Overview of Arm Transactional Memory Extension

Version 1.0

Non-Confidential

Issue
102873_0100_en

Overview of Arm Transactional Memory Extension

Release information

Document history

Issue	Date	Confidentiality	Change
0100	17 March 2022	Non-Confidential	Initial release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Contents

1 Overview

This guide describes transactional memory, which allows code to be executed atomically without the need to always implement performance-limiting synchronization methods. The Arm Transactional Memory Extension (TME) is the Arm implementation of transactional memory.

Modern complex systems execute on more than one processor. The management of memory accesses ensures that access is ordered when multiple processes or processors need to access memory locations at the same time. This management is also called memory synchronization. These systems ensure that accesses to memory occur atomically, which is important to ensure correct access to memory. Different implementations control memory synchronization, and we explain some of them in this guide. This guide is for developers and architects who want to learn about the concept of transactional memory, the Arm TME implementation, and how TME aids atomicity in system development.

Before you begin

This guide assumes that you are familiar with the Arm Instruction Set Architecture and the Arm AArch64 memory model. For more information, read our guides Armv8-A Instruction Set Architecture (ISA) and Armv8-A Memory model.

2 Introduction to atomic memory access in parallel processing systems

In multi-processing and multi-threaded systems, shared resources must be protected. This protection ensures that there is no conflict between threads which may need to read and modify the resources atomically.

Atomicity in memory access and code execution occurs when the actions of one observer on a system cannot be interrupted or be seen by another observer until the operation is complete. Atomic memory accesses are often used for critical code sections. In critical code sections, the code must be executed without any external interruption. In an atomic operation, any external observers must not be able to see any partial state of the atomic operation. External observers should only be able to see the previous state or the final state.

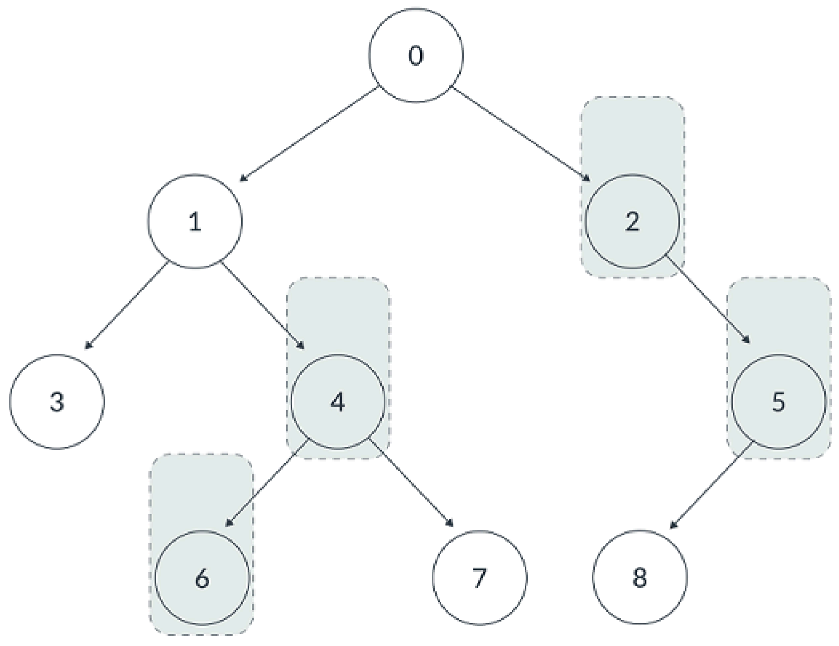
Atomic accesses are often protected using lock primitives to create a critical code segment. These lock mechanisms are often based on spinlock or mutex-based code segments to enforce the lock for the critical path. These mechanisms are overprotective and can act as a limit to performance in parallel processing systems. This is because they can halt execution in threads that are not specifically accessing the protected resource. A less restrictive and less pessimistic way to create critical code segments, which scale over large parallel processing systems, is to use the transactional memory method of protecting critical code segments.

Transactions do not remove the need for locks within the system. If multiple transactions try to access the same resource, transactions may require a mutex.

3 Why do we need transactional memory?

In distributed memory systems, multiple threads run across the system and access memory addresses. The following diagram shows a multi-process system in which the processes are making accesses to memory that might contend between two processes. A and B are two processes that are running on the wide system, and 2, 4, 5, and 6 are making accesses to memory. In a large system, these accesses may be to similar locations, which may cause contention:

