This presentation will discuss the major new CPU facilities added to the IBM zEnterprise EC12 system.

Earlier versions of this presentation were provided to members of the IBM Early-Support Program for the zEC12 in the summer of 2012 and at the IBM Technical Disclosure Meeting in the autumn of 2012. If you attended such a presentation, please be advised that the SHARE-120 version contains numerous corrections and clarifications.

The Legal Stuff

- **The following terms are registered trademarks of the International Business Machines Corporation in the United States, other countries, or both:**
  - **IBM**
  - **IBM logo**
- **The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:**
  - **ESA/390**
  - **z/Architecture**
  - **z/OS**
  - **z/VM**
- **The following are trademarks or registered trademarks of other companies:**
  - **IEEE is a trademark of the Institute of Electrical and Electronics Engineers, Inc. in the United States, other countries, or both.**
  - **Java is a trademark of Oracle America, Inc. in the United States, other countries, or both.**
  - **Linux is a registered trademark of Linus Torvalds in the United States, other countries or both.**
  - **Unicode is a registered trademark of Unicode, Incorporated in the United States, other countries, or both.**
  - **Other trademarks and registered trademarks are the properties of their respective companies.**

- All information contained in this document is subject to change without notice. The products described in this document are not intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.
- While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.
- The information in contained in this document is provided on an "AS IS" basis. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.
- This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.
- All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only

This slide reviews the trademarks that may be shown in the presentation. Also, this slide includes various disclaimers as to the content of the presentation.

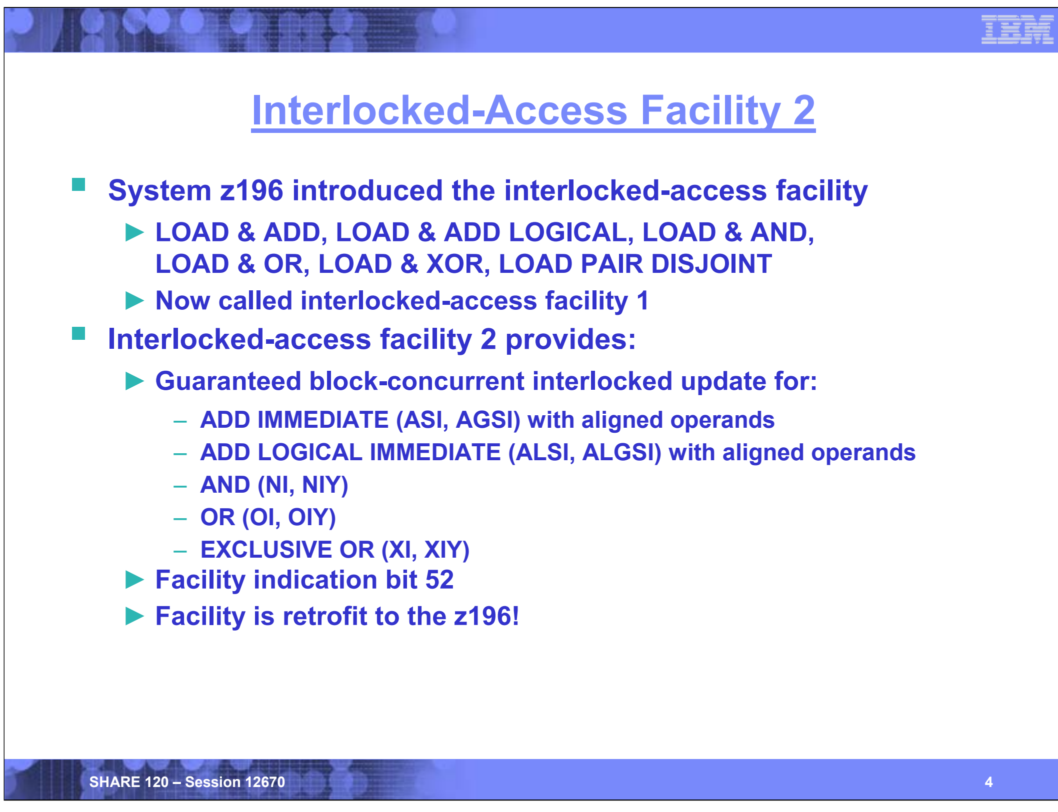
Topics du Jour

- Interlocked-access facility 2
- DFP zoned-conversion facility
- Execution-hint facility
- Load-and-trap facility
- Miscellaneous general instructions
- Transactional-execution facility
- Processor-assist facility
- Enhanced-DAT facility 2
- Local-TLB-clearing facility

This slide enumerates the new CPU facilities introduced in the IBM z/Enterprise EC12. Each of these facilities will be discussed in detail in the subsequent slides.

The session describes the CPU facilities and instructions in detail, and assumes a familiarity with assembler language, machine-instruction formats, and basic z/Architecture.

Much of the material described in today's presentation is related to the characteristics of multiprocessing … particularly, in improving the performance of MP applications that share common memory locations.



The interlocked-access facility was introduced in the System z196 processor in September of 2010, and included the instructions listed on this slide.

These instructions provide improved performance for certain sequences of operations that may be executed in a multiprocessing environment. The instructions provide a block-concurrent, interlocked update for loading, performing and operation, and storing a result (commonly known as an atomic operation). This facility is now called the interlocked-access facility 1.

The interlocked-access facility 2 provides an assurance that the instructions listed here will also perform in an interlocked, block-concurrent manner. Many of these instructions such as AND (NI), EXCLUSIVE OR (XI), and OR (OI) have existed since the original S/360, with programming notes advising that they cannot safely be used in an MP environment. The interlocked-access facility 2 changes that, assuring that these instructions will perform in an interlocked manner.

The interlocked-access facility 2 was actually present in the System z196, but no facility indication was originally provided. Through a firmware upgrade, the interlocked-access facility 2 indication is now provided on all z196 processors.

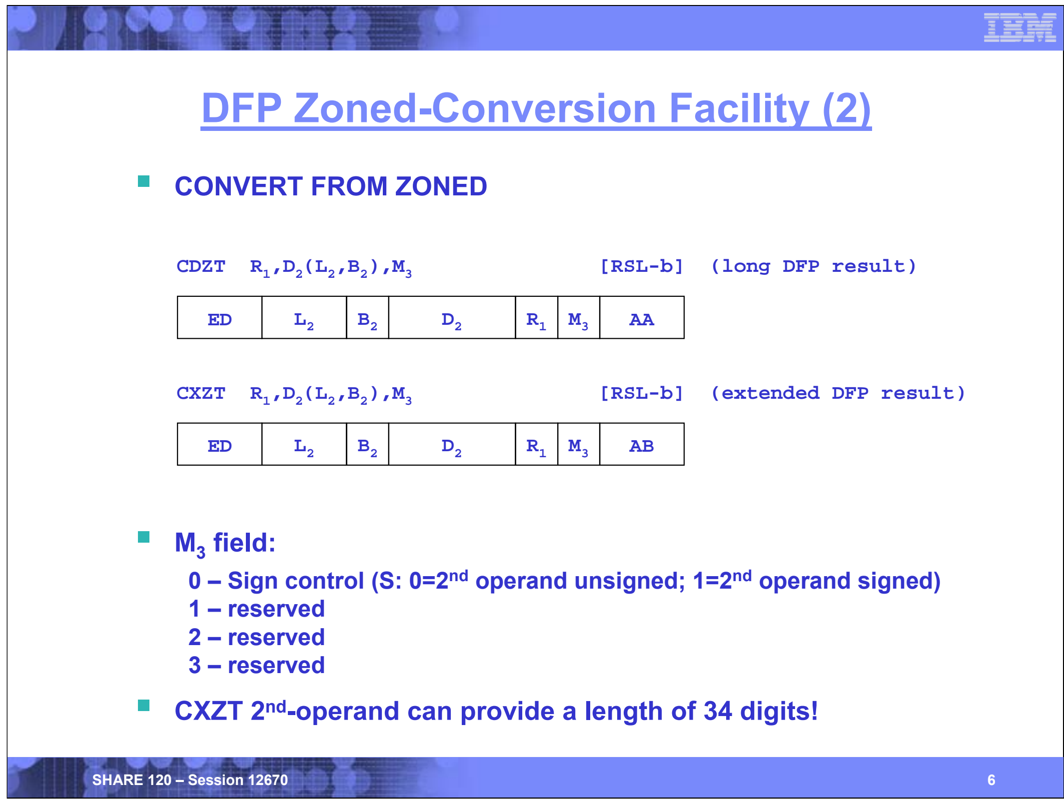
DFP Zoned-Conversion Facility

- Adds instructions for converting between DFP and zoned format
  - May provide substantial performance improvement for applications that use packed-decimal data
  - By converting to DFP and performing calculations using DFP instructions, numerous storage accesses may be avoided
  - Four new instructions:
    - Long / extended DFP format
    - To / from zoned format
  - Facility indication bit 48
- Formally documents zone code 0011 binary (ASCII format)

The DFP zoned-conversion facility provides four new instructions for converting between the zoned-format in storage and a decimal-floating-point (DFP) format value in a floating-point register.

Applications that use zoned and/or packed data formats may yield increased performance by adapting to perform the arithmetic operations in DFP, while retaining the legacy packed or zoned formats.

Also, as a part of this update, the architecture has also been adapted to formally document the zone code of 0011 binary as representing the ASCII numeric zone.



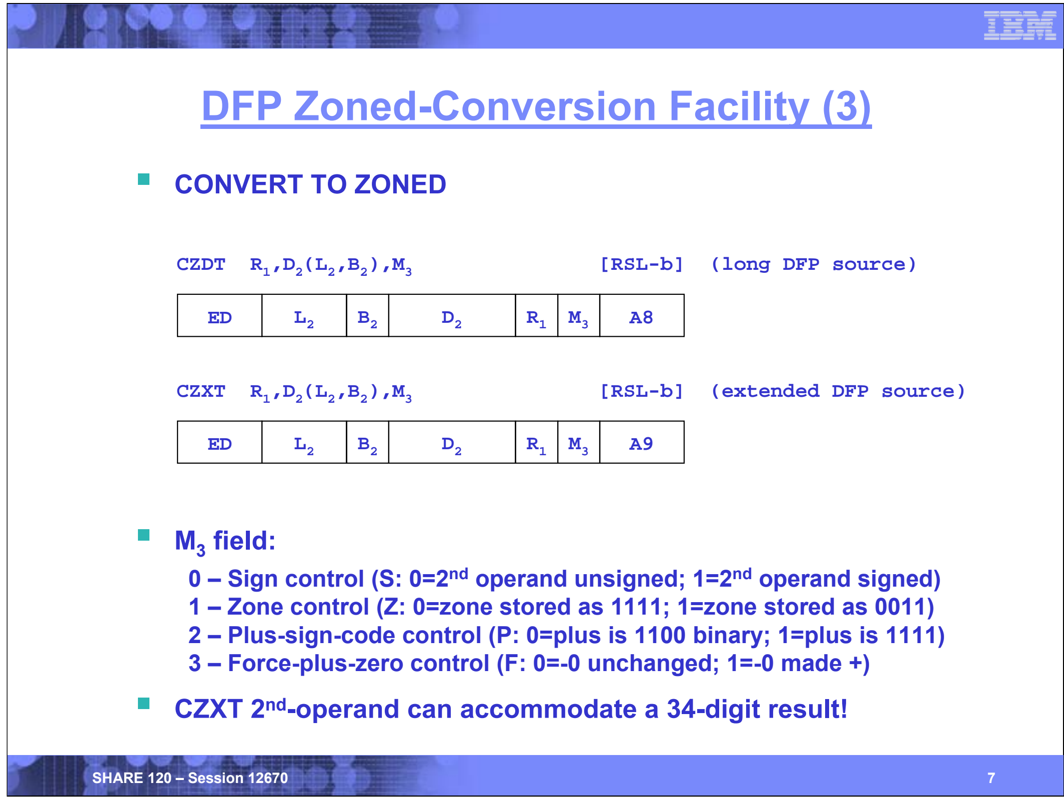
The CONVERT FROM ZONED instructions are of the RSL instruction format (subformat b). There are instructions for converting a zoned value that result in either a long (64-bit) DFP value or an extended (128-bit) value.

