abort returns control to the ***beq*** following ***tbegin.*** in the critical section entrance sequence, allowing for an abort handler to react appropriately.

```
busy_lock:
   li r3, TLE_BUSY_LOCK
   tabort r3              #Abort TLE transaction
```

8.6.3 Handling TLE Abort

A TLE transaction may fail for one of a variety of causes, persistent and transient. Persistent causes are certain—or at least highly likely—to cause future attempts to execute the same transaction to fail. How-

PRELIMINARY

ever, for transient causes, it is possible that the failure cause may not be re-encountered in a subsequent attempt. Thus, persistent aborts are handled by taking a non-transactional path that involves the actual acquisition of the lock, while transient aborts retry the critical section using TLE.

The following example illustrates the handling of aborts in TLE. It is assumed that the address of the lock is in GPR 3. The immediate value of the ***andis.*** instruction selects the Failure Persistent bit in the upper half of TEXASR to be tested.

```
tle_abort:
   mfspr r4, TEXASRU    # Read high-order half
                        # of TEXASR
   andis. r5,r4,0x0100  # determine whether failure
                       # is likely to be persistent
   bne tle_acquire_lock   #Persistent, acquire lock
                          #enter critical sec
   b tle_entry            #Transient, try TLE again
```

This example can be extended to keep track of the number of transient aborts and fall back on the acquisition of the lock after the number of transient failures reaches some threshold. It can also be extended to handle reentrant locks. Acquisition of TLE locks is described in a subsequent example.

8.6.4 TLE Exit Section Critical Path

The following example illustrates the instruction sequence used to exit a TLE critical section. The CR0 value set by ***tend.*** indicates whether the current thread was in a transaction. If so, the exited critical section was entered speculatively, and the transaction is ended. If not, the execution takes a path to release the lock.

Release of an acquired TLE lock is described in a subsequent example.

```
tle_exit:
   tend.                     #End the current trans-
                            #action, if any
   bng- tle_release_lock    #Release lock, if was
                            #not in a transaction
```

8.6.5 Acquisition and Release of TLE Locks

The steps for acquiring and releasing a lock associated with a TLE critical section are nearly identical to those for acquiring and releasing conventional locks that are not elided, as described in Section B.2. The only difference is special care must be taken to prevent loads and stores inside the critical section protected by the lock from being performed before the *Store Conditional* instruction to the lock variable.

The ***isync*** from the *Acquire Lock and Import Shared Storage* sequence described in Section B.2.1.1 is insufficient for ordering subsequent instructions. Instead, a stronger storage access ordering instruction is needed. The following example shows the resulting steps for acquiring a TLE lock, replacing the *isync* instruction with a *sync* instruction. In this example it is assumed that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5.

```
tle_acquire_lock:
   lwarx r10,0,r3,1         #Load lock and reserve
   cmpw r4,r10              #Skip ahead if
   bne- wait                #Lock not free
   stwcx. r5,0,r3           #Try to set lock
   bne- tle_acquire_lock  #Loop if lost reservation
   sync                     #Import barrier for TLE

critical_section1:
```

The instruction sequence necessary for the release of a TLE lock is identical to the conventional lock release sequence, where ***lwsync*** is sufficient to prevent the store that releases the lock from being performed before the loads and stores in the critical section protected by the lock.

---------------------------- End text --------------------------------

Book 3-S:

Section 2.6 Processor Compatibility Register

Update Figure 5, adding bit 2 as “TM”. Add the following description for bit 2 to the description of the PCR.

-------------------------- Begin text ---------------------------------

Bit	Description
2	**Transactional Memory** (TM) [Category: Transactional Memory] This bit controls the availability, in problem state, of the instructions and facilities in the Transactional Memory category as it was defined in the latest version of the architecture for which new problem-state resources are made available; if the Transactional Memory category was not defined in that version of the architecture, then Transactional Memory instructions and facilities are unavailable. 0 The instructions and facilities in the Transactional Memory category are available in problem state.

PRELIMINARY

1 The instructions and facilities in the Transactional Memory category are unavailable in problem state.

---------------------------- End text --------------------------------

Section 3.2.1 Machine State Register

The MSR is augmented with a new two bit Transaction State field (TS) from the set of available full-function range bits. These bits encode three transaction states described in Book II, section 8.2. Changes to MSR[TS] that are caused by Transactional Memory instructions, and by invocation of the transaction's failure handler, take effect immediately (even though these instructions and events are not context synchronizing).

Add the following descriptions for bits 29-31 and associated programming note to the descxription of the MSR.

-------------------------- Begin text --------------------------------

Bit	Description
29:30	***Transaction State*** (TS) [Category: Transactional Memory] 00 Non-transactional 01 Suspended 10 Transactional 11 Reserved Changes to MSR[TS] that are caused by Transactional Memory instructions, and by invocation of the transaction's failure handler, take effect immediately (even though these instructions and events are not context synchronizing).
31	***Transactional Memory Available*** (TM) [Category: Transactional Memory] **0** The thread cannot execute any Transactional Memory instructions or access any Transactional Memory registers. **1** The thread can execute Transactional Memory instructions and access Transactional Memory registers.

Programming Note

To access Transactional Memory registers and execute Transactional Memory instructions, it must also be true that $HFSCR_{TM}$=1 or the processor is in hypervisor state. See <cross ref to HFSCR description> for more information.

----------------------------End text --------------------------------

Exclude instructions that are forbidden in Transactional mode from being traced

-------------------------- Begin text --------------------------------

Bit	Description
53	***Single-Step Trace Enable*** (SE) [Category: Trace] 0 The thread executes instructions normally. 1 The thread generates a Single-Step type Trace interrupt after successfully completing the execution of the next instruction, unless that instruction is ***hrfid*** or ***rfid***, which are never traced. Successful completion means that the instruction caused no other interrupt and, if the processor is in the Transactional state <TM>, is not one of the instructions that is forbidden in Transactional state (e.g., ***dcbf***; see <crossref to failure cause section>).

----------------------------End text --------------------------------

Section 3.2.1+ State Transitions Associated with the Transactional Memory Facility [Category: Transactional Memory]

Add the following section after Section 3.2.1, giving details about MSR bit state transitions.