Figure 3-1: Transactional memory

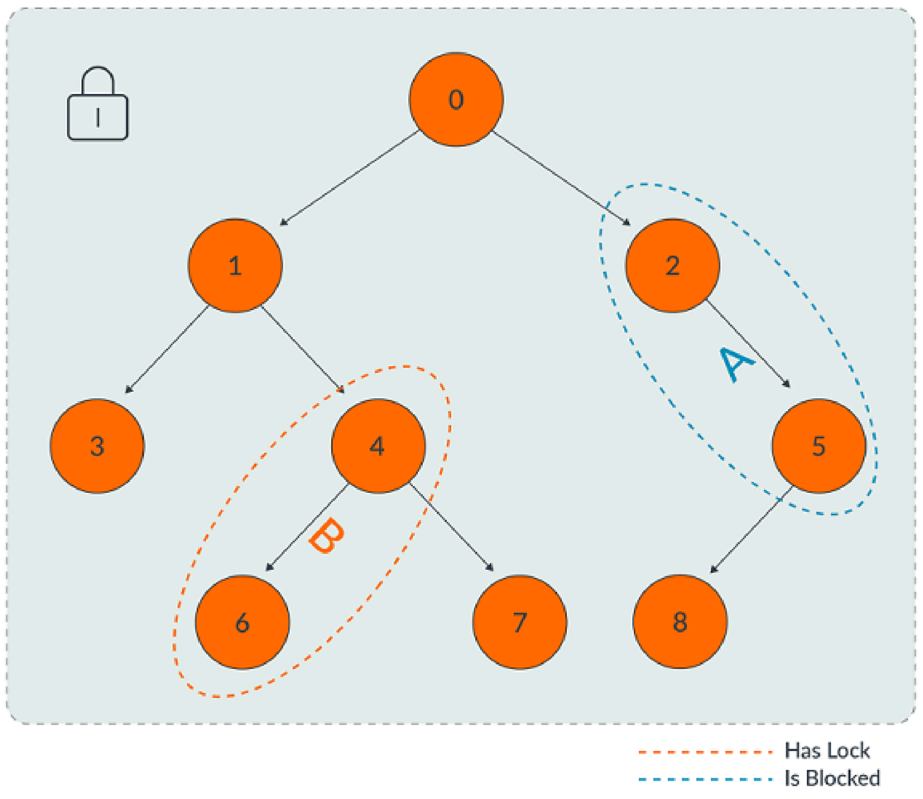


In many systems, there is a requirement for atomicity in memory access from the threads. This means that a mechanism must ensure that sensitive memory locations and peripherals are accessed by one thread at a time. There are many ways to apply lock in a complex system. These mechanisms include coarse-grained locks, fine-grained locks, and lock-free execution. The following sections describes these locks in more detail.

Coarse-grained lock, also called global lock

With a coarse-grained lock, also called a global lock, atomicity is maintained in a specific thread by locking access to the entire system whenever any critical code or data access is required. Locking access ensures that there is no interference from any other observer in the system. This is because any other code that is trying to access any memory space stalls until the global lock is removed. An example is shown in the following diagram. In the diagram, threads A and B make memory accesses within the system, and the system utilizes a global lock mechanism:

Figure 3-2: Coarse-grained lock



In the example, Thread A has taken a lock on the memory system to ensure atomic access. Because the lock is global, it stops any other accesses in the system until Lock A is removed. The operation of a global lock is restrictive, especially if Thread B does not access any similar addresses but is stalled until the lock taken by A is removed.

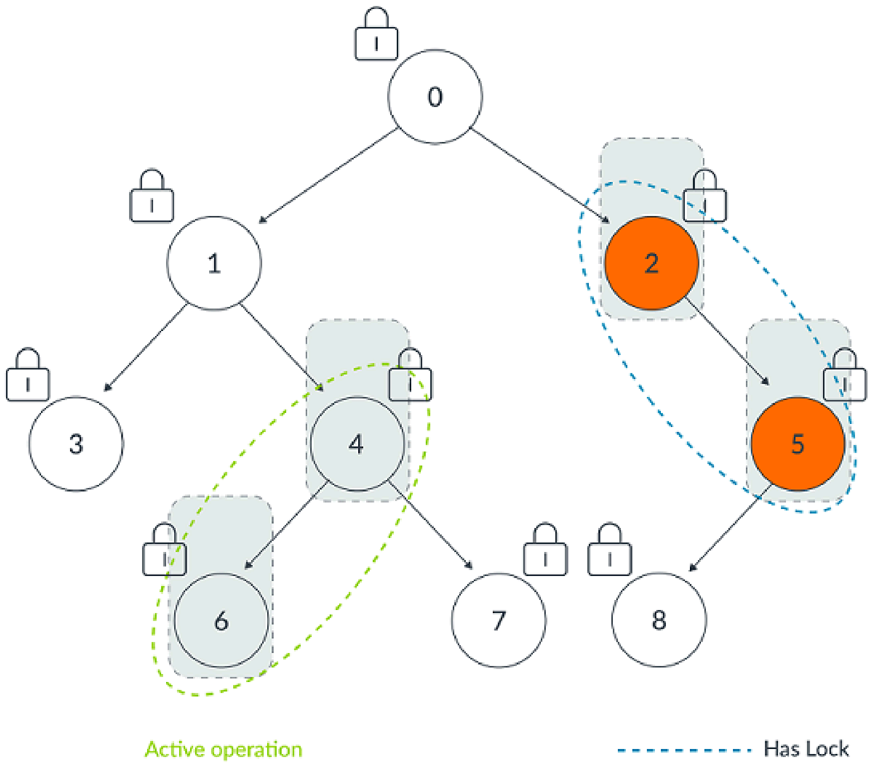
Fine-grained lock

When implementing fine-grained locking, the global lock use is analyzed and optimized to instances where it is needed. This means that locks are localized to ensure that they only work on the addresses that are being used. Therefore, other threads do not stall as much in operation.

Some software analysis and development is required to implement a fine-grained locking mechanism.

The following diagram shows how a fine-grained locking mechanism allows concurrent execution where the addresses do not contend. Thread A and thread B can both execute. This is because the lock is local to each process only and does not affect the whole memory system:

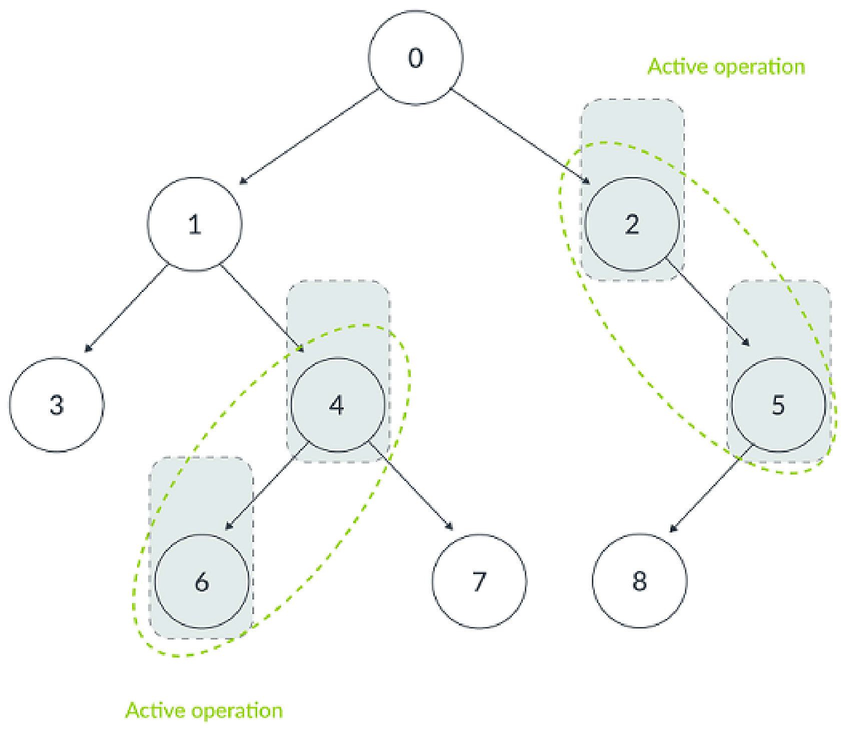
Figure 3-3: Fine-grained lock



Lock-free execution

Lock-free execution is a thread-safe access to shared data that does not use synchronization primitives like mutexes. For implementation on the host ISA, lock-free execution requires that the host ISA has atomic read-modify-write primitives utilizing hardware support. Lock-free execution can restrict the types of data structures that are used. This means that lock-free execution can give better performance in a multi-threaded platform, although better performance cannot be guaranteed. The following diagram shows a representation of lock-free execution. Threads A and B can execute without needing any locks, which gives the best performance for the system:

Figure 3-4: Lock free exectution



Coarse-grained lock, fine-grained lock, and lock-free execution are often used in the development of an Operating System (OS), to allow the use of shared resources between the many different threads on the OS.

The development of complex software systems can start with a coarse-grained lock, to ensure correct operation at the cost of overall performance. OS development often proceeds through fine-grained lock, so that more execution and analysis can be carried out on the system. Finally, OS software systems are developed to execute most memory accesses as lock-free execution. Lock-free execution takes more development time than the other lock methods, but does give the best performance for the OS

Note

These lock methods are described to show systems in which transactional memory can help development. Detailed discussion of these methods is beyond the scope of this guide.

Sometimes, developers want to give a performance boost to their distributed and shared code without the longer development time that is required of a fine-grained or lock-free system. Hardware Transactional Memory (HTM) addresses this need.

When implementing HTM, the Processor Element (PE) must have hardware extensions that allow transactions to be executed and failures to be detected. This means that HTM is not suitable for all types of development.

4 Hardware Transactional Memory

Hardware Transactional Memory (HTM) allows hardware extensions to ensure that memory accesses and code execution are atomic around specific code areas that are defined by the code developer.

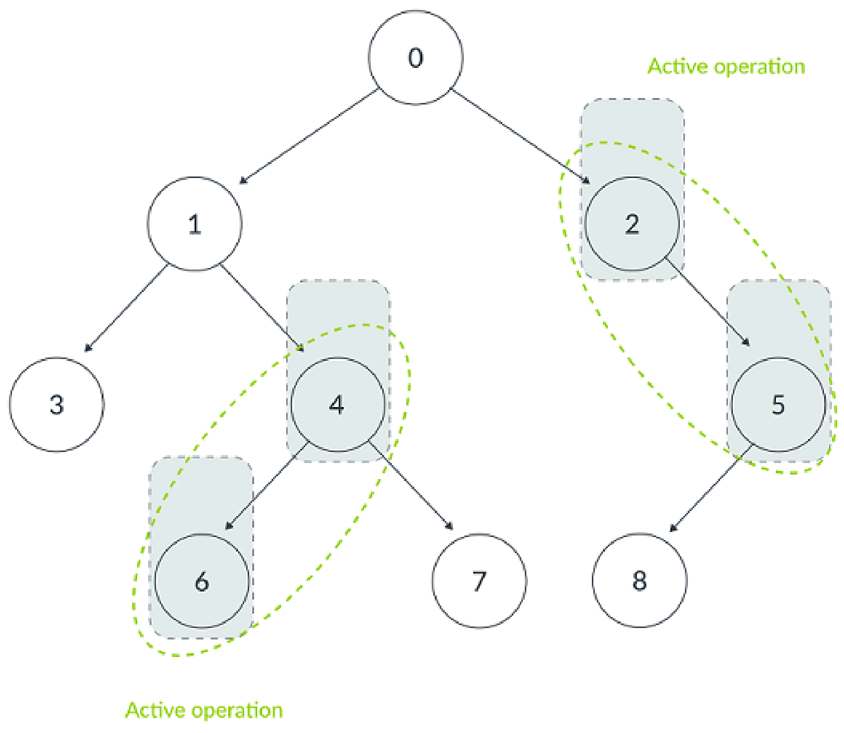
Transactional memory can be used to remove the need for global locks. Transactional memory can also remove the need for the lock analysis that is required to implement fine-grained locks. This lock analysis is required when threads run similar code but may not access the same memory location.

The transaction can be set around the critical code section. The transaction either:

- Allows the full section of critical code to complete
- Retains the full context of the start of the transaction to enable the transaction to re-execute

The following diagram shows a transactional memory implementation in operation. Thread A and Thread B can execute without any stalling, as long as there is no contention sensed by the system. If the system senses any contention, some code needs to execute to ensure that the memory access for each thread is not broken:

Figure 4-1: Lock-free execution



In the example, Thread A and Thread B both continue to execute. This is because there is no address contention on any of the accesses, so that the code execution in both cases can complete.