The first operand in the R_1 field specifies a floating-point register into which the DFP-format result is placed.

The second operand is the address of a storage-operand containing the zoned value to be converted; the L_2 field indicates the length of the zoned value in bytes.

The M_3 field contains a one-bit sign control which indicates whether the second-operand is to be treated as a signed or unsigned value.

Note, because of the capacity of DFP number representations, CXZT is capable of accommodating a 34-digit length … substantially larger than can be accommodated by normal packed-decimal instructions.



The CONVERT TO ZONED instructions are of the RSL instruction format (subformat b). There are instructions for converting either a long or extended DFP value into a zoned value.

The first operand is the R_1 field specifies a floating-point register containing the DFP number to be converted.

The second operand is the address of a storage-operand into which the zoned result will be placed; the L_2 field indicates the length of the zoned value in bytes.

The M_3 field contains four separate controls:

• Bit zero controls the sign of the result.

• Bit one controls the resulting zone (0 means EBCDIC, 1 means ASCII)

• Bit 2 specifies the encoding of a positive sign value in the result

• Bit 3 indicates whether a DFP -0 value should be made positive.

Note, because of the capacity of DFP number representations, CXXT is capable of accommodating a 34-digit length … substantially larger than can be accommodated by normal packed-decimal instructions.

Execution-Hint Facility

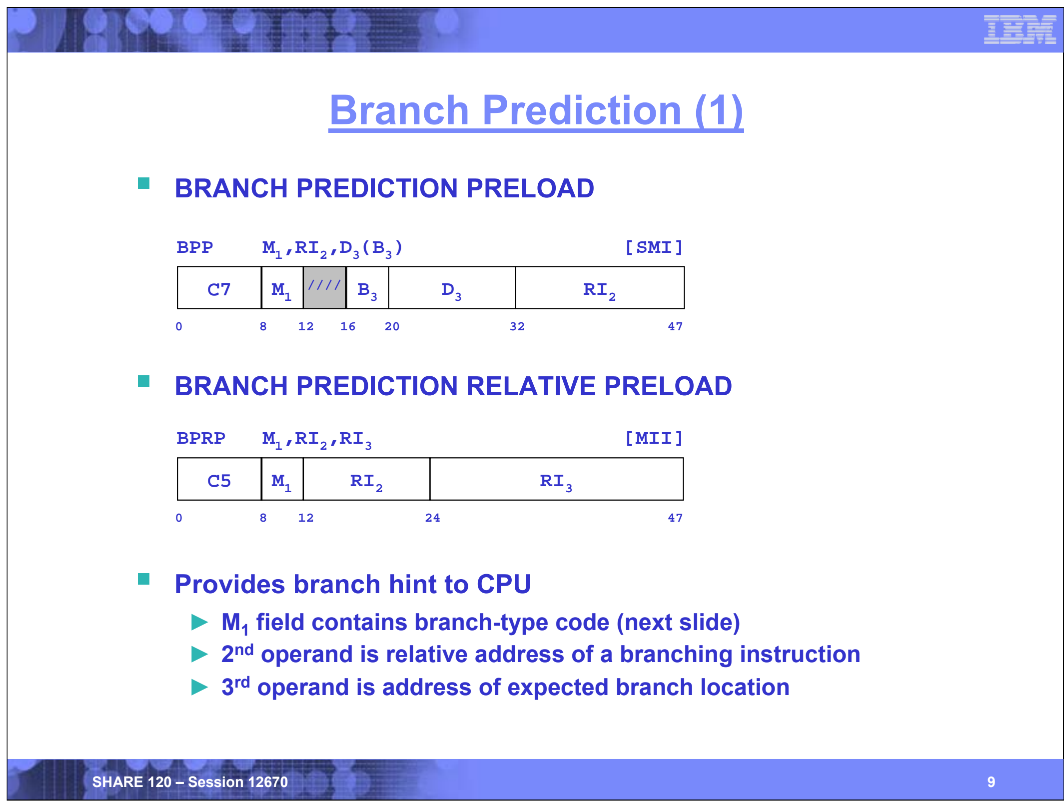
- Provides the following instructions:
  - BRANCH PREDICTION PRELOAD
  - BRANCH PREDICTION RELATIVE PRELOAD
  - NEXT INSTRUCTION ACCESS INTENT
- When the facility is installed, these instructions provide hints to the CPU as to anticipated branches and operand accesses
  - May provide performance improvement (if used properly)
  - May degrade performance (if abused)
  - Otherwise, instructions act as no-ops, and do not affect conceptual sequence of execution.
- Facility indication bit 49

The execution-hint facility provides three instructions which can be used to provide hints to the CPU as to various branching conditions and the storage-access intent of a subsequent instruction. These instructions are used by IBM compilers to optimize instruction flow in the CPU pipeline.

When properly used, these instructions may improve performance. However, when improperly used, these instructions may actually degrade performance by mis-directing CPU branch prediction logic and prefetching controls.

However, regardless of how the instructions are used, they otherwise act as no-operation instructions (no-ops), and do not affect the logic of program execution.

This facility – as well as several others in the zEC12 – are indicated by facility indication 49 (as stored by STORE FACILITY LIST EXTENDED).



BRANCH PREDICTION PRELOAD and BRANCH PREDICTION RELATIVE PRELOAD are instructions to provide the CPU's branch-prediction logic with a direct assertion of the programmer's intent for a branch instruction.

In both instructions:

• The M_1 field contains a code designating the type of branch instruction designated by the second operand (see the next slide for details).

• The RI_2 field contains a signed relative-immediate address (relative to the PSW instruction address) that designates the branch instruction.

The third operand designates the anticipated branch location of the instruction designated by the second operand. For BPP, the third operand is a classic base-and-displacement form, and for BPRP, the third operand is a 24-bit (!) signed value that is relative to the PSW instruction address.

Branch Prediction (2)

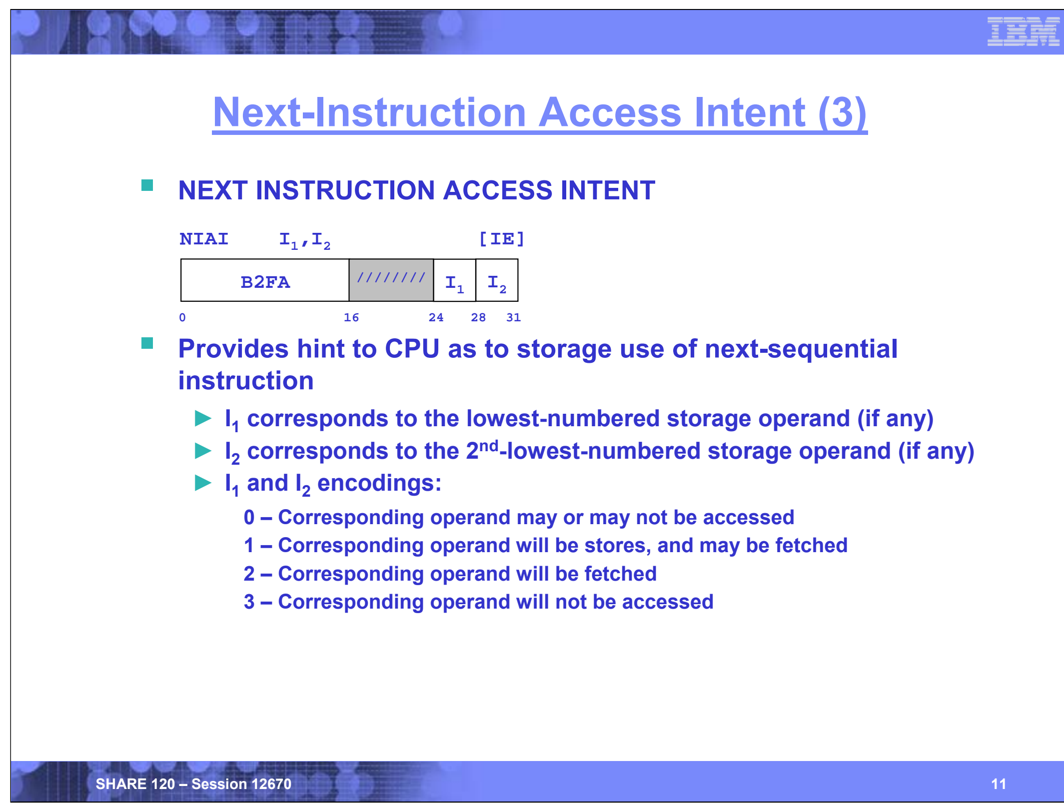
BRANCH PREDICTION PRELOAD M_1 Codes

M_1 Code	Inst. Leng	Corresponding Branch Instruction (designated by RI_2 field)	Usage
0	4	BC	Branch table
1-4	--	Reserved	--
5	2	BALR, BASR, BCR	Static calling linkage
6	2	BCR	Returning linkage
7	2	BALR, BASR, BCR	Dynamic calling linkage
8	4	BC, BCT, BRXH, BRXLE, BXH, BXLE, BRC, BRCT, BRCTG, BCTGR	Cond. or uncond. branches
9	4	BAL, BAS, BRAS	Static calling linkage
10	4	BC	Uncond. return linkage
11	4	BAL, BAS	Dynamic calling linkage
12	6	BRCTH, BRCL, BCTG, BXHG, BXLEG, BRXHG, BRXLG, CGRJ, CLGRJ, CRJ, CLRJ, CGIJ, CLGIJ, CIJ, CLIJ, CGRB, CLGRB, CRB, CLRB, CGIB, CLGIB, CIB, CLIB	Conditional or unconditional branches
13	6	BRASL	Static calling linkage
14	4	EX	
15	6	EXRL	

This slide enumerates the code values that may be specified in the M_1 field.

Note that the same instruction appears for different codes. For example, BALR, BASR, and BCR are used in both codes 5 and 7, and BCR also appears in code 6. Code 5 indicates that the designated branch instruction is used for calling a subroutine, and the target location is expected to always be a single location. Code 7 also indicates that the designated branch instruction is used for calling a subroutine, but the target location may be dynamically determined by the program. Code 6 indicates a BCR instruction that is used to return from a called subroutine. Similar abstractions appear for codes 9 and 11.

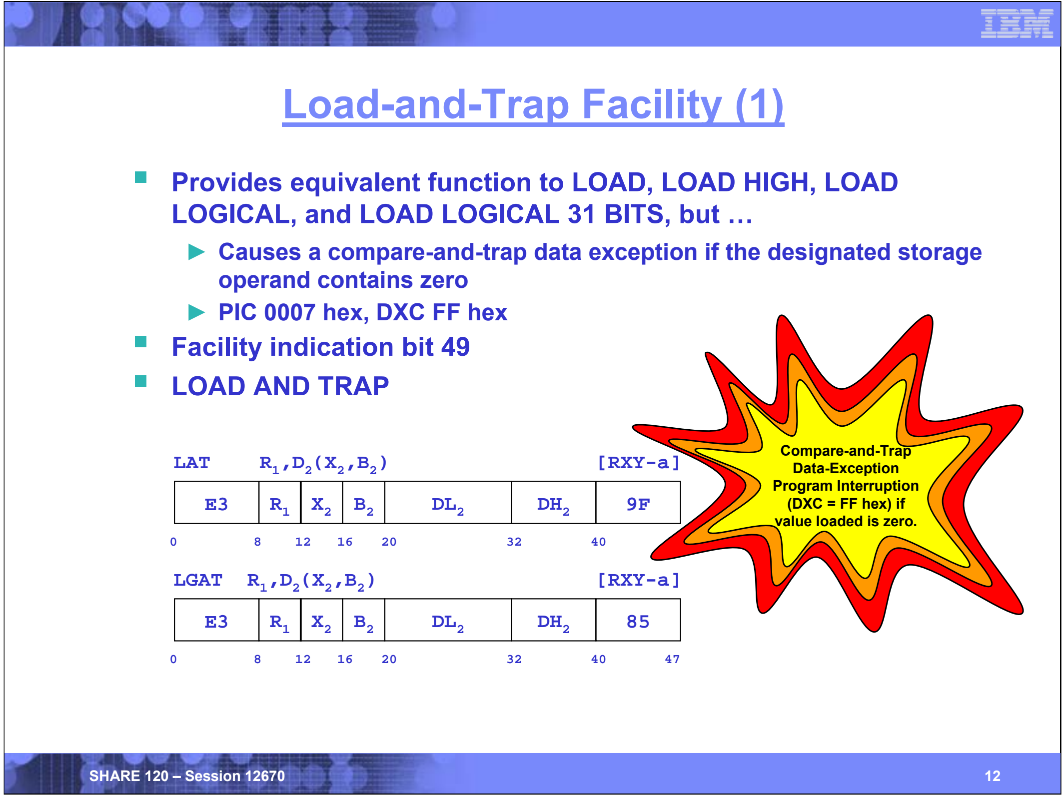
Performance may be degraded if the second operand does not designate a branch instruction that is used in accordance with the M_1 encoding, or if the branch instruction does not branch to the location specified by the third operand.