-------------------------- Begin text --------------------------------

Updates to MSR_{TS} and MSR_{TM} caused by , ***rfid***, ***hrfid***, or ***mtmsrd*** occur as described in Table 2. The value written, and whether or not the instruction causes an interrupt, are dependent on the current values of MSR_{TS} and MSR_{TM}, and the values being written to these fields. When the setting of MSR_{TS} causes an illegal state transition, a TM Bad Thing type Program interrupt is generated.

Programming Note

The transition rules are the same for ***mtmsrd*** as for the ***rfid***-type instructions because if a transition were illegal for ***mtmsrd*** but allowed for ***rfid***, or vice versa, software could use the instruction for which the transition is allowed to achieve the effect of the other instruction.

PRELIMINARY

Table 2 shows all the Transaction State transitions that can be requested by , ***rfid***, ***hrfid***, and ***mtmsrd***. The table covers behavior when TM is enabled by the PCR. For causes of the TM Bad Thing exception when TM is disabled by the PCR, see <crossref to TM Bad Thing>. In the table, the contents of MSR_{TS} and MSR_{TM} are abbreviated in the form AB, where A represents MSR_{TS} (N, T or S) and B represents MSR_{TM} (0 or 1). “x” in the “B” position means that the entry covers both MSR_{TM} values, with the same value applying in all columns of a given row for a given instance of the transition. (E.g., the first row means that the transition from N0 to N0 is allowed and results in N0, and that the transition from N0 to N1 is allowed and results in N1.) “Input $MSR_{TS}MSR_{TM}$” in the second column refers to the MSR_{TS} and MSR_{TM} values supplied by SRR1 for ***rfid***, HSRR1 for ***hrfid***, or register RS for ***mtmsrd***.

Current $MSR_{TS}MSR_{TM}$	Input $MSR_{TS}MSR_{TM}$	Resulting MSR_{TS} MSR_{TM}	Comments
N0	Nx	Nx	May occur in the context of a Transactional Memory type of Facility Unavailable interrupt handler, enabling/disabling transactions for user-level applications.
	All others - Illegal[1]	N0	
T0	N/A		Unreachable state
S0	N0[2]	S0	Operating system code that is not TM aware may attempt to set TS and TM to zero, thinking they’re reserved bits. Change is suppressed.
	T1	T1	May occur at an ***rfid*** returning to an application whose transaction was suspended on interrupt.
	Sx	Sx	This case may occur for an ***rfid*** returning to an application whose suspended transaction was interrupted.
	All others - Illegal[1]	S0	
N1	Nx	Nx	After a ***treclaim***, the OS dispatches Nx program.
	All others -IIllegal[1]	N0	
T1	All	N1	Disallowed instructions in Transactional state.
S1	T1	T1	May occur after ***trechkpt.*** when returning to an application.
	Sx	Sx	
	All others - Illegal[1]	S0	

Notes:
1. Generate TM Bad Thing type Program interrupt. “All others" includes all attempts to set MSR_{TS} to 0b11 (reserved value).
2. Instruction completes, change to MSR_{TM} suppressed,

Table 2: Transaction state transitions that can be requested by , ***rfid***, ***hrfid***, and ***mtmsrd***.

PRELIMINARY

Programming Note

For *[h]rfid*, and *mtmsrd*, the attempted transition from S0 to N0 is suppressed in order that interrupt handlers that are "unaware" of transactional memory, and load an MSR value that has not been updated to take account of transactional memory, will continue to work correctly. (If the interrupt occurs when a transaction is running or suspended, the interrupt will set MSR[TS || TM] to S0. If the interrupt handler attempts to load an MSR value that has not been updated to take account of transactional memory, that MSR value will have TS || TM = N0. It is desirable that the interrupt handler remain in state S0, so that it can return normally to the interrupted transaction.)

--------------------------- End text -------------------------------

PRELIMINARY

Section 3.3.1 System Linkage Instructions

Change the RTL and verbal descriptions of rfid, and hrfid operation to account for the case in which the change to TS is suppressed.

------------------------- Begin text -------------------------------

Return From Interrupt Doubleword XL-form

rfid

19	///	///	///	18	/
0	6	11	16	21	31

```
MSR51 ← (MSR3 & SRR151) | ((¬MSR3) & MSR51)
MSR3 ← MSR3 & SRR13
if (MSR29:31 ¬= 0b010 | SRR129:31 ¬= 0b000) then
   MSR29:31 <- SRR129:31
MSR48 ← SRR148 | SRR149
MSR58 ← SRR158 | SRR149
MSR59 ← SRR159 | SRR149
MSR0:2 4:28 32 37:41 49:50 52:57 60:63 ← SRR10:2 4:28 32 37:41 49:50 52:57 60:63
NIA ←iea SRR00:61 || 0b00
```

If MSR_3=1 then bits 3 and 51 of SRR1 are placed into the corresponding bits of the MSR. If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of SRR1 are not equal to 0b000, then the value of bits 29 through 31 of SRR1 is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of SRR1 is placed into MSR_{48}. The result of ORing bits 58 and 49 of SRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of SRR1 is placed into MSR_{59}. Bits 0:2, 4:28, 32, 37:41, 49:50, 52:57, and 60:63 of SRR1 are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition (see Table 2, “Transaction state transitions that can be requested by , rfid, hrfid, and mtmsrd.,” on page 25) or when TM is disabled by the PCR, a transition to Problem state with an active transaction, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the ***rfid*** instruction. Otherwise, if the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $SRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or ${}^{32}0$ || $SRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

---------------------------- End text --------------------------------

------------------------- Begin text -------------------------------

PRELIMINARY

Hypervisor Return From Interrupt Doubleword *XL-form*

hrfid

19	///	///	///	274	/
0	6	11	16	21	31

```
if (MSR_{29:31} ¬= 0b010 | HSRR1_{29:31} ¬= 0b000) then
   MSR_{29:31} <- HSRR1_{29:31}
MSR_{48} ← HSRR1_{48} | HSRR1_{49}
MSR_{58} ← HSRR1_{58} | HSRR1_{49}
MSR_{59} ← HSRR1_{59} | HSRR1_{49}
MSR_{0:28 32 37:41 49:57 60:63} ← HSRR1_{0:28 32 37:41 49:57 60:63}
NIA ←_{iea} HSRR0_{0:61} || 0b00
```

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of HSRR1 are not equal to 0b000, then the value of bits 29 through 31 of HSRR1 is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of HSRR1 is placed into MSR_{48}. The result of ORing bits 58 and 49 of HSRR1 is placed into MSR_{58}. The result of ORing bits 59 and 49 of HSRR1 is placed into MSR_{59}. Bits 0:28, 32, 37:41, 49:57, and 60:63 of HSRR1 are placed into the corresponding bits of the MSR.