Transactional memory works by defining an area of code, called a transaction, that you want to execute.

A transaction should not be interrupted. A transaction needs to be isolated from other operational code in the system. Defining a transaction requires the start and the end of the code section to be marked for the PE. The transaction executes with no system context or memory accesses that are visible to external observers until the transaction has fully completed execution. At this point, all the system context is updated. This update allows any external observer to see the results of the transaction.

5 Basic transactional memory function

The basis of transactional execution means that the developer must define where a transaction starts and ends.

An example of a transaction is shown in the following code:

```
Start
TransactionStart() ; setting the PE into Transactional state
 …
 Many loads
 Many stores
 …
TransactionCheck() ; Has there been any contention
CBNZ Start   ; Roll back all context and try again else…
TransactionCommit() ; If check is ok then we can commit and continue
```

Often, a critical code section requires atomicity in the memory accesses of the critical section. A transaction allows code to be executed atomically. This means that a transaction may be used for critical code sections requiring atomicity.

At the end of the critical code section or the transaction, the PE can check if there are any memory access contentions in the execution stream.

If there has not been any contention in the execution stream, the buffered accesses are committed, and the execution continues from the end of the transaction.

If a contention is detected or observed, the transaction fails. If the transaction fails, all buffered memory operations and PE context are rolled back to the start of the transaction. The execution can either retry the transaction, or fall through to another mechanism for synchronization.

In most Hardware Transactional Memory (HTM) systems, the transaction passes and execution continues after the transaction is committed. This means that there is no observation of any locks. On the rare occasion that a transaction fails, code needs to allow an execution path to continue after the failure. This failure mode code is often called fallback code, because this is the code to fall to when the transaction fails. The fallback code either tries to rerun the transaction, or tries to use a more restrictive lock to allow the code to run without interruption.

HTM provides strong isolation between observers, but allows better performance than you would achieve with a strong coarse-grained lock strategy. Isolation is managed through the transaction execution. The memory system can:

- Detect access to the memory locations that are being used throughout the transaction, and any nested transactions
- Ensure that no external observer interferes with the atomicity of the transaction

In both cases, the execution can only be considered best effort. This is because if there is any breaking of the isolation or external interference, the code cannot be guaranteed to execute because the transaction can fail. If the transaction fails, fallback code is needed to continue execution and work around the transaction failure.

Transactional memory sizes

In a system that implements transactional memory, the hardware monitors:

- The transaction and buffer that all memory accesses made during the transaction
- Any other code execution, to check that there is no conflict with the addresses that are used in the transaction

Software users need to understand which areas of memory allow transactional memory execution and data access. This means that we need to be able to define the sizes of available transactional memory on a system.

Transactional memory systems define the Transactional Reservation Granule (TRG). The TRG is the smallest contiguous memory block size that incorporates the transaction operation. In many systems, this memory block is between 4 bytes and 512 bytes.

The transactional memory system monitors the memory operations that a transaction performs. Addresses that are accessed by read operations form the Transactional Read Set. Addresses that are accessed by write operations form the Transactional Write Set. The sizes of these sets are defined as multiples of the TRG.

The transaction fails if the number of entries in the Transactional Read/Write Set exceeds the capacity of the implementation. In many implementations, the size of the cache determines the number of entries in Transactional Read/Write Set.

Transactional memory isolation

The benefit of transactional memory is that it gives strong isolation to the executing code. Isolation is based on containment and non-interference from other transactions, and from external, concurrent code. Isolation occurs when the transactional memory system presents both strong containment and non-interference.

Transaction memory containment

Transactional memory guarantees memory containment during the transaction. This is because the transaction memory accesses either complete, or do not complete, for any external observer.

The following code show two processes that are running concurrently. The initial value in the address register X0 is the same in both processes, and the initial value of the memory location is 0:

P1	P2

`MOV X1, #1`
 	
`STR X1, [X0]`
 	
`MOV X2, #2`
 	
`STR X2, [X0]`
	`LDR X3, [X0]`

The preceding code shows a system without transactional memory. The following table shows the values that are seen by the external observer, P2, during the code execution. This example shows the case where the P2 load could occur at any point in the execution of P1. This means that we can see a P2 result in each occasion:

P1	-	P2
X1	X2	X3
0	0	0
1	0	1
1	2	2

In the preceding table, we can see that the operations in the process P1 are not contained. This is because all the memory updates in P1 are seen by the external observer P2.

The following code shows a system that incorporates transactional memory. In any transactional system, the coherency check is on the memory accesses. The transaction should be used around the critical code only, in this case P1. The HTM implementation will check for any contention externally to that transaction, in this case P2:

P1	P2

`MOV X1, #1`
 	
`STR X1, [X0]`
 	
`MOV X2, #2`
 	
`STR X2, [X0]`
 	
`TransactionCommit()`
	`LDR X3, [X0]`

The following table shows the values that the external observer, P2, can see:

P1	-	P2
X1	X2	X3
0	0	0
1	0	0
1	2	2

P2, the observer that is external to the transaction, must never be able to see the memory value from the store to X1 where the memory has 1 written. Therefore, the value that P2 loads into X3 can only be either 0 or 2.

If P2 load does occur during the transaction, then the transaction can fail and might rerun.

Non-interference

Transactional memory ensures that there is no interference from an external observer to memory accesses within a transaction.

Let’s look at two processes which are accessing the same area of memory. The memory location pointed to by both X0 registers is initialized to 0.