The NEXT INSTRUCTION ACCESS INTENT instruction provides a means for the program to indicate the anticipated use of the storage locations designated by the next instruction.

The two operands are each 4-bit immediate fields that indicate the anticipated usage of the storage operand(s) of the next sequential instruction. The I_1 field represents the lowest-numbered storage operand, and the I_2 field represents the second-lowest-numbered storage operand (if any). The encodings are listed on the slide.

Performance may be degraded if the subsequent instruction uses its storage operands differently than that specified by the respective I_1 and I_2 fields.



The load-and-trap facility provides a means by which a value can be loaded from storage; if the value contains zero, then a compare-and-trap data exception is recognized. The instructions are equivalent to executing a load instruction followed by a compare-and-trap instruction with a comparand of zero.

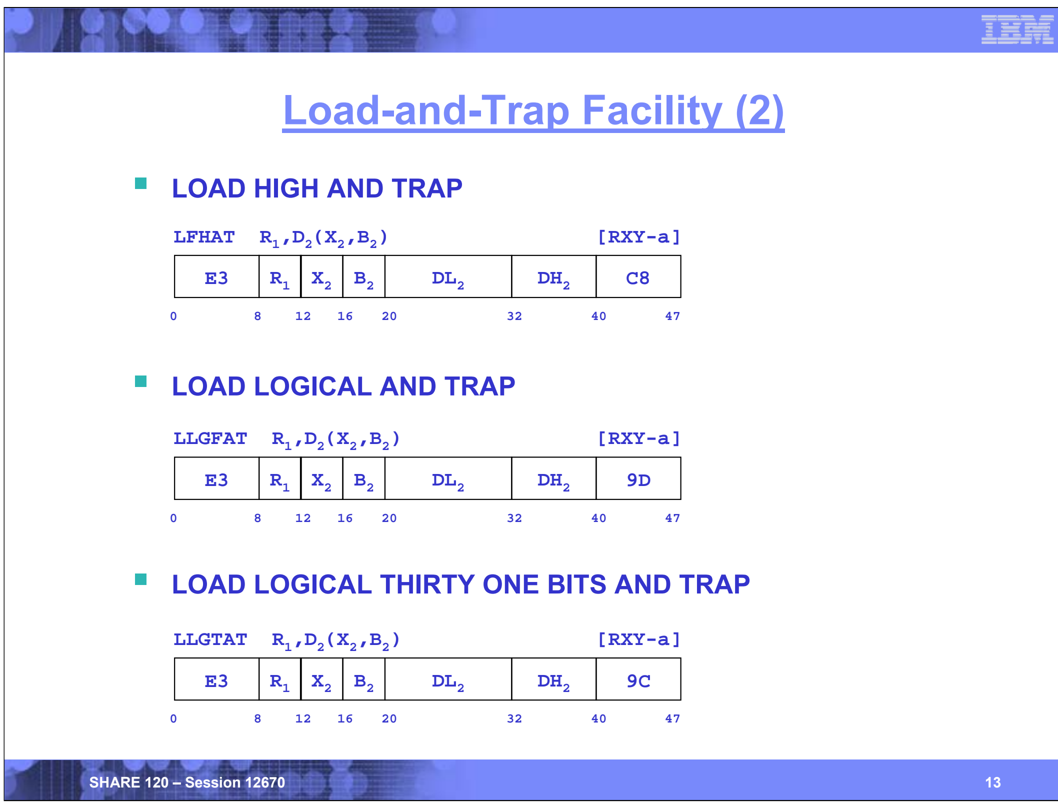
Each instruction is of the RXY instruction format (subformat –a), meaning that the second operand designates a storage location by means of a base register, an index register, and a 20-bit signed displacement field (that is, a long displacement). The result is loaded into the register designated by the R_1 field.

• LAT loads a 32-bit value into bits 32-63 of the register, and leaves the remaining bits unchanged.

• LGAT loads a 64-bit value into bits 0-63 of the register.

In either case, if the value loaded is zero, then a compare-and-trap data exception is recognized. The program-interruption code is 0007, and the data-exception code is FF hex.

This facility – as well as several others in the zEC12 – are indicated by facility indication 49 (as stored by STORE FACILITY LIST EXTENDED).

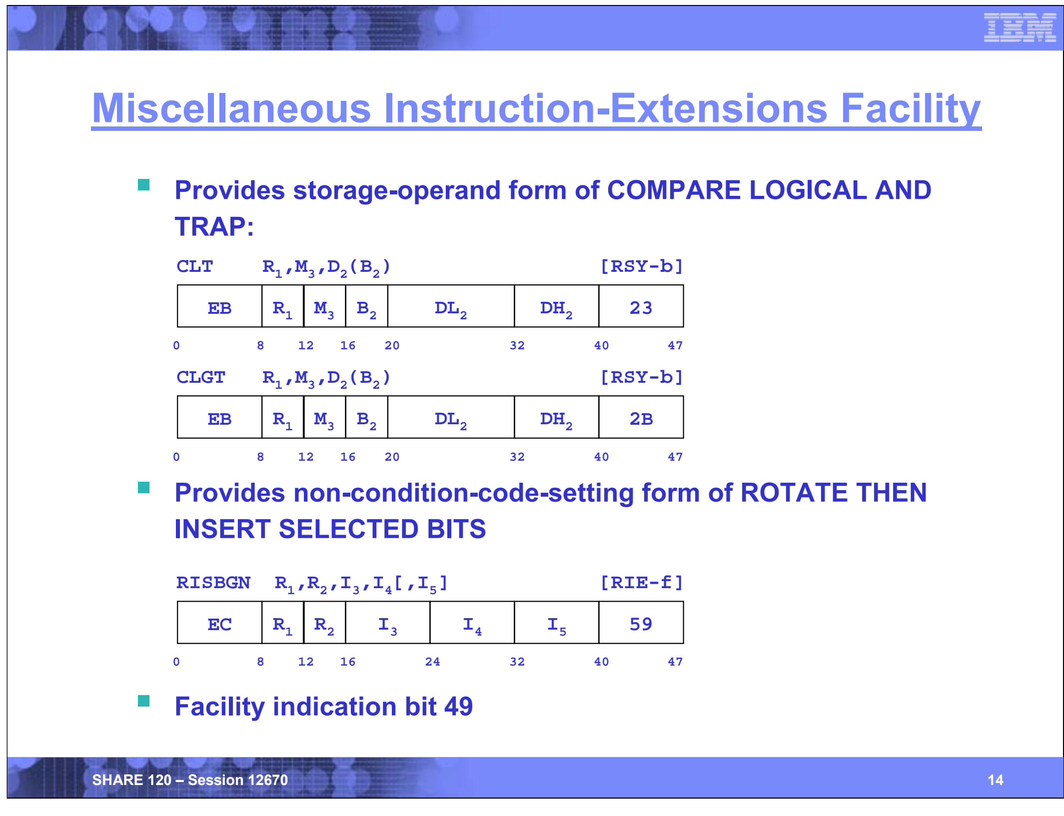


Continuing with the load-and-trap facility:

• LFHAT loads a 32-bit value into bits 0-31 of a register, and leaves bits 32-63 unchanged.

• LLGFAT loads a 32-bit value into bits 32-63 of a register, and bits 0-31 of the register are set to zero.

• LLFTAT loads a 31-bit value (bits 1-31 of the four-byte second-operand location) into bits 33-63 of a register, and bits 0-32 of the register are set to zero.

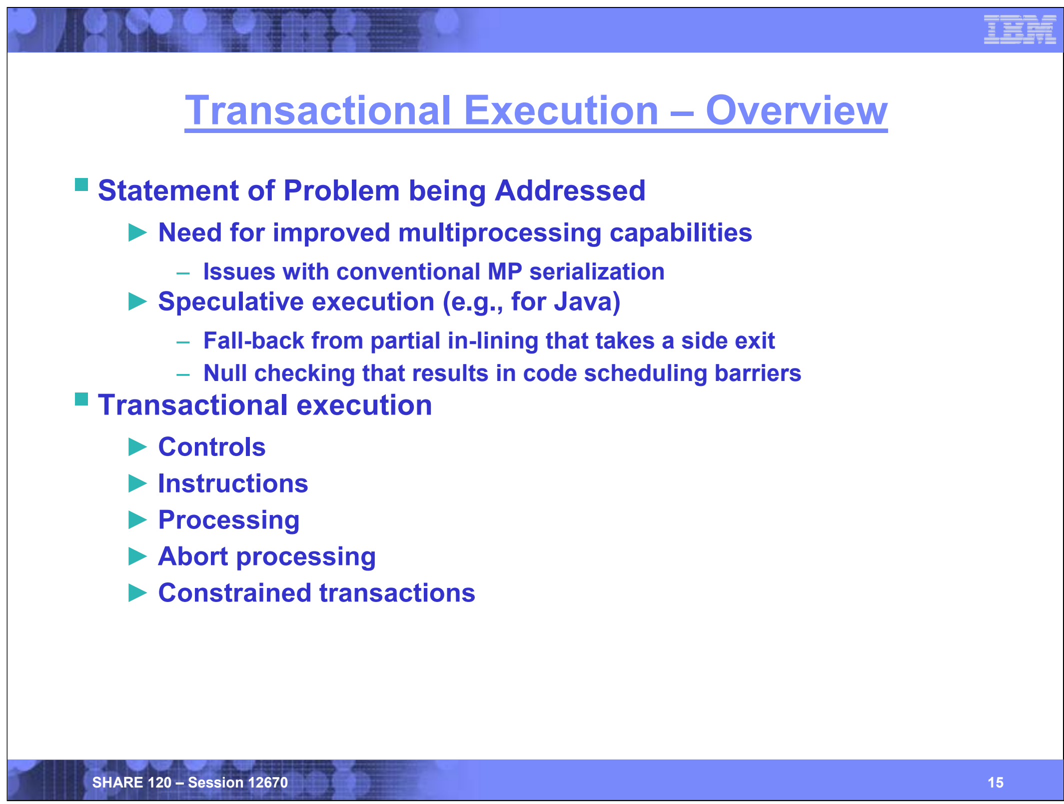


The miscellaneous instruction-extensions facility adds three instructions that are variations of instructions added in the System z10. This facility – as well as several others in the zEC12 – are indicated by facility indication 49 (as stored by STORE FACILITY LIST EXTENDED).

The z10 added the COMPARE LOGICAL AND TRAP instructions, each of which compared a value in a register with either another register or with an immediate field in the instruction. The two new forms of the instruction compare a value a register with a storage location; otherwise the operation is identical to the existing COMPARE LOGICAL AND TRAP instructions.

Both CLT and CLGT are of the RSY instruction format (subformat –b). The first operand contains either a 32-bit (CLT) or 64-bit (CLGT) value in a general register which is compared with a 4- or 8-byte operand in storage. The second operand is designated by a base register and a 20-bit signed displacement field. The trap conditions are specified by the 4-bit M_3 field (similar to the branch mask of branching instructions). Note, HLASM provides extended mnemonics for these instructions, similar to the existing extended mnemonics for compare-and-trap instructions.