If the instruction attempts to cause an illegal transaction state transition (see Table 2, “Transaction state transitions that can be requested by , rfid, hrfid, and mtmsrd.,” on page 25) or when TM is disabled by the PCR, a transition to Problem state with an active transaction, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the *hrfid* instruction. Otherwise, if the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address $HSRR0_{0:61}$ || 0b00 (when SF=1 in the new MSR value) or ${}^{32}0$ || $HSRR0_{32:61}$ || 0b00 (when SF=0 in the new MSR value). If the new MSR value enables one or more pending exceptions, the interrupt associated with the highest priority pending exception is generated; in this case the value placed into SRR0 or HSRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the instruction that would have been executed next had the interrupt not occurred.

---------------------------- End text --------------------------------

Section 3.3.2 Power-Saving Mode Instructions

The verbal descriptions of each of the power saving instructions is extended to note that execution in Transactional state results in a TM Bad Thing type of Program interrupt.

-------------------------- Begin text --------------------------------

Doze *XL-form*

doze

19	///	///	///	402	/
0	6	11	16	21	31

The thread is placed into doze power-saving level.

When the thread is in doze power-saving level, the state of all thread resources is maintained as if the thread was not in power-saving mode.

When the interrupt that causes exit from doze power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

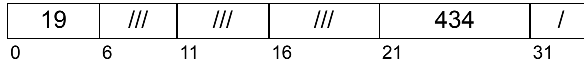
This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:
None

PRELIMINARY

Nap *XL-form*

nap



19	///	///	///	434	/
0	6	11	16	21	31

The thread is placed into nap power-saving level.

When the thread is in nap power-saving level, the state of the Decrementer and all hypervisor resources is maintained as if the thread was not in power-saving mode, and sufficient information is maintained to allow the hypervisor to resume execution.

When the interrupt that causes exit from nap power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

This instruction is hypervisor privileged and context synchronizing.

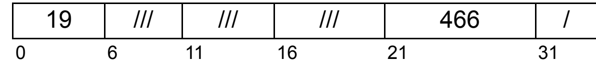
Special Registers Altered:
None

Programming Note

If the state of the Decrementer were not maintained and updated as if the thread was not in power-saving mode, Decrementer exceptions would not reliably cause exit from nap power-saving level even if Decrementer exceptions were enabled to cause exit.

Sleep *XL-form*

sleep



19	///	///	///	466	/
0	6	11	16	21	31

The thread is placed into sleep power-saving level.

When the thread is in sleep power-saving level, the state of all resources may be lost except for the HRMOR.

When the interrupt that causes exit from sleep power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:
None

Programming Note

If the state of the Decrementer is not maintained and updated, in sleep or rvwinkle power-saving level, as if the thread was not in power-saving mode, Decrementer exceptions will not reliably cause exit from power-saving mode even if Decrementer exceptions are enabled to cause exit.

Note

See the Notes that appear in the ***rvwinkle*** instruction description.

PRELIMINARY

Rip Van Winkle *XL-form*

rvwinkle

19	///	///	///	498	/
0	6	11	16	21	31

The thread is placed into rvwinkle power-saving level.

When the thread is in rvwinkle power-saving level, the state of all resources may be lost except for the HRMOR.

When the interrupt that causes exit from rvwinkle power-saving level occurs, resource state is as described in the preceding paragraph, except that if the exception that caused the exit is a System Reset, Machine Check, or Hypervisor Maintenance exception, resource state that would be lost if the exception occurred when the thread was not in power-saving mode may be lost.

An attempt to execute this instruction in Suspended state will result in a TM Bad Thing type of Program interrupt. <TM>

This instruction is hypervisor privileged and context synchronizing.

Special Registers Altered:
None

Programming Note

In the short story by Washington Irving, Rip Van Winkle is a man who fell asleep on a green knoll and awoke twenty years later.

Note

See the Notes that appear in the ***sleep*** instruction description.

---------------------------- End text --------------------------------

Section 4.4.2+ Transactional Memory Instructions [Category: Transactional Memory]

Insert the following section after Section 4.4.2.

-------------------------- Begin text --------------------------------

Privileged software that makes the Transactional Memory Facility available to applications takes on the responsibility of managing the facility's resources and the application's transactional state during interrupt handling, service calls, task switches, and its own use of TM. In addition to the existing instructions like ***rfid*** and problem state TM instructions that play a role in this management, ***treclaim*** and ***trechkpt.*** may be used, as described below. See <crossref to sec 3.2.1+> for additional information about managing the TM facility and associated state transitions.

Transaction Reclaim *X-form*

treclaim. RA

31	///	RA	///	942	1
0	6	11	16	21	31

```
CR0 <-  0 || MSR_TS || 0

if MSR_TS = 0b10 | MSR_TS = 0b01 then
      #Transactional or Suspended
      if RA = 0 then cause <- 0x00000001
       else        cause <- GPR(RA)_56:63 || 0x000001
      if TEXASR_FS = 0 then
        Discard speculative storage updates
      TMRecordFailure(cause)
      Revert speculative registers to pre-transac-
tional values
      Discard all resources related to current
transaction

MSR_TS <- 0b00                  #Non-transactional
```

The ***treclaim.*** instruction frees the transactional facility for use by a new transaction. It sets condition register field 0 to 0 || MSR_{TS} || 0. If the transactional facility is in the Transactional state or Suspended state, failure recording is performed as defined in Section 8.3.2. If RA is 0, the failure cause is set to 0x00000001, otherwise it is set to $GPR(RA)_{56:63}$ || 0x000001. The speculative registers are reverted to their pre-transactional values, and all resources related to the current transaction are discarded, including any transactional storage updates (if they weren't already discarded for a pending failure).