The following code shows a system, without transactional memory, running two concurrent processes, P1 and P2:

P1	P2

`LDR X2, [X0]`
 	
`LDR X3, [X0]`
	`MOV X1, #1`
	
	`STR X1, [X0]`

The following table shows that the state of P2 can have multiple effects on the values that are observed by P1. We can see that the P2 store occurs at different stages of the execution of P1, to show the effect on the values in P1:

P1	-	P2
X2	X3	X1
0	0	0
0	0	1
1	0	1
1	1	1

If the process is executed in a system with transactional memory, there is no interference during the transaction from the store at P2.

The following code shows an example of a system with transactional memory, where P1 and P2 are executing in parallel:

P1	P2

`LDR X2, [X0]`
 	
`LDR X3, [X0]`
 	
`TransactionCommit()`
	`MOV X1, #1`
	
	`STR X1, [X0]`

The values in X2 and X3 must both be 0, where P2 has not executed before P1. Alternatively, if P2 has executed before the transaction, the values must both be 1. There is no case where X2 and X3 are different values.

The following table shows all possible values that can be observed during the transaction by P1 while P2 is executing in parallel:

P1	-	P2
X2	X3	X1
0	0	0
1	1	1
1	1	1
1	1	1

If P2 did execute during the transaction, then the transaction can fail and might rerun.

6 The Arm Transactional Memory Extension

The Arm Transactional Memory Extension (TME) offers an execution state-based, isolated, best effort Hardware Transaction Memory (HTM) extension. The TME adds a new state, Transactional state, to the execution of the PE. A transaction is defined as all the instructions that are executing while the PE is in the Transactional state.

Any instructions that are executing outside a transaction are defined as executing in non-Transactional state.

The Arm TME adds new instructions to the Arm ISA to allow the definition of transactions within the instruction stream. The instructions can bound critical sections to define the transaction limits. The instructions are also designed to manage the transaction operation.

Here are the new instructions that are added to the PE for TME:

- `TSTART <Xd>  ; Transaction start instruction`
- `TCOMMIT   ; Transaction code section end`
- `TCANCEL <#imm6> ; Transaction cancel with reason code`
- `TTEST   : Transaction status and nesting level test`

Let's look at these instructions in detail.

Transaction start instruction

The `TSTART` instruction takes the form `TSTART <Xd>` where `Xd` is a destination register for the status value from the transaction. For example:

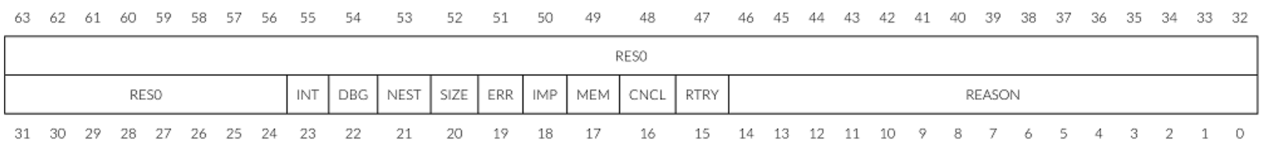
```
TSTART x15 ; X15 will hold the failure reason information
```

If the `TSTART` instruction is executed in non-Transactional state, then the PE enters Transactional state. `Xd` is populated with a pass or fail value, depending on whether the transaction has been started successfully or has failed.

If the `TSTART` is successful, then the register value is 0.

If there is a transaction failure, then the destination register is populated with an encoding of the failure reason. `X15` can be seen, in this example, as a transaction status register.

The bitfield encoding of this status register is shown in the following diagram:



The list defines each bit or bit field in the returned register:

- `INT` shows that an interrupt was asserted, but not taken, while in a transaction. When the transaction finishes and the non-transactional PE state is restored, the interrupt has not been taken. The `INT` bit is for information and is not a specific failure cause.
- `DBG` shows that an asynchronous external debug exception has caused the transaction to fail.
- `NEST` allows a set depth of nesting to be defined. If this depth is exceeded with the `NEST` bit set, the transaction fails.
- `SIZE` is used where the HTM design only allows a certain level of memory buffering for the context saving. This bit shows that the limit has been exceeded.
- `ERR` shows that an execution has caused transaction failure, in a case where architectural execution is not allowed in a transaction.
- `MEM` shows that the HTM system has detected a memory conflict in the transaction. The fallback code can directly retry the transaction. You can find more information in the explanation of the `RTRY` bit in this list.
- `CNCL` shows that a `TCANCEL` instruction was executed. The REASON field shows the cause of the cancel instruction. The reason field is a direct mapping of the 6-bit encoding of the `TCANCEL` instruction. The most significant bit of the 6-bit immediate `TCANCEL` value defines the `RTRY` value in the encoding.
- `RTRY` shows that the transaction can be restarted. The transaction might not fail in the next attempt. For example, if a memory conflict is detected during the transaction, the conflict might have cleared when retried. `RTRY` is additional information and, by itself, is not a failure cause.
- `IMP` shows that an implementation issue might cause the transaction failure. `IMP` also shows any failure mode that does not fall into any of the above categories. `RTRY` value depends on the implementation and reason for failure.

If the `TSTART` instruction is executed in Transactional state, then it maintains the current Transactional state. However, the `TSTART` instruction indicates nested execution in the status register `NEST` bit.

Transaction end instruction

The `TCOMMIT` instruction terminates a transaction.

This instruction commits all system context modifications during the current transaction, including any system context modifications during any nested transactions, to the current architectural state. It also changes the PE state from Transactional state to non-Transactional state.

This instruction is **UNDEFINED** if executed in non-Transactional state.

If the `TCOMMIT` instruction is executed in nested Transactional state, then it will not commit the system context. If the nesting depth is at a level of 1, it indicates that the PE is moving to an outer transaction from a nested transaction.