ROTATE THEN INSERT SELECTED BITS (RISBG) is one of the most powerful instructions in the CPU. Added in the z10, it provides the means of rotating and extracting bits from a register, and setting the condition code based on the result. Compiler- and system-development groups found that having the condition code set was not always optimal, so a separate instruction, RISBGN, was developed which does not set the CC. Otherwise, execution is identical to the original RISBG instruction.



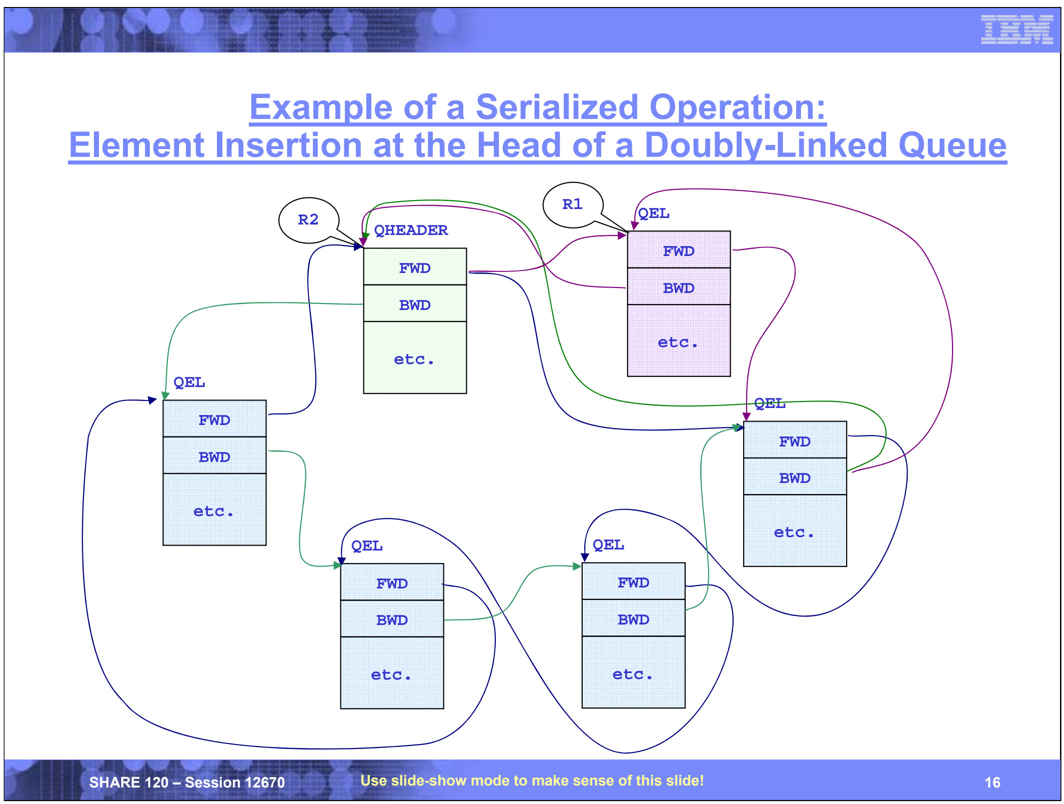
The majority of the slides in this presentation describe the transactional-execution facility (or TX facility, for short).

We begin with a description of two very different problems being addressed by the facility:

• The need for improved multiprocessing capabilities that address limitations in existing serialization, and

• The means by which a program can speculatively execute a code path, and – based on observed state of the program or exceptions encountered – efficiently withdraw said execution, making it appear as if the execution never occurred.

The following slides will discuss the controls, instruction, processing, abort handling, and a special form of TX called a constrained transaction.



This slide illustrates a doubly-linked list into which a new queue element is to be inserted at the head of queue. The queue header (shown in green) and the existing queue elements (shown in blue) each contain a forward and backward pointer. Thus, in order to insert an element into the queue, multiple discontiguous storage locations must appear to be simultaneously updated (as observed by other CPUs and the channel subsystem) in order for the queue to retain integrity.

• The forward pointer of the queue header must be updated to point to the newly-inserted element.

• The backward pointer of original first element must be updated to point to the newly-inserted element.

• The forward pointer of the inserted element must point to original first element on the queue.

• The backward pointer of the inserted element must point queue header.

In order to maintain the integrity of the queue, the program will usually acquire some form of serialization such as a lock (also known as a semaphore or mutex). This technique is illustrated on the following slide.

Example of a Serialized Operation: Sample Code Fragment using Locks

```
* R1  - address of the new queue element to be inserted.
* R2  - address of the insertion point (i.e., head of queue).

NEW      USING  QEL,1              Make new 1st QEL addressable.
HDR      USING  QEL,2              Make queue header addressable.
OLD      USING  QEL,3              Make old 1st QEL addressable.

         SETLOCK OBTAIN, …         Serialize access to queue.
         LG     3,HDR.QEL_FWD      Point to original 1st element.
         STG    1,HDR.QEL_FWD      Update header's forward pointer.
         STG    1,OLD.QEL_BWD      Update orig. element's back ptr.
         STG    2,NEW.QEL_BWD      Update new element's backward ptr.
         STG    3,NEW.QEL_FWD      Update new element's forward ptr.
         SETLOCK RELEASE, …

         …

QEL      DSECT                     Common DSECT for header or QEL.
QEL_FWD  DS     AD                 Forward pointer.
QEL_BWD  DS     AD                 Backward pointer.
         DS     XL48               Queue element payload.
```


This assembler programming example shows how the update to these four storage locations can be accomplished using classic locking mechanisms.

Shown in the first highlighted section of code, the SETLOCK OBTAIN macro instruction is used to illustrate any number of locking, semaphore, or other serialization techniques that may be used to ensure that only one CPU is executing the following code fragment at any one time.

After obtaining the serialization, the program loads the address of the original first element on the queue into general register 3, and then proceeds to update the four key objects needed to insert the new element.

Finally, after performing the update, the SETLOCK RELEASE macro instruction (in the second highlighted section) illustrates the releasing of the serialization.

Note: In this example, because the forward and backward pointers appear in the same location in both the queue header and queue element, a single DSECT (QEL) is used.

Problems with Conventional Serialization:

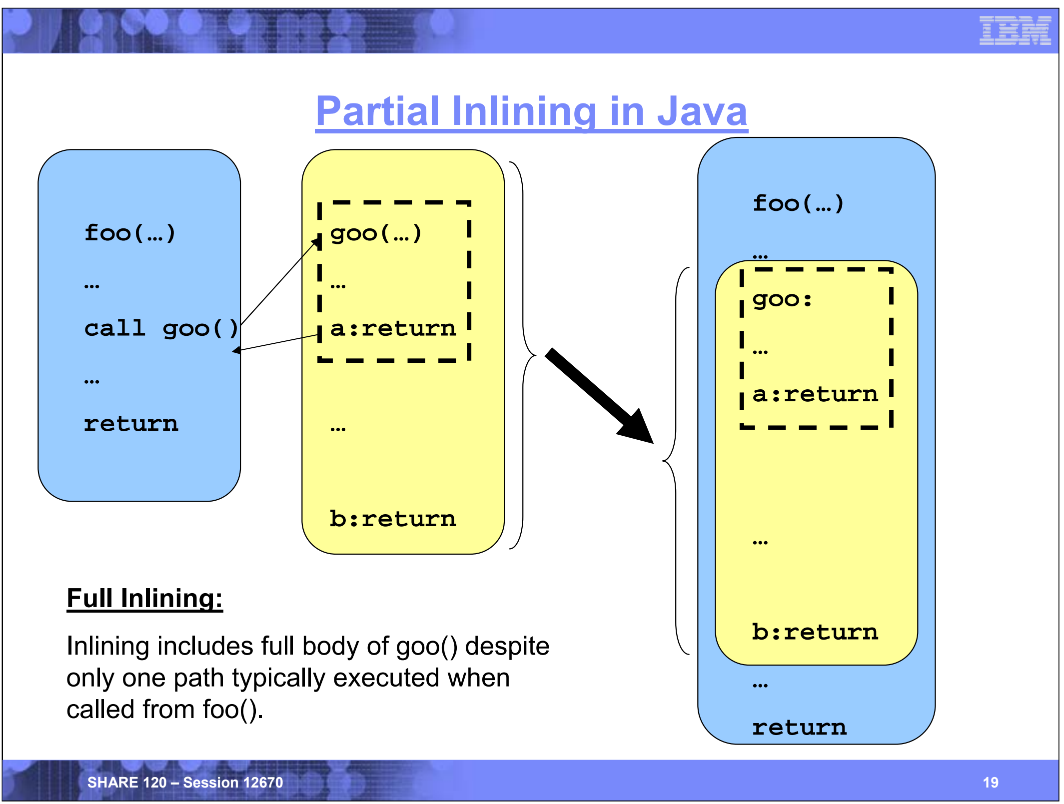
- Coarse-grained locking
  - Usually require serializing a much broader resource than what is being actually being accessed
  - With finer-grained serialization, multiple locks may be required
    - Hierarchy issues, potential dead-locks
    - Natural access may not lend itself to imposed hierarchy
- Various recovery issues
  - Lock cannot be acquired in a timely manner
  - Unexpected event encountered while locked

The problem with using classic locking techniques is that, in general, they serialize a much broader scope of resources than is actually being accessed. For example, in the queue illustration, the queue may contain millions of elements, yet only one is being updated. Even in a multiprocessing environment, many such data structures are serialized by coarse-grained lock, when it is rare to have multiple CPUs update the same location. (Nonetheless, the rare case must always be handled properly.)

Finer-grained serialization may exploit multiple levels of locking, but with such a hierarchy, there is the issue of potential deadlocks if the locks are not acquired and released in the proper sequence. Furthermore, finer-grained serialization imposes a regimen on the program that may be more complicated and error prone.

Additionally, the program must accommodate scenarios where either (a) the lock cannot be obtained, and (b) a task holding a lock encounters an unexpected condition (for example, abnormal end). Often the occurrence of “b” results in “a.”

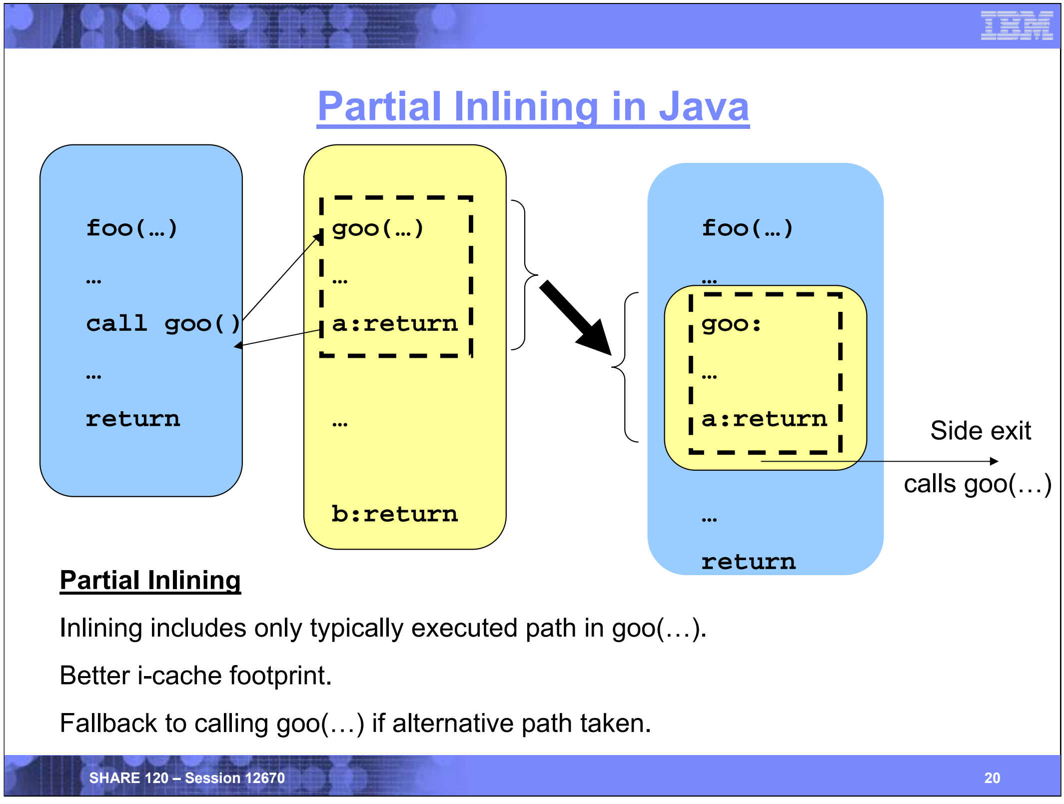
Many years ago, IBM developed the PERFORM LOCKED OPERATION (PLO) instruction which provided a means by which separate storage locations could be updated under serialization provided by a configuration-wide lock. However, PLO does not co-exist well with classic forms of serialization that use compare-and-swap types of updates.