The transaction state is set to Non-transactional.

If an attempt is made to execute ***treclaim.*** in Non-transactional state, a TM Bad Thing type Program interrupt will be generated.

This instruction is privileged.

Special Registers Altered
CR0 TEXASR TFIAR TS

PRELIMINARY

Programming Note

The ***treclaim.*** instruction can be used by an interrupt handler to deallocate the current thread's transactional resources in preparation for subsequent use of the facility by a new transaction. (***tabort*** is not appropriate for this use, because (a) the interrupt handler is in Suspended state and ***tabort*** in Suspended state leaves the thread in Suspended state, and (b) ***tabort*** in Suspended state does not restore the speculative registers to their pre-transaction values.) After ***treclaim.*** is executed, when the interrupted program is next dispatched it should be resumed by first using ***trechkpt.*** to restore the pre-transactional register values into the speculative register checkpoint. Failure handling for that program will occur when the program next attempts to execute an instruction in the Transactional state, which will cause the failure handler to be invoked because TDOOMED will be 1. (This will be immediate if the program was in the Transactional state when the interrupt occurred, or will be after ***tresume.*** is executed if the program was in the Suspended state when the interrupt occurred.)

Transaction Recheckpoint X-form

trechkpt.

31	///	///	///	1006	1
0	6	11	16	21	31

```
CR0 <-  0 || MSR_TS || 0

MSR_TS <- 0b01
TDOOMED <- 1
copy registers to speculative register checkpoint
```

The ***trechkpt.*** instruction copies the current (pre-transactional, saved and restored by the operating system) register state to the speculative register checkpoint. It sets condition register field 0 to 0 || MSR_{TS} || 0. The pre-transactional values are loaded into the speculative register checkpoint. TDOOMED is set to 0b1.

The transaction state is set to Suspended.

If an attempt is made to execute this instruction in Transactional or Suspended state or when $TEXASR_{FS}$=0, a TM Bad Thing type Program interrupt will be generated.

This instruction is privileged.

Special Registers Altered

CR0 TS

---------------------------- End text --------------------------------

Section 4.4.3 Move To/From System Register Instructions

Extend the mtmsrd description to cover TM-related state transitions (specifically the suppression of TM change).

------------------------- Begin text --------------------------------

Move To Machine State Register Doubleword X-form

mtmsrd RS,L

31	RS	///	L	///	178	/
0	6	11	15	16	21	31

```
if L = 0 then
   if (MSR_29:31 ¬= 0b010 | RS_29:31 ¬= 0b000) then
      MSR_29:31 <- RS_29:31
   MSR_48 ← (RS)_48 | (RS)_49
   MSR_58 ← (RS)_58 | (RS)_49
   MSR_59 ← (RS)_59 | (RS)_49
   MSR_0:2 4:28 32:47 49:50 52:57 60:62
          ← (RS)_0:2 4:28 32:47 49:50 52:57 60:62
else
   MSR_48 62 ← (RS)_48 62
```

The MSR is set based on the contents of register RS and of the L field.

L=0:

If bits 29 through 31 of the MSR are not equal to 0b010 or bits 29 through 31 of RS are not equal to 0b000, then the value of bits 29 through 31 of RS is placed into bits 29 through 31 of the MSR. The result of ORing bits 48 and 49 of register RS is placed into MSR_{48}. The result of ORing bits 58 and 49 of register RS is placed into MSR_{58}. The result of ORing bits 59 and 49 of register RS is placed into MSR_{59}. Bits 0:2, 4:28, 32:47, 49:50, 52:57, and 60:62 of register RS are placed into the corresponding bits of the MSR.

L=1:

Bits 48 and 62 of register RS are placed into the corresponding bits of the MSR. The remaining bits of the MSR are unchanged.

If the instruction attempts to cause an illegal transaction state transition (see Table 2, "Transaction state transitions that can be requested by , rfid, hrfid, and mtmsrd.," on page 25) or when TM is disabled by the PCR, a transition to Problem state with an active transaction, a TM Bad Thing type Program interrupt is generated (unless a higher-priority exception is pending). If this interrupt is generated, the value placed into SRR0 by the interrupt processing mechanism (see Section 6.4.3) is the address of the ***mtmsrd*** instruction.

PRELIMINARY

This instruction is privileged.

If L=0 this instruction is context synchronizing. If L=1 this instruction is execution synchronizing; in addition, the alterations of the EE and RI bits take effect as soon as the instruction completes.

Special Registers Altered:

MSR

---------------------------- End text --------------------------------

Add SPRs to the mtspr/mfspr tables.

-------------------------- Begin text --------------------------------

PRELIMINARY

PRELIMINARY

Figure 1. SPR encodings (Sheet 1 of 2)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
1	00000 00001	XER	no	no	64	B
8	00000 01000	LR	no	no	64	B
9	00000 01001	CTR	no	no	64	B
13	00000 01101	AMR	no[6]	no	64	S
17	00000 10001	DSCR	yes	yes	64	STM
18	00000 10010	DSISR	yes	yes	32	S
19	00000 10011	DAR	yes	yes	64	S
22	00000 10110	DEC	yes	yes	32	B
25	00000 11001	SDR1	hypv[3]	hypv[3]	64	S
26	00000 11010	SRR0	yes	yes	64	B
27	00000 11011	SRR1	yes	yes	64	B
28	00000 11100	CFAR	yes	yes	64	S
29	00000 11101	AMR	yes[6]	yes	64	S
128	00100 00000	TFHAR	no	no	64	TM
129	00100 00001	TFIAR	no	no	64	TM
130	00100 00010	TEXASR	no	no	64	TM
131	00100 00011	TEXASRU	no	no	32	TM
136	00100 01000	CTRL	-	no	32	S
152	00100 11000	CTRL	yes	-	32	S
157	00100 11101	UAMOR	yes[7]	yes	64	S
256	01000 00000	VRSAVE	no	no	32	B
259	01000 00011	SPRG3	-	no	64	B
268	01000 01100	TB	-	no	64	B
269	01000 01101	TBU	-	no	32	B
272-275	01000 100xx	SPRG[0-3]	yes	yes	64	B
282	01000 11010	EAR	hypv[3]	hypv[3]	32	EC
284	01000 11100	TBL	hypv[3]	-	32	B
285	01000 11101	TBU	hypv[3]	-	32	B
286	01000 11110	TBU40	hypv	-	64	S
287	01000 11111	PVR	-	yes	32	B
304	01001 10000	HSPRG0	hypv[3]	hypv[3]	64	S
305	01001 10001	HSPRG1	hypv[3]	hypv[3]	64	S
306	01001 10010	HDSISR	hypv[3]	hypv[3]	32	S
307	01001 10011	HDAR	hypv[3]	hypv[3]	64	S
308	01001 10100	SPURR	hypv[3]	yes	64	S
309	01001 10101	PURR	hypv[3]	yes	64	S
310	01001 10110	HDEC	hypv[3]	hypv[3]	32	S
312	01001 11000	RMOR	hypv[3]	hypv[3]	64	S
313	01001 11001	HRMOR	hypv[3]	hypv[3]	64	S
314	01001 11010	HSRR0	hypv[3]	hypv[3]	64	S
315	01001 11011	HSRR1	hypv[3]	hypv[3]	64	S