The following code gives a simple function to show how these instructions can be used in a simple transaction:

```
Simple_function
 TSTART X16   ; Start the transaction
```

```
  CBNZ X16, FAIL ; Check the status register for pass
       ; or branch to fallback
  LDR  X2, [X0] ; Some critical code
  LDR  X3, [X0] ; may include memory accesses
  TCOMMIT    ; finish transaction and update status
  RET
 FAIL
  BL Fail   ; some code to catch execution if
       ; transaction fails
```

Transaction cancel instruction

The `TCANCEL` instruction takes the form `TCANCEL <#imm6>` where the `<#imm6>` value defines the reason for the cancellation of the transaction. An example of this instruction is:

```
TCANCEL #0xFFFF ; Cancel transaction with RTRY set to 1
```

This instruction cancels the transactional mode and cancels, or rewinds, all context state modifications and the PC. The immediate value sets the failure reason code in the returned destination register. Execution is passed to the instruction following the `TSTART` instruction.

`Bit[15]` of the immediate value is the `RTRY` bit. The `RTRY` bit indicates whether a failure could be immediately retried, or should fail the transaction and execute the fallback code.

Transaction test instruction

The `TTEST` instruction takes the form `TTEST <Xd>` where `<Xd>` is the destination register that has the returned value for the transaction test result. An example of this instruction is:

```
TTEST X3 ; Test whether in transactional state and nesting depth
```

This instruction returns the status of the current execution in terms of the Transactional state and whether there is any level of transaction nesting. The instruction returns 0 if executed in non-Transactional state, and 1 if executed in an outer transaction. The instruction performs an incrementing count for any further level of nesting transactions.

Transaction nesting

Transactions run a flattened nesting model. In a flattened nesting model, any transactions that are called from within a transaction indicate a move to nested transaction execution. However, when the `TCOMMIT` instruction is executed on the nested transaction, the instruction will not update the system context until the outer transaction is committed. An internal counter for nest depth is incremented for every `TSTART` instruction, and decremented for every `TCOMMIT` instruction. This gives the depth of nesting in Transactional state.

In a flattened nesting model, all nested transactions share a transaction read/write set with the outer transaction. Any `TCANCEL` or transaction failure fails immediately to the outermost transaction.

Transaction development is not expected to use nesting. However, the development does allow for modular code where a larger transaction calls common subroutines that also use transactions.

The following code shows an example of inadvertent nesting of transactions. In this example, the transactional state is set up in the critical code path where the transactional function `signal_lock()` is called. This will cause a second level of transactional state:

```
signal_lock   ; simple lock flag
 TSTART X4  ; start transaction
 CBNZ fail  ; fallback if fail
 LDR X3, [X0]
 MOV X2, #1
 STR X2, [X0]
 TCOMMIT   ; This may not update the context if
     ; this is a nested transaction
fail BL Fallthrough

Critical_Code  ;code which will call the nested transaction
 TSTART X8  ;Start of the outer transaction
 CBNZ CCFail
 MOV X0, #Lock_Signal_Address
 BL signal_lock
 <critical code segment>
 <many loads and stores>
 TCOMMIT     ; At this point all outer and inner
       ; transaction context is updated
```

The critical code transaction is classed as an outer transaction. This is because it is the first instance of the `TSTART` instruction. This code calls the function `signal_lock()`, which also is defined as a transaction. The function `signal_lock()` is called a nested transaction. System context is only updated when the outer transaction `TCOMMIT` is executed. The outer transaction update must include any context that is updated in the nested transaction.

If the nested transaction fails or is canceled, then the outer transaction must also fail.

7 Interrupts in transactional memory systems

A transaction does not allow any change of Exception level or any interrupts to be serviced on the PE. An interrupt or other exception that changes the flow of execution is called non-sequential execution. Any interrupts that are asserted during a transaction are pended until the transaction is completed or fails.

If less latency on interrupt servicing is needed, some implementations might allow an interrupt to fail a transaction. If an interrupt causes a transaction failure, then the status register that is returned has the IMP bit set to show the failure.

If the implementation does allow for interrupts to fail transactions:

- The core is executing in non-Transactional state with interrupts masked.
- The core enters Transactional state.
- Interrupts are unmasked in the Transactional state.
- An interrupt is asserted, and the implementation allows interrupts to fail the transaction.
- The PE then fails the transaction and unwinds the context back to the status before the Transactional state was entered. In this case, the interrupts were masked. This means that the pending interrupt may not be taken at the current Exception level.

Where all the above conditions are met, the PE must set the status register on exit of Transactional state. To allow servicing of this exception, the IMP bit and the INT bit are both set to allow the fallback code information to unmask interrupts. This cannot be allowed to retry as this may cause a livelock . The livelock could occur where the transaction will be retried and fail every time while the interrupt is pending.

8 Execution constraints in the Arm Hardware Transactional Memory Extension

The Arm Hardware Transactional Memory (HTM) Extension allows transactions to be run on HTM-based PEs, to ensure that the hardware and software can monitor the transaction execution, and to detect any memory access contention. Some constraints need to be placed on system operations and code execution. This section of the guide describes the effects of the Arm HTM extension to system operations, for example exceptions, and context changing instructions when executing transactions.

Within a transaction, instructions must execute in a simple execution model where any exceptions or system context changing instructions fail the transaction. This means that all asynchronous exceptions at the execution Exception level are blocked during the period that the PE is executing in Transactional state.

Many context-changing instructions are not allowed within a transaction or have modified execution behavior when executed in Transactional state. Here are some examples:

- Exception generating instructions - `SVC, HVC, SMC`
- Exception Return instructions - `ERET, ERETAB`
- System instructions - `AT, TLBI, DC (except ZVA)`
- Hint instructions - `WFI`
- Barrier instructions - `DSB, ESB`

These instructions cause a transaction failure if they are executed. Similar instructions, for example, `MSR, SYS, DMB`, and `ISB` , also modify behavior if executed in Transactional state. The Arm Transactional Memory Extension specification document includes tables of all instructions that cause transaction failure or have their execution modified.