This slide illustrates a process that Java exploits to improve the performance of certain function calls.

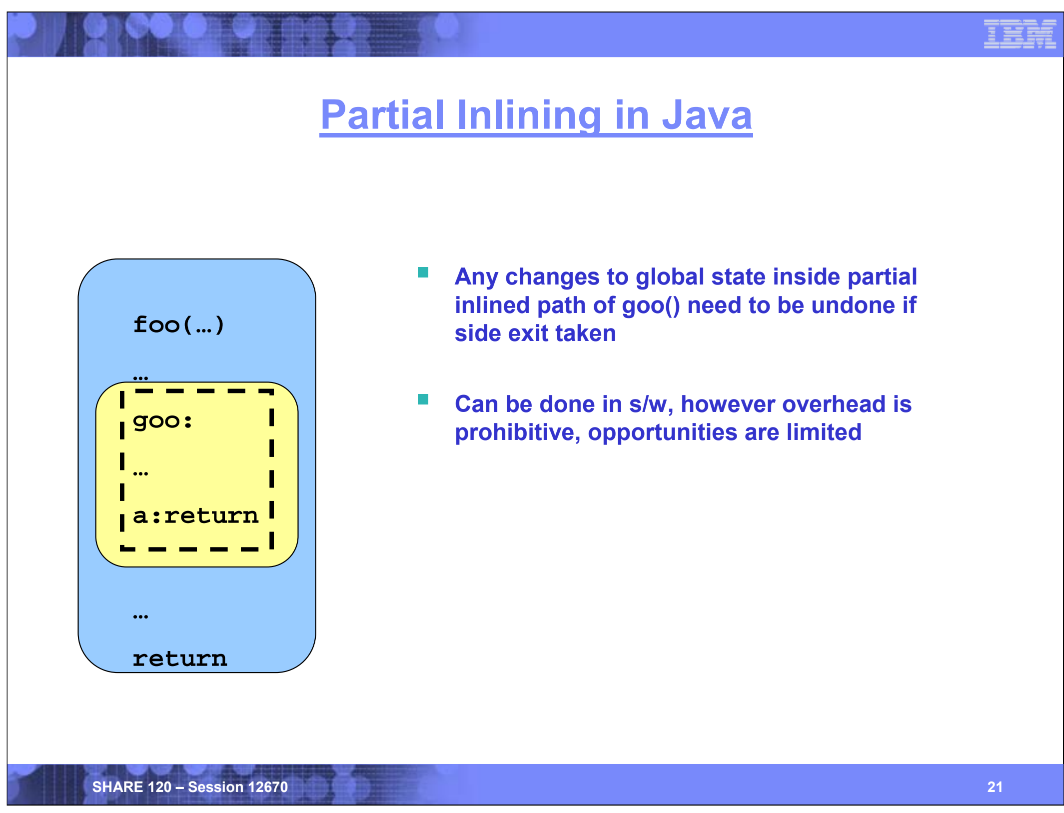
On the left, we see the function foo() calling function goo(). Function goo() has a normal sequence of execution which returns at location “a,” and an alternate sequence of execution which returns at location “b.”

Java may restructure the functions such that a copy of the called function goo() is contained within the calling function foo(); this operation is called in-lining, and may be beneficial in minimizing instruction-cache references. If Java performs full in-lining of function goo(), it incorporates the function in its entirety inside the calling function foo(). This has the disadvantage of making foo() larger than it needs to be, increasing its instruction-cache footprint.

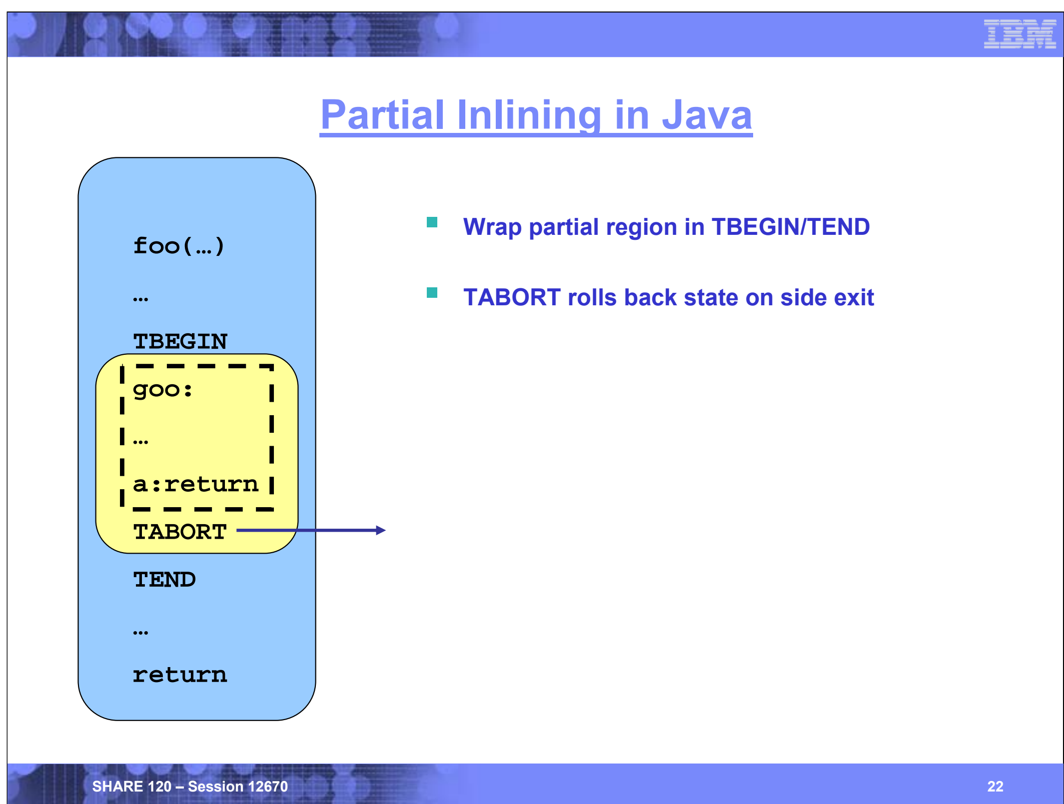


In this slide, we illustrate the partial in-lining of function goo(). Only the commonly-executed sequence of code is placed into the in-lined version. This yields a better instruction-cache footprint.

However, if the execution of the partially-in-lined version of goo() determines that it needs to execute the alternate code sequence, it must call the full version of goo() to execute that code sequence.



The problem with partial in-lining of the function goo() is that if the alternate code path must be called, then any state changes made by the in-line version must be undone. This can be accomplished in software, however (a) the overhead is prohibitive, and (b) it significantly increases the complexity of code generation.



This code fragment illustrates the bracketing of the partially-in-lined version of goo() with new instructions that are part of the transactional-execution facility: TBEGIN and TEND.

While executing the partially-in-lined version of goo() within a transaction, any changes to storage are not visible to other CPUs and the I/O subsystem until the TEND instruction completes.

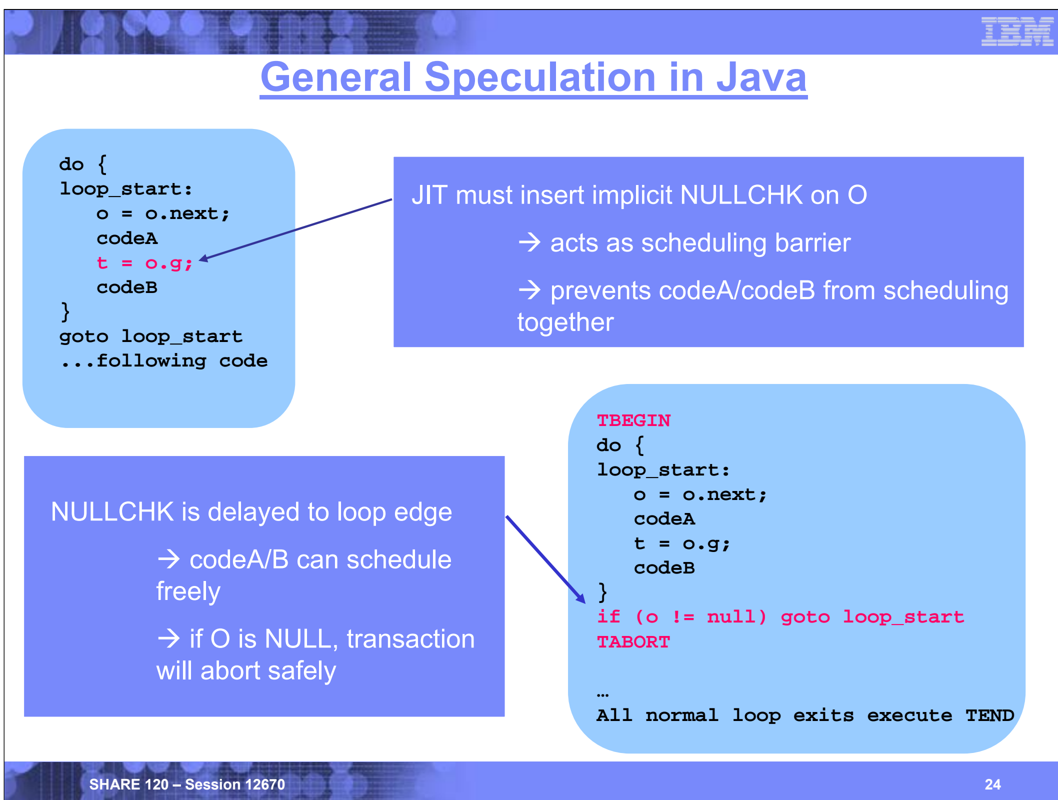
Alternatively, if the function detects a situation which requires the calling of the full goo() function, it can execute a TABORT instruction to deliberately abort the transaction. In this case, all transactional stores made during transactional execution are withdrawn, as if they never occurred.

General Speculation in Java

- Java imposes implicit NULLCHKs on de-referenced pointers
- NULLCHKs are strongly ordered with respect to other global state changes and exception checks
- Strong ordering acts as a pipeline code scheduling barrier

Another characteristic of Java is that it imposes strongly-ordered null checking on dereferenced pointers. That is, the checking must appear to occur before any use of the pointer is attempted.

This ordering does not necessarily lend itself to efficient scheduling of instructions in the pipeline. As we will see in the next slide, transactional execution provides a means of evading this strong ordering requirement.



In the code sequence shown in the upper left, two code fragments, codeA and codeB are shown. In between the two fragments, a pointer "O" is dereferenced to fetch the member "g".

Ordinarily, Java would have to insert a check of "O" to ensure it was not null, before attempting to execute codeB.

However, in the example on the lower right, Java can bracket this code sequence with a transaction. In this case, the checking for a null value of "O" can be deferred, such that codeA and codeB can be scheduled more efficiently. Subsequently, the value of "O" can be checked, and if null, cause the entire code sequence to be aborted.