PRELIMINARY

Figure 1. SPR encodings (Sheet 2 of 2)

decimal	SPR[1]	Register Name	Privileged		Length (bits)	Cat[2]
	$spr_{5:9}$ $spr_{0:4}$		mtspr	mfspr		
318	01001 11110	LPCR	hypv[3]	hypv[3]	64	S
319	01001 11111	LPIDR	hypv[3]	hypv[3]	32	S
336	01010 10000	HMER	hypv[3,4]	hypv[3]	64	S
337	01010 10001	HMEER	hypv[3]	hypv[3]	64	S
338	01010 10010	PCR	hypv[3]	hypv[3]	64	S
339	01010 10011	HEIR	hypv[3]	hypv[3]	32	S
349	01010 11101	AMOR	hypv[3]	hypv[3]	64	S
512	10000 00000	SPEFSCR	no	no	32	SP
526	10000 01110	ATB/ATBL	-	no	64	ATB
527	10000 01111	ATBU	-	no	32	ATB
768-783	11000 0xxxx	perf_mon	-	no	64	S.PM
784-799	11000 1xxxx	perf_mon	yes	yes	64	S.PM
896	11100 00000	PPR	no	no	64	S
898	11100 00010	PPR32	no	no	32	B[5]
1013	11111 10101	DABR	hypv[3]	hypv[3]	64	S
1015	11111 10111	DABRX	hypv[3]	hypv[3]	64	S
1023	11111 11111	PIR	-	yes	32	S

- This register is not defined for this instruction.

[1] Note that the order of the two 5-bit halves of the SPR number is reversed.

[2] See Section 1.4.5 of Book I.

[3] This register is a hypervisor resource, and can be accessed by this instruction only in hypervisor state (see Chapter 2).

[4] This register cannot be directly written. Instead, bits in the register corresponding to 0 bits in (RS) can be cleared using *mtspr SPR,RS*.

[5] The register is Category: Phased-in.

[6] The value specified in register RS may be masked by the contents of the [U]AMOR before being placed into the AMR; see the ***mtspr*** instruction description.

[7] The value specified in register RS may be ANDed with the contents of the AMOR before being placed into the UAMOR; see the ***mtspr*** instruction description.

All SPR numbers that are not shown above and are not implementation-specific are reserved.

---------------------------- End text --------------------------------

Just before the special registers altered part of the mtspr description, add a statement that a move to a TM SPR in other than Non-transactional state causes a TM Bad Thing type Program interrupt.

-------------------------- Begin text --------------------------------

Execution of this instruction specifying an SPR number that is not defined for the implementation, including SPR numbers that are shown in Figure 1 but are in a category that is not supported by the implementation, causes one of the following.

- if spr_0=0:
  - if MSR_{PR}=1: Hypervisor Emulation Assistance interrupt
  - if MSR_{PR}=0: Hypervisor Emulation Assistance interrupt for SPR 0 and no operation (i.e. the instruction is treated as a no-op) for all other SPRs
- if spr_0=1:
  - if MSR_{PR}=1: Privileged Instruction type Program interrupt
  - if MSR_{PR}=0: no operation (i.e. the instruction is treated as a no-op)

If an attempt is made to execute ***mtspr*** specifying a TM SPR in other than Non-transactional state, a TM Bad Thing type Program interrupt is generated.

Special Registers Altered:

See Figure 1

---------------------------- End text --------------------------------

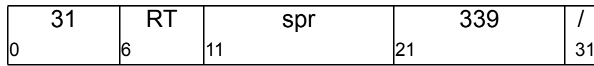
Zero the output for mfspr specifying the TFIAR when executed from a privilege level lower than that in which the TFIAR was set.

-------------------------- Begin text --------------------------------

PRELIMINARY

Move From Special Purpose Register XFX-form

mfspr RT,SPR




```
n ← spr5:9 || spr0:4
if length(SPR(n)) = 64 then
  if (n ¬= 129) | (MSRHV PR = 0b10)|
     (TFIARHV PR=MSRHV PR) |
     ((MSRHV PR = 0b00) & (TFIARHV PR= 0b01))then
     RT ← SPR(n)
  else
    RT ← 0
else
  RT ← 320 || SPR(n)
```

The SPR field denotes a Special Purpose Register, encoded as shown in Figure 1. If the designated Special Purpose Register is the TFIAR and TFIAR indicates the failure was recorded in a state more privileged than the current state, register RT is set to zero.<tm> Otherwise, the contents of the designated Special Purpose Register are placed into register RT. For Special Purpose Registers that are 32 bits long, the low-order 32 bits of RT receive the contents of the Special Purpose Register and the high-order 32 bits of RT are set to zero.

Programming Note

Note that when a problem state transaction's failure is recorded in hypervisor state and there is a subsequent need for a context switch in privileged, non-hypervisor state, an attempt to save TFIAR will result in zeros being saved. This is harmless because if the original application ever tries to read the TFIAR, it would read zeros anyway, since the failure took place in hypervisor state.