Memory management and translation

Memory types that are architecturally guaranteed to support transactional memory are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read and Write allocation. The memory type must not be transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read and Write allocation, the memory type must not be transient.

Transactional memory is based on the coherency and sharing of data. Transactional memory systems are usually implemented as part of the cache system that uses the coherency extensions. This implementation allows these memory types to be guaranteed. Although there is no architectural reason for memory types not to have transactional extensions, the following memory types can be poor targets to implement transactional memory:

- Device memory – Any reads or writes to device memory can cause side effects so it is not easy to wind back the transaction if it fails
- Non-cacheable memory – Transactions rely on the memory accesses only being observable at the end of the transaction so there would need to be some buffering implemented with the non-cacheable memory
- Memory that does not support hardware cache coherency – This is because the system needs to check all accesses for coherency with the transaction memory region

If transactional memory is not implemented in certain areas of an address map, and the PE tries to access those addresses while in a Transactional state, then the transaction fails with the IMP bit set in the returned status register.

We recommend that memory accesses within a transaction all have the same memory attributes. If they do not, the accesses may be **CONSTRAINED UNPREDICTABLE**.

Address translation

Address translations will occur during transactions. Transaction execution does not change the way that address translation operates, and does not change:

- Translation table walking
- Updating of the Translation Lookaside Buffer (TLB)

The operation of the Memory Management Unit is described in our Memory Management guide.

Translations can update the TLB even if the transaction fails.

When an address translation fails in non-Transactional state, it will generate an exception. Address translation exceptions are blocked in Transactional state. If a translation exception is sensed, the transaction will fail with an `ERR` response. When a PE is executing in Transactional state, the transaction will also fail if:

- Another PE broadcasts a TLB Invalidate command, `TLBI`, and
- Translated addresses within the transaction read/write set are being invalidated

Hardware management of access flags and dirty flags in translation table

Hardware management of access flags and dirty flags is a requirement for the better performance during transactions. This is because of the high software overhead for any flag management within a transaction where the access is made within a transaction, and then must be reset if the transaction fails.

External observers can see hardware management changes to the flags while transactions are executing, and not just at the commit stage. If a change has been made to the flags during the transaction, then a transaction failure occurs.

9 Debug in PEs with the Transactional Memory Extension

Debug systems on Arm cores are often defined as invasive, for example halting and self-hosting, and non-invasive, for example trace and performance monitoring.

Invasive debug

Often, debug operations cause a break in the simple sequential execution that a transaction requires, and causes a transaction to fail. There is a specific bit in the status register, the `DBG` bit, that is returned to the transaction after the `TSTART` instruction to define a debug transaction failure.

Any debug instruction, for example `BRK`, fails a transaction, with the DBG bit set in the status register for the transaction.

`TSTART` instructions cannot be single-stepped into. Software stepping cannot be enabled or disabled on the PE, because this would mean using an `MSR` instruction to a context-changing internal register. As we discussed in the Execution constraints in The Arm Transactional Memory Extension, single stepping will fail the transaction.

External debug requests and reset catches are treated like other asynchronous exceptions. This means that they are pended until the Transactional state is exited for completion or failure. Like with interrupts, an external debug request can fail a transaction when asserted, if an implementation requires a lower latency to debug requests. Exception catch and OS unlock catch cannot occur inside transaction. This is because exceptions and System register updates cannot occur in Transactional state.

Non-invasive debug

Most non-invasive systems are outside of the central PE execution. These systems are usually classed as separate observers. They are classed as separate observers so that they do not fail transactions in operation, and can give some debug capability on systems that implement non-invasive debug methods.

Arm Embedded Trace Macrocell

The Arm Embedded Trace Macrocell for transactional memory-based systems is modified to ensure that it can identify transaction start, commit, and failure in the trace packet protocol. The identifiers are markers for the trace stream. The identifiers can guide whether the instructions are deemed to have executed (commit) or have not succeeded (fail). The trace system discards execution trace on a failure. All code that is executing in Transactional state is marked as Speculative until the outer transaction is completed (commit or fail). The Embedded Trace Extension architecture allows system designers to define how they can resolve the transaction. This means that system designers can decide whether any trace can be collected during a failed transaction.

Statistical profiling

The Arm v8.2-A architecture provides Statistical Profile Extensions to aid code optimization. See the Statistical Profiling Extension for more information on these extensions.

The Statistical Profile Extensions have been extended to ensure that Transactional state can be defined and filtered if necessary.

Performance Monitoring Unit

The Performance Monitoring Unit (PMU) is extended in systems with the Transactional Memory Extension (TME) to be able to count the executed instruction cycles within a transaction if the transaction commits. PMU events available for HTM processing elements include:

- All TSTART instructions retired
- Any TCOMMIT that is executed (retired)
- Any transaction failure. There are separate events for different failure operation.

If there is a `TCANCEL` or FAIL, then the PMU count resets to the `TSTART` value.

The PMU can be configured with counter-overflow asserting interrupts. If these are being used in the Transactional state, the interrupt can be pended until the transaction commits. If the transaction fails, then the count is reset to the value at which `TSTART` was executed and therefore will not overflow and will not assert the interrupt.

Transaction Lock Elision An elision is the act of omitting a part or merging of parts. In lock elision, the use of a lock is merged into another, less prohibitive, method for synchronization as an alternative if the non-blocking method fails.

In transactional operations, the Transaction Lock Elision (TLE) uses the Hardware Transactional Memory (HTM) to reduce the potential delay in a spinlock that is protecting some critical code. If there is no contention and the transaction can commit, then the more restrictive lock will not be used. If the transaction does not commit, then the routine continues with a standard spinlock to reduce the contention with a potential drop in performance of the system.