The vast majority of the time, one expects that a non-null value of "O" will be used, thus the transaction will not be aborted. In the rare case of an abort, the abort-handler code can deal with the null pointer.

Transactional-Execution (TX) Mode

- New CPU state:
  - Introduced in the zEnterprise EC-12 CPU architecture
  - Initiated by TRANSACTION BEGIN instruction
  - Ended by either:
    - Outermost TRANSACTION END (TEND) instruction
    - Transaction abort
- While in the transactional-execution mode:
  - All storage accesses by the CPU appear to be block-concurrent to other CPUs and the channel subsystem
  - Transactional store accesses are either:
    - Committed to storage when the outermost transaction ends normally (via TEND), or
    - Completely abandoned if the transaction is aborted

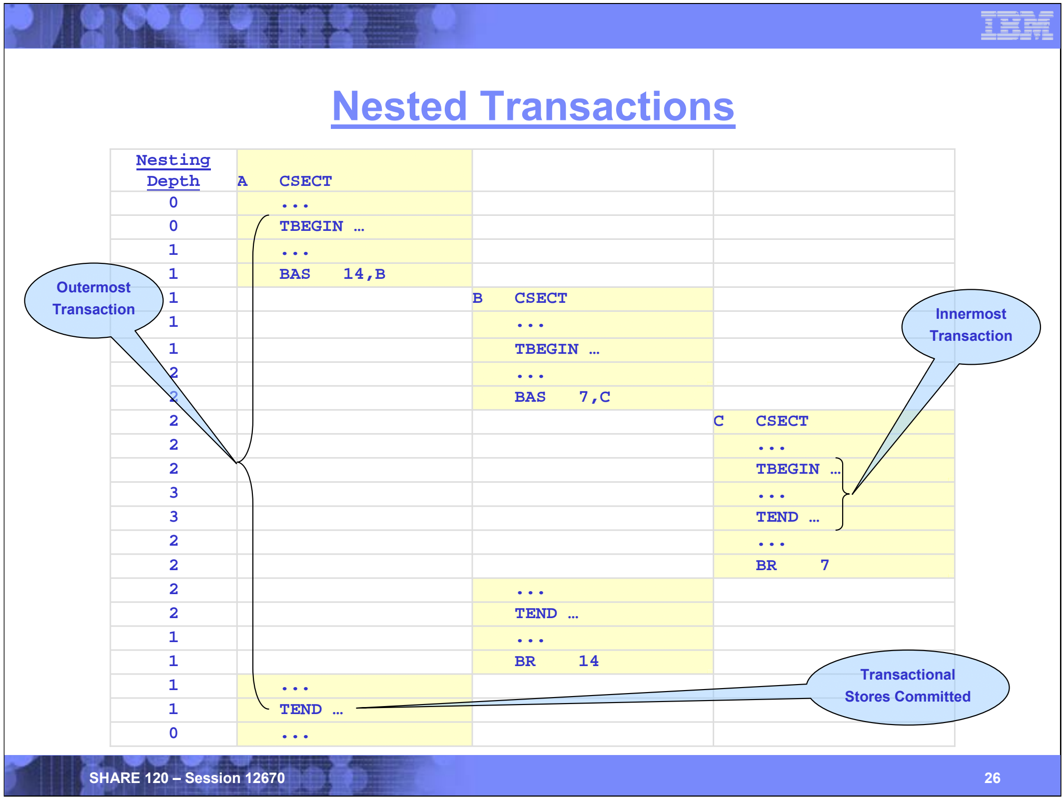
The transactional-execution (TX) facility adds a new CPU state to the processor – the transactional-execution mode.

TX mode is started by an outermost TRANSACTION BEGIN instruction. The term outermost is used, because transactional execution can be nested (as shown on the following slide). TX mode is ended by either of the following:

• An outermost TRANSACTION END instruction being executed, or

• The transaction being aborted.

While the CPU is in the TX mode, all storage accesses by the CPU appear to be block concurrent (that is, they happen all at once), as observed by other CPUs and by the I/O subsystem. These transactional stores are either (a) committed to storage and made visible to other CPUs when the outermost TRANSACTION END instruction completes, or (b) completely abandoned if the transaction is aborted.



This slide illustrates the nesting of transactions, and the transaction nesting depth (TND).

Initially, when the CPU is not in the TX mode, the nesting depth is zero, as shown in the column on the left.

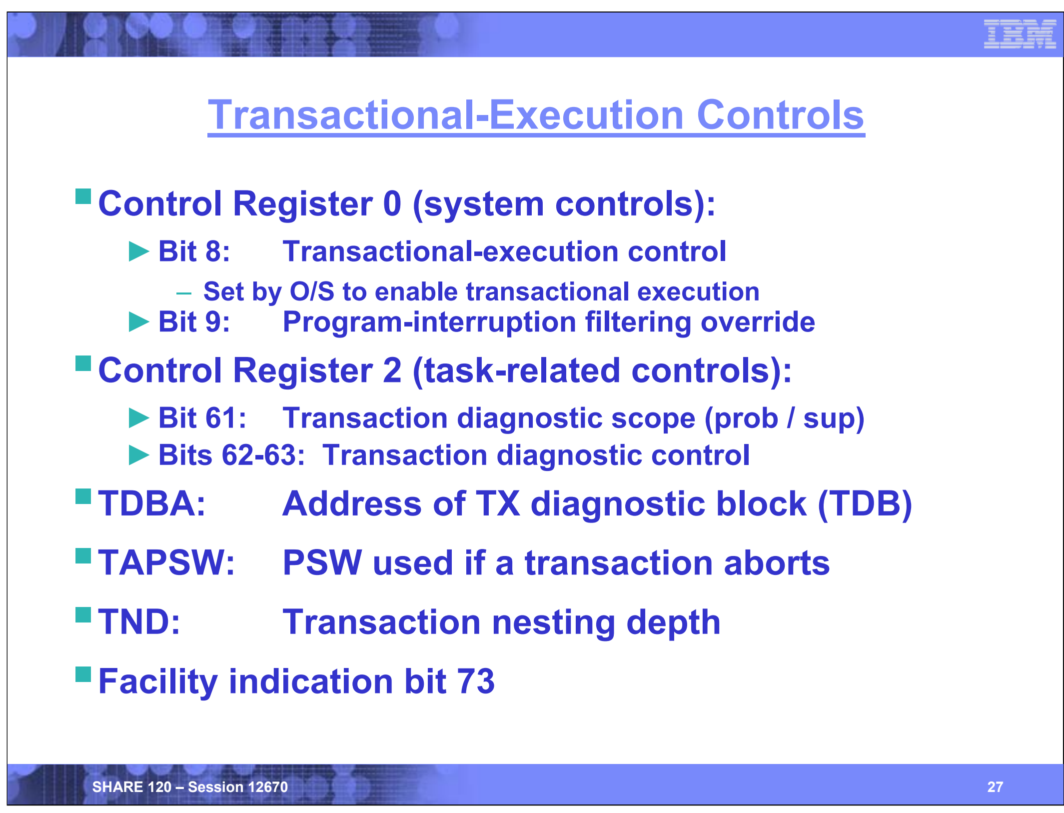
When control section A executes the TBEGIN instruction (that is, the outermost TBEGIN), the CPU enters the TX mode, and the transaction nesting depth (TND) is set to one. Control section A then calls control section B.

CSECT B also contains transactionally-executed code. When CSECT B executes its TBEGIN, the nesting depth is incremented to two. CSECT B then calls CSECT C.

As with B, CSECT C also contains transactionally-executed code. The execution of its TBEGIN instruction causes the TND to be incremented to three. CSECT C then ends its transaction with a TRANSACTION END (TEND) instruction. In this case, the CPU remains in the TX mode, but the nesting depth is decremented to two. CSECT C then returns to its caller.

CSECT B also executes a TEND instruction, causing the CPU to remain in the TX mode, but decrementing the nesting depth to one. CSECT B then returns to its caller.

CSECT A also exectues a TEND instruction. This causes the nesting depth to decrement to zero, thus the CPU commits all stores made during transactional execution to memory and then leaves the TX mode.



There are several new controls affecting the transactional-execution facility:

Control register zero contains system-wide controls, as follows:

• Bit 8 indicates that the OS has enabled the TX facility. Because the facility requires OS support, this bit is set to zero by default, such that OS's that do not support TX can execute compatibly on a zEC12.

• Bit 9 is used in conjunction with program-interruption filtering, to be discussed later. The OS can override any program interruption filtering.

Control register two contains task-related controls used in the debugging of a transaction. These bits are set by various system-level debuggers.

• Bit 61 indicates the scope of control for the transaction-diagnostic control in bits 62-63 (that is, whether the control affects only problem state or both problem state and supervisor state).

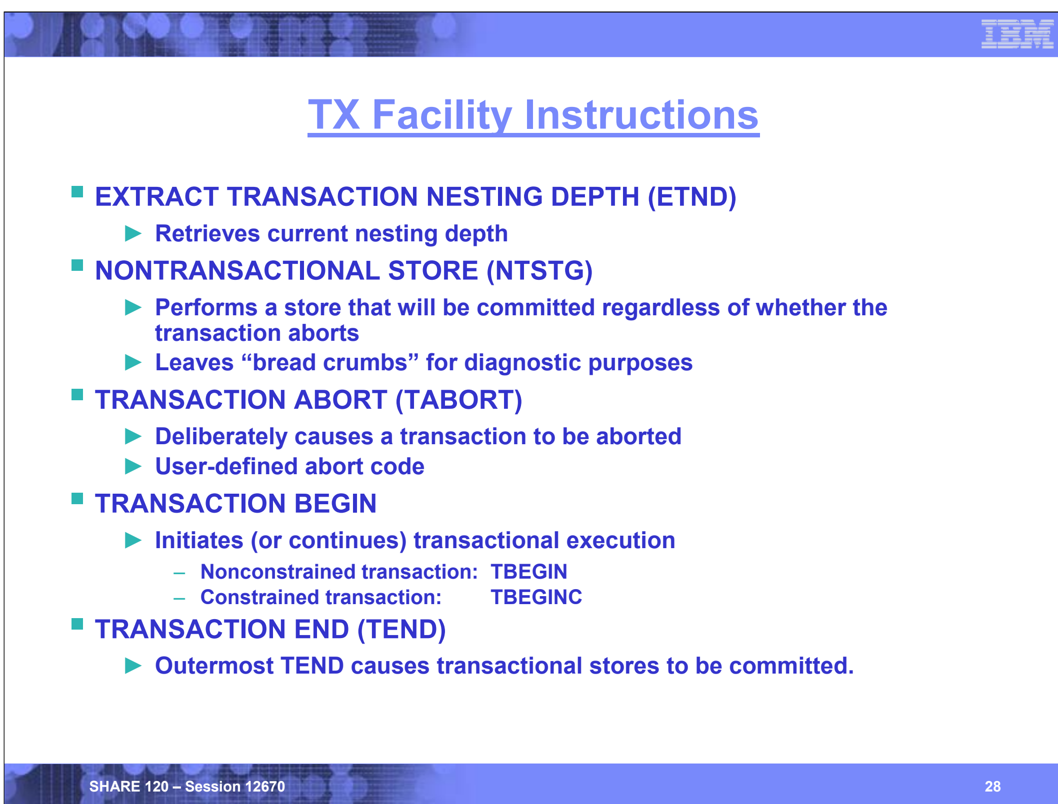
• Bits 62-63 contain a diagnostic control. When nonzero, the diagnostic control causes various random aborts of transactions, thus allowing the testing of their abort handler.

An outermost TBEGIN instruction can specify the address of a transaction diagnostic block (TDB) into which various information is stored if the transaction is aborted. The address of this block is maintained in the TDB address (TDBA).