---------------------------- End text --------------------------------

Section 5.7.8: **Reference and Change Recording** To convey that transactional accesses are permitted to modify R/C/TS bits before the transaction commits, and even if the transaction fails, make the following changes in Figure 25:

-------------------------- Begin text --------------------------------

Status of Access	R	C
Indexed *Move Assist* insn w 0 len in XER	No	No
Storage protection violation	Acc[1]	No
Out-of-order I-fetch or Load-type Inst'n (including transactional Load-type inst'n or ***dcbtst***)	Acc	No
Out-of-order Store-type inst'n, including transactional Store-type inst'n, excluding ***dcbtst***		
Would be required by the sequential execution model in the absence of system-caused or imprecise interrupts[3], or transaction failure	Acc	Acc[1 2]
All other cases	Acc	No
In-order *Load*-type or *Store*-type insn, access not performed[4]		
Load-type insn	Acc	No
Store-type insn	Acc	Acc[2]
Other in-order access		
I-fetch	Yes	No
Ordinary *Load*, ***eciwx***	Yes	No
Other ordinary *Store*, ***ecowx***, ***dcbz***	Yes	Yes
icbi, ***dcbt***, ***dcbtst***, ***dcbst***, ***dcbf***[***l***]	Acc	No

"Acc" means that it is acceptable to set the bit.
1 It is preferable not to set the bit.
2 If C is set, R is also set unless it is already set.
3 For Floating-Point Enabled Exception type Program interrupts, "imprecise" refers to the exception mode controlled by $MSR_{FE0\ FE1}$.
4 This case does not apply to the *Touch* instructions, because they do not cause a storage access.

Figure 2. Setting the Reference and Change bits

---------------------------- End text --------------------------------

Section 5.9.3.1 SLB Management Instructions

Add the following programming note after the programming note that begins the section, noting the risk associated with modifying segment mappings of storage that has been accessed transactionally.

------------------------- Begin text --------------------------------

PRELIMINARY

Programming Note

Changes to the segment mappings in the presence of active transactions may compromise transactional semantics if the transaction has accessed a segment that is assigned a new VSID. Consequently, when modifying segment mappings, it is the responsibility of the OS or hypervisor to ensure that any transaction that may have touched the modified segment is aborted, using a ***tabort.*** or ***treclaim.*** instruction.

---------------------------- End text --------------------------------

Section 5.9.3.3 TLB Management Instructions

Insert the following programming note at the beginning of the section, explaining that tlbie maintains transaction consistency, but the tlbia and tlbiel do not.

-------------------------- Begin text --------------------------------

Programming Note

Changes to the page table in the presence of active transactions may compromise transactional semantics if a page accessed by a translation is remapped within the lifetime of a transaction. Through the use of a ***tlbie*** instruction to the unmapped page, an operating system or hypervisor can ensure that any transaction that has touched the affected page is aborted.

Changes to local translation lookaside buffers, through the ***tlbia*** and ***tlbiel*** instructions have no effect on transactions. Consequently, if these instructions are used to invalidate TLB entries after the unmapping of a page, it is the responsibility of the OS or hypervisor to ensure that any transaction that may have touched the modified page is aborted, using a ***tabort.*** or ***treclaim*** instruction.

---------------------------- End text --------------------------------

Section 5.10.1 Page Table Updates

Add text to the fourth paragraph in the section, stating that the execution of the sequence will cause effected transactions to fail.

-------------------------- Begin text --------------------------------

Software must execute ***tlbie*** and ***tlbsync*** instructions only as part of the following sequence, and must ensure that no other thread will execute a “conflicting instruction” while the instructions in the sequence are executing on the given thread. In addition to achieving the required system synchronization, the sequence will cause transactions that include accesses to the effected page(s) to fail.

---------------------------- End text --------------------------------

Sectino 6.4.3 Interrupt Processing

Add a new bullet to the first programming note, suggesting that treclaim should be executed in an interrupt handler prior to dispatching a new program.

-------------------------- Begin text --------------------------------

Programming Note

In general, when an interrupt occurs, the following instructions should be executed by the operating system before dispatching a “new” program.

- ***stbcx.***, ***sthcx.***, ***stwcx.,*** or ***stdcx.***, to clear the reservation if one is outstanding, to ensure that a ***lbarx***, ***lharx***, ***lwarx***, or ***ldarx*** in the interrupted program is not paired with a ***stbcx.***, ***sthcx.***, ***stwcx.,*** or ***stdcx.*** in the “new” program.
- ***sync***, to ensure that all storage accesses caused by the interrupted program will be performed with respect to another thread before the program is resumed on that other thread.
- ***isync*** or ***rfid***, to ensure that the instructions in the “new” program execute in the “new” context.
- ***treclaim***, to ensure that any previous use of the transactional facility is terminated.

---------------------------- End text --------------------------------

Section 6.5 Interrupt Definitions

TM Unavailable is a type of Facility Unavailable interrupt. See RFC 2230 for details. TM Bad Thing is a type of Program interrupt. There are no new vectors or state transitions to describe. The value of TM may be changed by interrupts, as indicated below.

-------------------------- Begin text --------------------------------

PRELIMINARY

Interrupt Type	MSR Bit IR	DR	FE0	FE1	EE	RI	ME	HV
System Reset	0	0	0	0	0	0	p	1
Machine Check	0	0	0	0	0	0	0	1
Data Storage	0	0	0	0	0	0	-	m
Data Segment	0	0	0	0	0	0	-	m
Instruction Storage	0	0	0	0	0	0	-	m
Instruction Segment	0	0	0	0	0	0	-	m
External	0	0	0	0	0	h	-	e
Alignment	0	0	0	0	0	0	-	m
Program	0	0	0	0	0	0	-	m
FP Unavailable[3]	0	0	0	0	0	0	-	m
Decrementer	0	0	0	0	0	0	-	m
Hypervisor Decrementer	0	0	0	0	0	-	-	1
System Call	0	0	0	0	0	0	-	s
Trace	0	0	0	0	0	0	-	m
Hypervisor Data Storage	0	0	0	0	0	-	-	1
Hypervisor Instr. Storage.	0	0	0	0	0	-	-	1
Hypv Emulation Assistance	0	0	0	0	0	-	-	1
Hypervisor Maintenance	0	0	0	0	0	-	-	1
Performance Monitor	0	0	0	0	0	0	-	m
Vector Unavailable[1]	0	0	0	0	0	0	-	m
VSX Unavailable[2]	0	0	0	0	0	0	-	m

0 bit is set to 0
1 bit is set to 1
p bit is set to 1 if interrupt ocurred while the thread was in power-saving mode; otherwise not altered
- bit is not altered
m if $LPES_1$=0, set to 1; otherwise not altered
e if $LPES_0$=0, set to 1; otherwise not altered
h if $LPES_0$=1, set to 0; otherwise not altered
s if LEV=1 or $LPES_1$=0, set to 1; otherwise not altered

Settings for Other Bits

Bits BE, FP, PMM, PR, SE, VEC[1], VSX[2], and TM[4] are set to 0.