Currently, TLE is the most common use of transactional memory extensions in multiprocessing systems.

The following code shows how to use a transactional lock elision where a lock is acquired if the transaction fails. W6 is set up with a retry count for multiple retry options and W0 has our lock value:

```
loop:       TSTART X5           (A1)
        CBNZ X5 fallback        (A2)
        LDAR W5, [X1]       (A3)
        CBZ W5, transaction (A4)
        TCANCEL #0xFFFF     (A5)
fallback:   TBZ X5, #15, lock   (B1)
        SUB W6, W6, #1      (B2)
        CBZ W6, lock        (B3)
acq_loop: LDAR W5, [X1]     (C1)
        CBNZ W5, acq_loop   (C2)
        B loop          (C3)
lock:       PRFM PSTL1KEEP, [X1]    (D1)
lock_loop:  LDAXR W5, [X1]      (D2)
        CBNZ W5, lock_loop  (D3)
        STXR W5, W0, [X1]   (D4)
        CBNZ W5, lock_loop  (D5)
        DMB ISH         (D6)
```

```
    transaction:                 (E1)
            <main code>      (E2)
            <many loads/stores> (E3)
```

Here is an explanation of the code:

The transaction is started at point (A1). The transaction executes and commits if there is no failure, which ensures that there is no need to run the restrictive lock code. The check after the `TSTART` only moves to the fallback code if the transaction fails (A2) for any reason.

We must ensure that the coarse lock address is within the transaction read set. This means that the lock address must be in the coherent cache to be guaranteed to work with the transactional memory. Therefore, we execute an explicit load (A3) to force the address into cache. We use a load acquire instruction to ensure that any accesses to the address appear in order. The lock value is 0 for unlock. We can compare the returned lock value with 0 (A4) to see that there is no current lock that is taken. This means that we can execute the transaction main body code (E1-En) with the lock address now part of the transaction read set. Any external access to this address immediately fails the transaction.

If the lock value check fails, there is another process already using the restrictive lock and we cannot allow the transaction to proceed. At this point, we must cancel the transaction with a nonzero value that reflects the reason for failure. We can set the `RTRY` flag to indicate an external memory conflict. The lock will be removed in the next pass. This means that we can try to run the transaction after the external lock has been removed to give better performance. We can use the return value `0xFFFF` to ensure that the msbit is set (A5) for the `RTRY` value.

The fallback loop (B1-B3) catches any transaction fail or locked peripheral and checks whether we can retry the transaction. We initially check bit-15 of the returned status register that is returned after the failure of the transaction (B1). We also check that we still have a zero retry count (B2) which allows us to attempt the transaction again (B3). If there were no available retries, we wait until the current lock is cleared before we can retry the transaction.

We continue to loop while loading until lock value until the lock is cleared by an external source (C1-C3). When the lock is cleared, we can execute the `transaction:` code (E1-En) providing that the transaction does not fail due to external interference.

The `lock:` routine is used only if there are no possible retries either from the test on the status register bit-15 (B1), or from the retry count being nonzero (B2). This is a read modify write routine in which:

1. The code initially ensures that there is a valid unique lock value in our transactional write set using the exclusive load. This value allows us to check for any other actor that is potentially writing this address (D1-D2).
2. Execution loads the flag and loops until the flag value is zero (unlocked) (D3).
3. The exclusive store at (D5) checks that there is no other lock requested.
4. A coarse-grained lock is claimed with a store exclusive. This enables us to check that we have unique ownership of this location (D4-D5). The memory barrier (D6) ensures that the lock store has completed.

10 Check your knowledge

Q1: What is the most restrictive type of lock that is used in system development to ensure atomicity in memory accesses?

A1: The most restrictive, and pessimistic, lock method is the global or coarse-grained lock, which locks all other memory access while the atomic operation is executing.

Q2: Are all the transactions in a Hardware Transactional Memory (HTM) system guaranteed to complete?

A2: No. Because HTM transactions execute on a best effort policy, there is no guarantee that they execute to the final commit.

Q3: In an Arm HTM transaction where another external actor already has a lock, which instruction can be used to cause the transaction to fail? How can the instruction recommend that the transaction be run again instead of executing a more restrictive lock?

A3: If there is a requirement to stop a transaction, use the `TCANCEL` instruction. This instruction has encoding to give a reason code for the cancellation. This encoding includes a bit[15] `RTRY`, which signals that a transaction can be re-executed.

Q4: Can any Arm instruction be used in a transaction?

A4: No, there are some restrictions on instructions that can be used in a transaction, where:

- Instructions that are used in a transaction must obey the simple execution model
- Must not cause any exception, for example, `SVC` or `SMC`
- Must not make any changes to the System register context, examples of which are `WFI` or `TLBI`.

11 Related information

Here are some resources related to material in this guide:

- A-Profile Architecture Specifications
- Arm Community: Forums, articles, and blogs on many Arm topics from Arm experts
- Learn the architecture: AArch64 memory management: This guide includes an explanation of the operation MMU
- Learn the architecture: A series of guides that explain aspects of the Arm architecture

Other architecture-related resources:

- Platform Design
- Porting and Optimizing HPC Applications for Arm SVE Documentation
- Statistical Profiling Extension

Useful links to training:

- What is architecture? The training can be accessed if you have an Arm account, or if you have paid for training access.

12 Next steps

This guide introduces the basics of Hardware Transactional Memory (HTM) and the Arm Transactional Memory Extension (TME). The guide defines the new instructions added to control the execution in Transactional state. The guide is written to introduce the Arm TME and how transactions can be used.

If you would like to learn more about how the Arm TME works within the Arm ISA architecture and within the wider systems, see the resources in Related information.