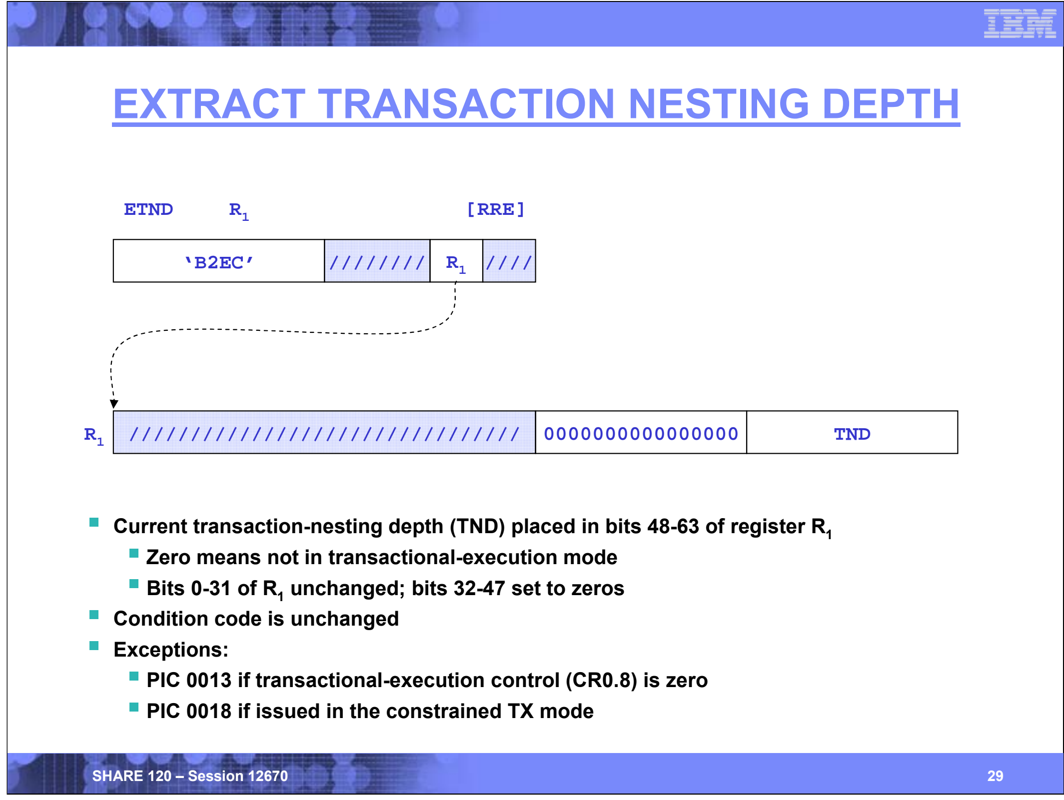
When an outermost transaction begins, it sets a transaction abort PSW (TAPSW) that is loaded if the transaction is aborted. More on this later.

The transaction nesting depth (TND) may be inspected by the program regardless of whether or not the CPU is in the TX mode.

The availability of the TX facility is indicated by facility bit 73 (as stored by STORE FACILITY LIST EXTENDED). Note, even though a processor may provide the transactional-execution facility to logical partitions (LPARs), the facility may not be available when running in a virtual machine under the z/VM operating system.



This slide summarizes the new instructions provided by the TX facility. Each of these will be described in detail in the following slides.

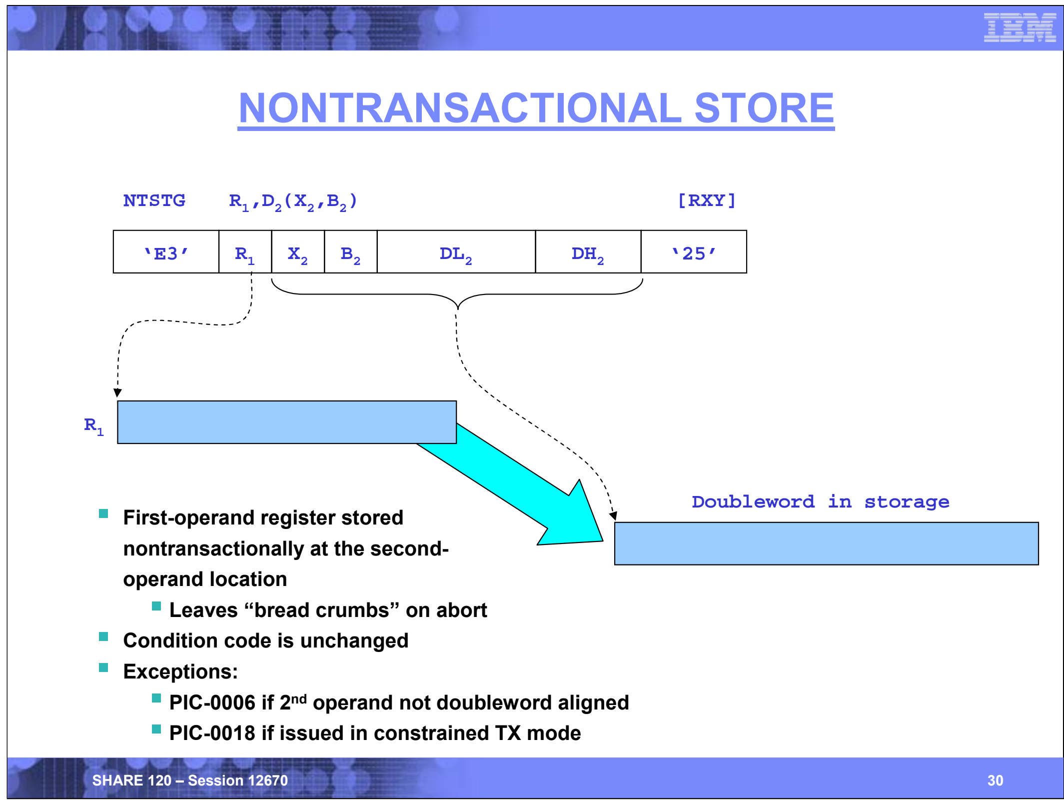


The EXTRACE TRANSACTION NESTING DEPTH (ETND) instruction provides a means by which the program can inspect the current transaction nesting depth (TND).

ENTD is an RRE-format instruction with a single general register operand. The current nesting depth is placed in bits 48-63 of the register, and zeros are placed in bits 32-47; bits 0-31 of the register are unmodified. Also, the condition code remains unchanged.

As will be seen in most of the TX instructions, there is a special-operation exception recognized if the OS has not enabled the TX facility (in CR0.8).

Also described here is a new program exception, the transaction-constraint exception, if the instruction is executed while the CPU is in the constrained TX mode. More on nonconstrained versus constrained towards the end of this presentation.



The astute observer will have noticed that debugging transactional execution may prove to be challenging. This is because when a transaction is aborted, all evidence of transactional stores vanish.

As we will shortly see, there is the possibility of having some diagnostic information retained in registers, however additional information in storage would also be useful.

The NONTRANSACTIONAL STORE (NTSTG) instruction provides a means by which stores are retained following the abort of a transaction.

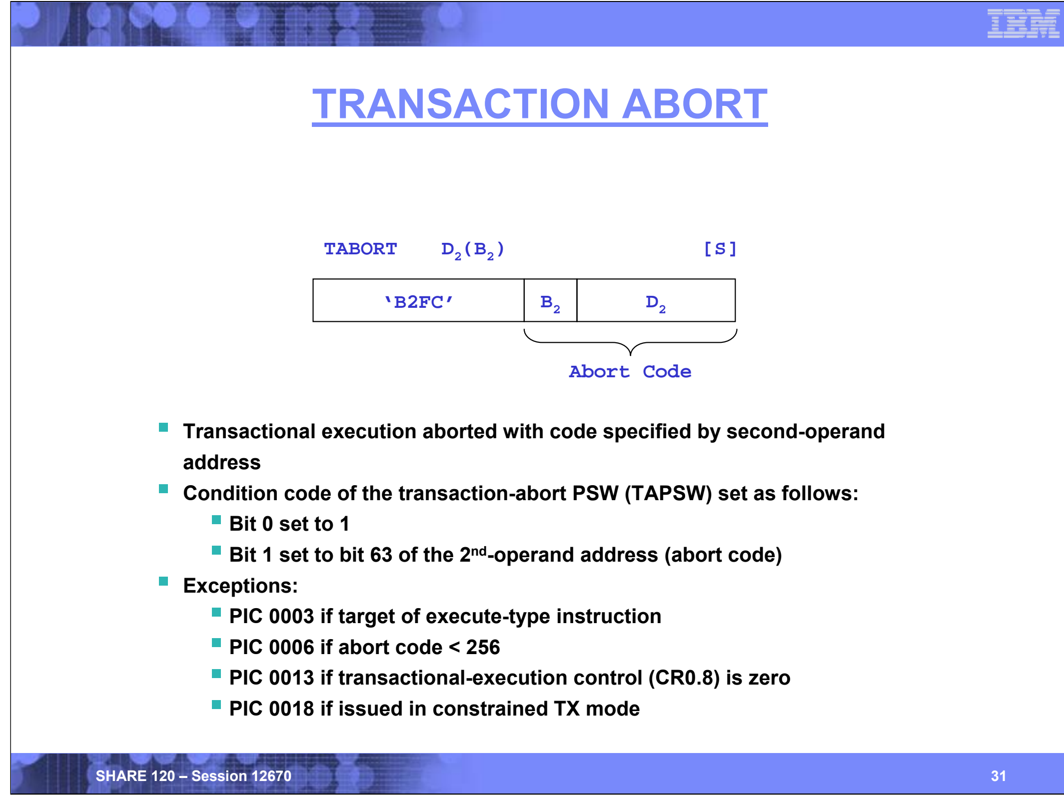
NTSTG is very similar to a regular STORE (STG) instruction. It is an RXY-format instruction, with the first operand designating a 64-bit register to be stored, and the second operand using a base, index, and long displacement to designate a storage location.

NTSTG differs from STG as follows:

• The storage operand must be on a doubleword boundary; otherwise, a specification exception is recognized.

• If NTSTG is executed in the constrained TX mode, a transaction-constraint exception is recognized. Regular STG can be used in the constrained TX mode.

• Stores made by NTSTG are retained if a transaction aborts; stores made by STG disappear.



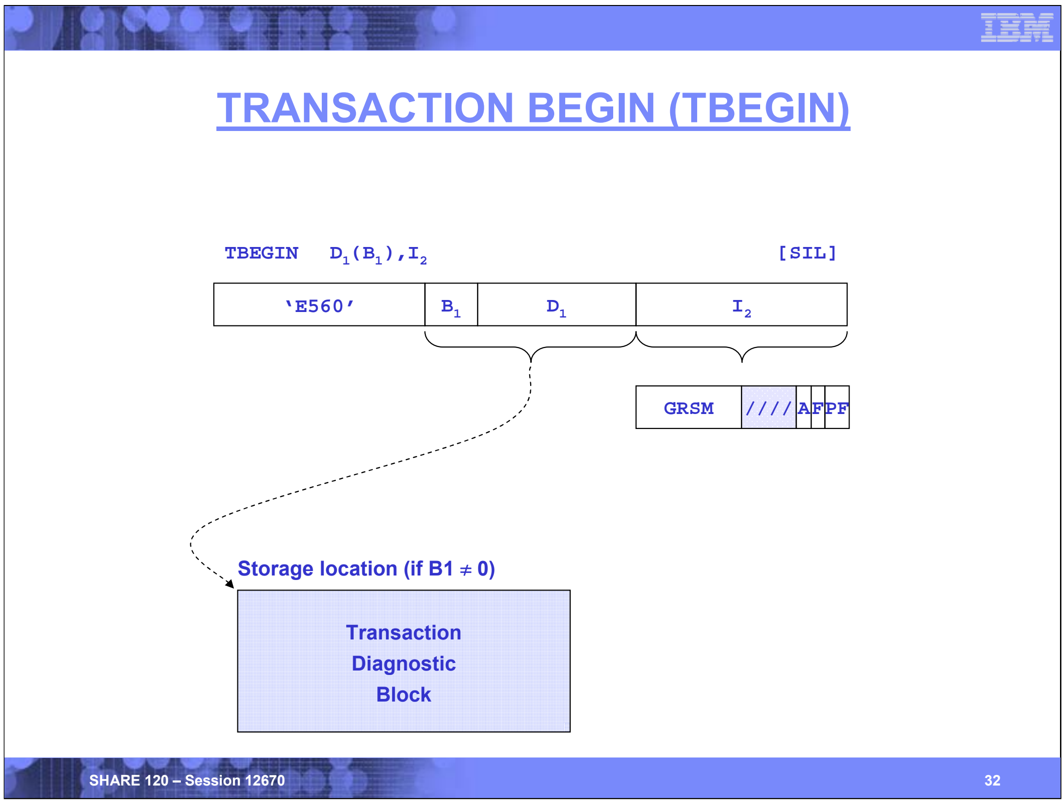
The TRANSACTION ABORT instruction provides the program with the means of deliberately aborting a transaction. It may be used, for example, to implement Java's partial in-lining or null-check reordering, described earlier.