If the interrupt results in HV being equal to 1, the LE bit is copied from the HILE bit; otherwise the LE bit is copied from the $LPCR_{ILE}$ bit.

The SF bit is set to 1.

If the TS field contained 0b10 (Transactional) when the interrupt occurred, the TS field is set to 0b01 (Suspended); otherwise the TS field is not altered.

Reserved bits are set as if written as 0.

[1] Category: Vector
[2] Category: Vector Scalar Emulation
[3] Category: Floating-Point
[4] Category: Transactional Memory

Figure 3. MSR setting due to interrupt

---------------------------- End text --------------------------------

Section 6.5.9 Program Interrupt

Insert the description of the TM Bad Thing after the description of the Floating-Point Enabled Exception.

------------------------- Begin text --------------------------------

PRELIMINARY

TM Bad Thing Exception[Category: Transactional Memory]

A TM Bad Thing exception is generated when any of the following occurs.

- An , ***rfid***, ***hrfid***, or ***mtmsrd*** instruction attempts to cause an illegal state transition (see <crossref to Section 3.2.1+>).
- An ***rfid***, ***hrfid***, or ***mtmsrd*** instruction attempts to cause a transition to Problem state with an active transaction (Transactional or Suspended state) when TM is disabled by the PCR (PCR_{TM}=1 or $PCR_{v2.06}$=1).
- An attempt is made to execute ***trechkpt.*** in Transactional or Suspended state or when $TEXASR_{FS}$=0.
- An attempt is made to execute ***tend.*** in Suspended state.
- An attempt is made to execute ***treclaim.*** in Non-transactional state.
- An attempt is made to execute a ***mtspr*** targeting a TM register in other than Non-transactional state.
- An attempt is made to execute a power saving instruction in Suspended state.

---------------------------- End text --------------------------------

Add bit 42 to SRR1 as the indicator for the TM Bad Thing exception.

------------------------- Begin text --------------------------------

SRR1

33:36	Set to 0.
42	Set to 1 for a TM Bad Thing Exception type Program interrupt; otherwise set to 0.
43	Set to 1 for a Floating-Point Enabled Exception type Program interrupt; otherwise set to 0.
44	Set to 0.
45	Set to 1 for a Privileged Instruction type Program interrupt; otherwise set to 0.
46	Set to 1 for a Trap type Program interrupt; otherwise set to 0.
47	Set to 0 if SRR0 contains the address of the instruction causing the exception and there is only one such instruction; otherwise set to 1.

Programming Note

$SRR1_{47}$ can be set to 1 only if the exception is a Floating-Point Enabled Exception and either $MSR_{FE0\ FE1}$ = 0b01 or 0b10 or $MSR_{FE0\ FE1}$ has just been changed from 0b00 to a nonzero value. ($SRR1_{47}$ is always set to 1 in the last case.)

Others Loaded from the MSR.

Exactly one of bits 42, 43, 45, and 46 is set to 1.

---------------------------- End text --------------------------------

Section 6.5.14 Trace Interrupt [Category: Trace]

Exclude instructions that are forbidden in Transactional state from being traced. The first paragraph of the section is modified as follows.

------------------------- Begin text --------------------------------

A Trace interrupt occurs when no higher priority exception exists and either MSR_{SE}=1 and any instruction except ***rfid*** or ***hrfid,*** is successfully completed, or MSR_{BE}=1 and a *Branch* instruction is completed. Successful completion means that the instruction caused no other interrupt and, if the processor is in the Transactional state <TM>, did not cause the transaction to fail in such a way that the instruction did not complete. (see <crossref to failure cause section>). Thus a Trace interrupt never occurs for a *System Call* instruction, for a *Trap* instruction that traps, or for a ***dcbf*** that is executed in Transactional state. The instruction that causes a Trace interrupt is called the "traced instruction".

When a Trace interrupt occurs, the following registers are set:

SRR0 Set to the effective address of the instruction that the thread would have attempted to execute next if no interrupt conditions were present.

SRR1

33:36 and 42:47
Set to an implementation-dependent value.

Others Loaded from the MSR.

MSR See Figure 3 on page 38.

Execution resumes at effective address 0x0000_0000_0000_0D00.

Extensions to the Trace facility are described in Appendix C.

PRELIMINARY

Programming Note

The following instructions are not traced.

- ***rfid***
- ***hrfid***
- ***sc***, and *Trap* instructions that trap
- other instructions that cause interrupts (other than Trace interrupts)
- the first instructions of any interrupt handler
- instructions that are emulated by software
- instructions, executed in Transactional state, that are disallowed in Transactional state
- instructions, executed in Transactional state, that cause types of accesses that are disallowed in Transactional state
- ***mtspr***, executed in Transactional state, specifying an SPR that is not part of the Transactional Memory speculative register state
- ***tbegin.*** executed at maximum nesting depth

In general, interrupt handlers can achieve the effect of tracing these instructions.

---------------------------- End text --------------------------------

Section 6.5.21+ Facility Unavailable Interrupt

Add the following programming note at the end of section.

------------------------- Begin text --------------------------------

Programming Note

For the case of an outer ***tbegin.***, the interrupt handler should either return to the ***tbegin.*** with MSR_{TM} = 1 (allowing the program to use transactions), or treat the attempt to initiate an outer transaction as a program error.

---------------------------- End text --------------------------------

Section 6.7 Exception Ordering

Extend the last paragraph of the section to include transaction failure and restore data segment interrupt, which was removed accidentally during integration of RFC 2108 going into v2.06. Also add clarification that data accesses need not be complete up to the point of interruption.

-------------------------- Begin text -------------------------------

Data Storage, Hypervisor Data Storage, Data Segment, and Alignment exceptions and transaction failure due to attempted access of a disallowed type while in Transactional state occur as if the storage operand were accessed one byte at a time in order of increasing effective address (with the obvious caveat if the operand includes both the maximum effective address and effective address 0). (The required ordering of exceptions on components of non-atomic accesses does not extend to the performing of the component accesses in the event of an exception. For example, if byte n causes a data storage exception, it is not necessarily true that the access to byte n-1 has been performed.)

---------------------------- End text --------------------------------

Section 6.7.2 Ordered Exceptions

Add the Transactional Memory type of (Hypervisor) Facility Unavailable interrupt(s). and the TM Bad Thing exception.

-------------------------- Begin text --------------------------------

PRELIMINARY

Instruction-Caused and Precise

1. Instruction Segment
2. [Hypervisor] Instruction Storage
3.a Hypervisor Emulation Assistance
3.b Program
   - Privileged Instruction
4. Function-Dependent
   4.a Fixed-Point and Branch
      1 Hypervisor Facility Unavailable
      2 Facility Unavailable
      3a Program
         - Trap
         - TM Bad Thing
      3b System Call
      3c [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      4 Trace
   4.b Floating-Point
      1 Hypervisor Facility Unavailable
      2 Floating-Point Unavailable
      3a Program
         - Precise Mode Floating-Pt Enabled Excep'n
      3b [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      4 Trace
   4.c Vector
      1 Hypervisor Facility Unavailable
      2 Vector Unavailable
      3a [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
      4 Trace

---------------------------- End text --------------------------------

Section 6.8 Interrupt Priorities

In the paragraph introducing the "instruction caused and precise" interrupts, add a statement that transaction failure takes priority over all but privileged instruction exceptions and group disallowed access types with "other" [H]DSI's, etc.

Add the (Hypervisor) Facility Unavailable interrupt.(s) and the TM Bad Thing exception here, too.

-------------------------- Begin text --------------------------------

4. Instruction-Dependent

   This exception is the third highest priority exception. When this exception is created, the interrupt mechanism waits for all possible Imprecise exceptions to be reported. It then generates the appropriate ordered interrupt if no higher priority exception exists when the interrupt is to be generated. Within this category a particular instruction may present more than a single exception. When this occurs, those exceptions are ordered in priority as indicated in the following lists. Where [Hypervisor] Data Storage, Data Segment, and Alignment exceptions are listed in the same item they have equal priority (i.e., the hardware may generate any one of the three interrupts for which an exception exists). For instructions that are forbidden in Transactional state, transaction failure takes priority over all interrupts except Privileged Instruction type of Program Interrupts. For data accesses that are forbidden in Transactional state, transaction failure has the same priority as the group of "other" [Hypervisor] Data Storage, Data Segment, and Alignment exceptions. (See <crossref to Bk2 Sec 7+.3.1>).

A. Fixed-Point Loads and Stores
   a. These exceptions are mutually exclusive and have the same priority:
      - Hypervisor Emulation Assistance
      - Program - Privileged Instruction
   b) Hypervisor Facility Unavailable
   c) Facility Unavailable
   d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
   e. Trace

B. Floating-Point Loads and Stores
   a. Hypervisor Emulation Assistance
   b) Hypervisor Facility Unavailable
   c) Floating-Point Unavailable
   d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
   e. Trace

C. Vector Loads and Stores
   a. Hypervisor Emulation Assistance
   b) Hypervisor Facility Unavailable
   c) Vector Unavailable
   d. [Hypervisor] Data Storage, [Hypervisor] Data Segment, or Alignment
   e. Trace

D. Other Floating-Point Instructions
   a) Hypervisor Facility Unavailable
   b) Floating-Point Unavailable
   c. Program - Precise Mode Floating-Point Enabled Exception
   d. Trace

E. Other Vector Instructions
   a) Hypervisor Facility Unavailable
   b) Vector Unavailable
   c. Trace

F. TM instruction, ***mt***/***fspr*** specifying TM SPR
   a.) Program - Privileged Instruction (only for ***treclaim.***, ***trechkpt.***., and ***mtspr***)
   b) Hypervisor Facility Unavailable
   c) Facility Unavailable
   d) Program - TM Bad Thing (only for ***treclaim.***, ***trechkpt.***, and ***mtspr***)

G. ***rfid, hrfid*** and ***mtmsr***[***d***]
   a) Program - Privileged Instruction
   b Program - TM Bad Thing exception for all except mtmsr.

PRELIMINARY

c. Program - Floating-Point Enabled Exception
d. Trace, for ***mtmsr***[***d***] only

H. Other Instructions
a. These exceptions are mutually exclusive and have the same priority:
- Program - Trap
- System Call
- Program - Privileged Instruction
- Hypervisor Emulation Assistance

b)Hypervisor Facility Unavailable
c)Facility Unavailable
d. Trace

I. [Hypervisor] Instruction Storage and Instruction Segment

These exceptions have the lowest priority in this category. They are recognized only when all instructions prior to the instruction causing one of these exceptions appear to have completed and that instruction is the next instruction to be executed. The two exceptions are mutually exclusive.

The priority of these exceptions is specified for completeness and to ensure that they are not given more favorable treatment. It is acceptable for an implementation to treat these exceptions as though they had a lower priority.

---------------------------- End text --------------------------------

Final Appendices

Appendix C. Platform Support Requirements

Add line for “Transactional Memory” after Trace, and mark it as a server requirement.

Appendix D. Complete SPR List

Carry the changes from the SPR tables in 3S 4.4.3 over to the table in this appendix.

Appendices G-J Opcode Map and Lists

Add the ***tbegin.***, ***tend.***, ***tabort.***, ***tabortwc.***, ***tabortwci.***, ***tabortdc.***, ***tabortdci.***, ***tsr.***, ***tcheck***, ***treclaim.***, and ***trechkpt.*** instructions to all the opcode maps and lists, indicating that the instructions are category: Transactional Memory.

---------------------------- End RFC -------------------------------

PRELIMINARY