TABORT is an S-format instruction, having a second-operand address designated by a base register and 12-bit unsigned displacement. However the operand address is not used to access storage; rather, the address forms the transaction abort code that is stored into any transaction diagnostic block (TDB) designated by the outermost TBEGIN instruction.

TX architecture reserves abort codes 0-255 for use by the CPU. If the abort code in the second-operand address is less than 256, then a specification exception is recognized.

When a transaction is aborted, the condition code in the transaction-abort PSW indicates the likelihood of successful execution if the transaction is attempted again. CC2 means there is a potential for successful completion, and CC3 means that there is little potential for successful completion. TABORT sets bit 18 of the transaction-abort PSW to one, and bit 19 of the transaction-abort PSW is set to bit 63 of the second-operand address. Thus, the condition code is set to either 2 or 3, depending on whether the rightmost bit of the address is 0 or 1, respectively.

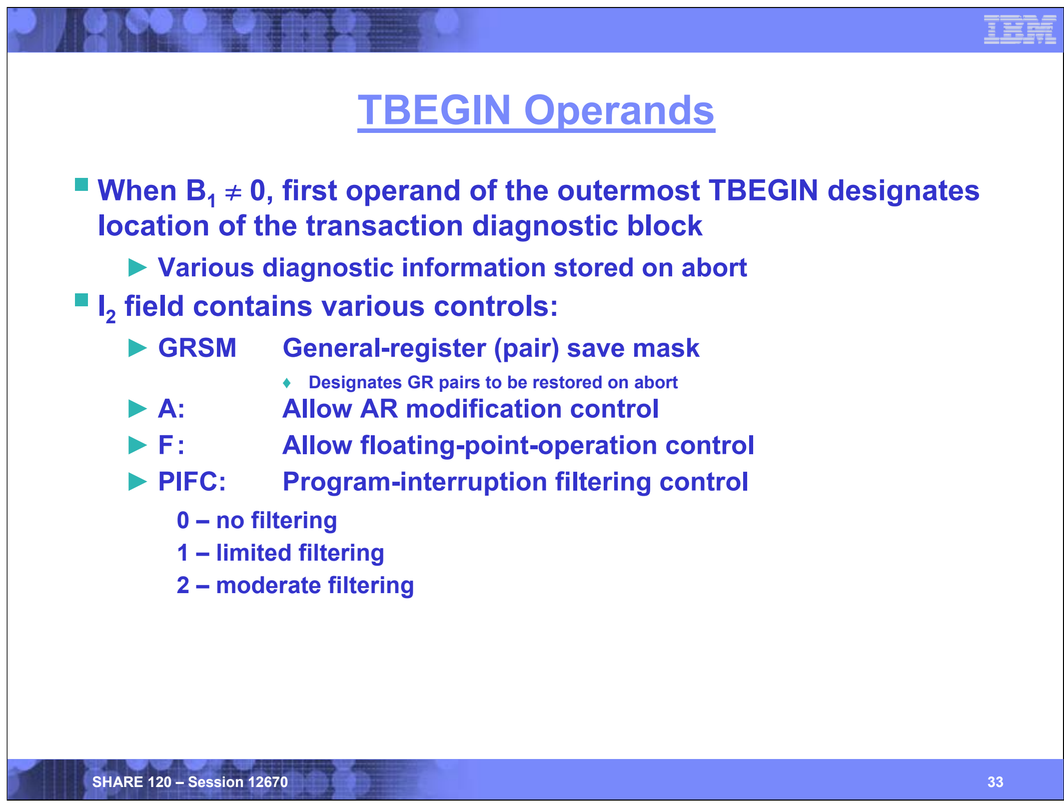
TABORT is unusual in that an execute exception (program interruption code 0003 hex) is recognized if TABORT is the target of an execute-type instruction (EX or EXRL).



The TRANSACTION BEGIN instruction is used to initiate or continue the execution of a *nonconstrained* transaction. We'll discuss nonconstrained versus constrained more in a later section.

TBEGIN is a SIL-format instruction having a storage operand designated by a base register and 12-bit unsigned displacement, and an immediate field containing various controls.

The operands are described further on the next slide.



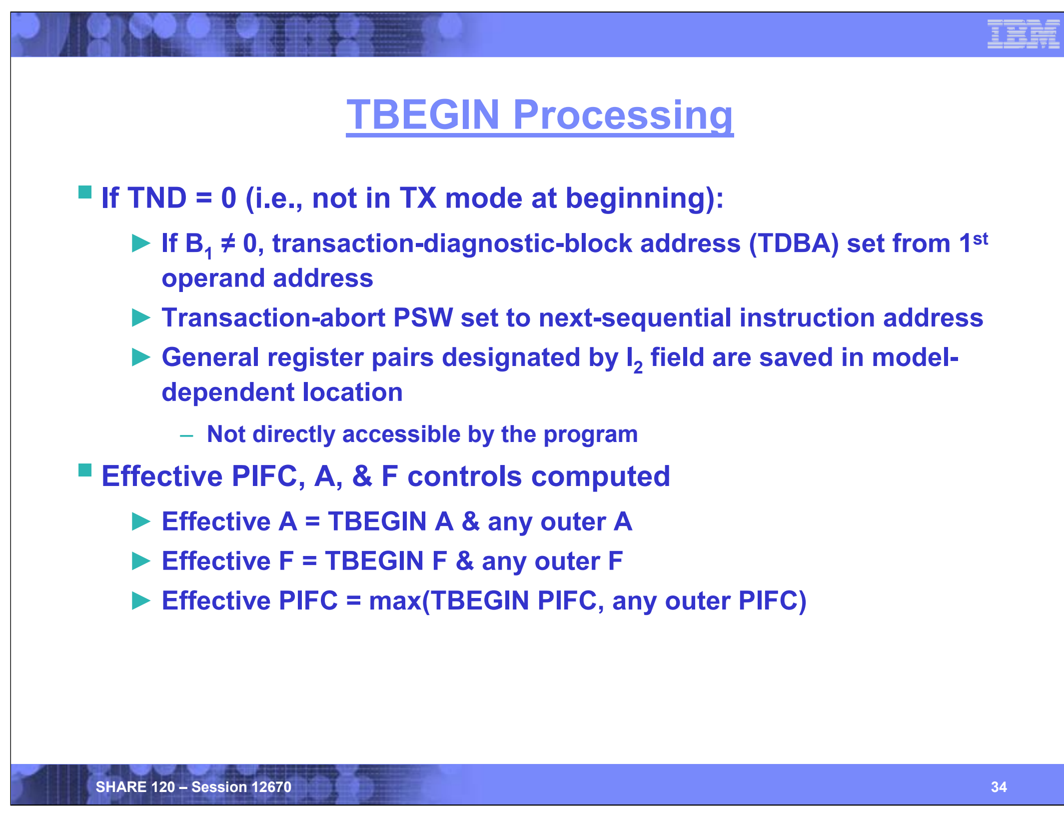
When an outermost TBEGIN is executed, and the base register of the first operand designates a register other than general register zero, the transaction-diagnostic-block address (TDBA) is set. If the transaction is aborted, diagnostic information will be (nontransactionally) saved in this block If the B_1 field of the instruction is zero, the TDBA is considered to be invalid, and no program-specified diagnostic information is saved.

The I_2 field contains controls, as follows

The first 8 bits of the I_2 field contain the general register save mask (GRSM). This field applies only to the outermost TBEGIN, and it is ignored for inner TBEGIN instructions. Each bit represents an even/odd pair of registers. If the bit is one, the contents of the register pair are preserved at the beginning of TX mode, and restored if the transaction is aborted. If the bit is zero, the register pair is neither saved nor restored.

General register are the only registers that may be saved, and if the transaction aborts, restored. Access registers and floating-point registers are not restored on an abort. The A and F controls, bits 12 and 13 of the I_2 field, respectively, provide a means by which the program can prohibit a transaction from altering ARs or using FP instructions. Unlike the GRSM (which applies only to the outermost TBEGIN), the A and F controls assume effective values for each nested level of a transaction.

Finally, bits 14-15 of the I_2 field contain a program-interruption filtering control. More on filtering in later slides.



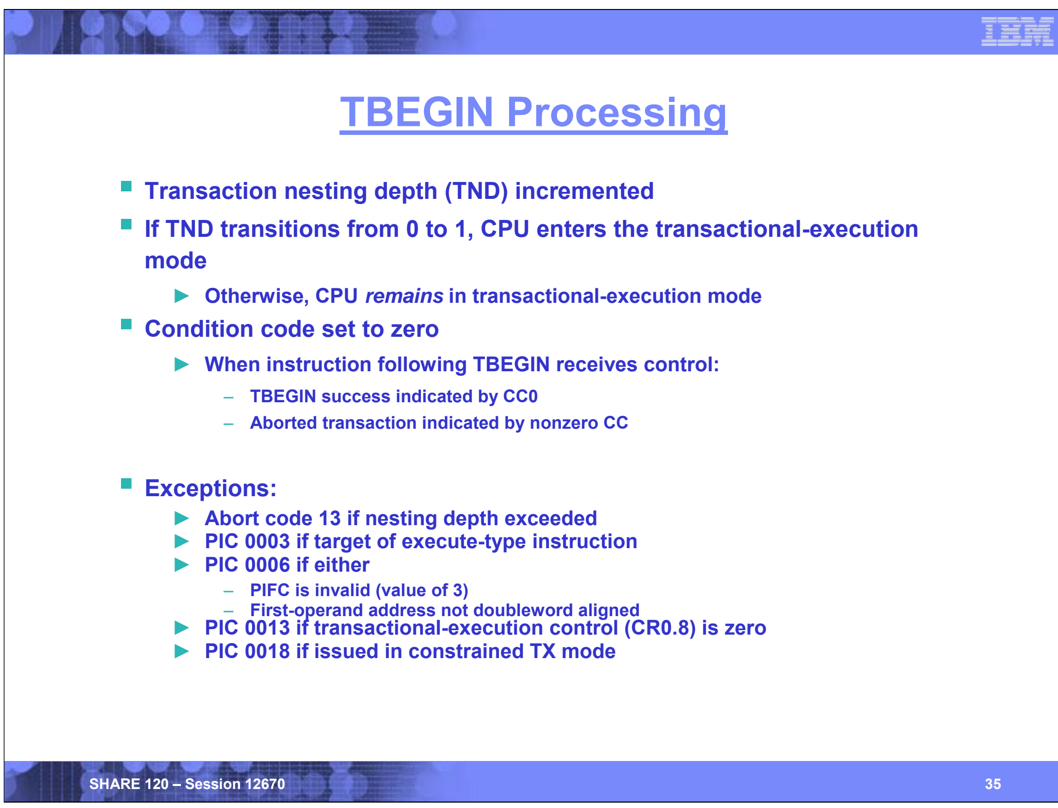
This slide described the step-by-step processing that occurs during the execution of a TBEGIN instruction.

If the transaction-nesting depth (TND) is zero (that is, the CPU is not in the TX mode), then the following occurs:

1. If the B_1 field designates a register other than zero, the transaction-diagnostic-block address (TDBA) is set. If the B_1 field is zero, the TDBA is considered to be invalid.
2. The transaction-abort PSW is set to point to the instruction following the outermost TBEGIN.
3. Any general register pairs designated by the GRSM field of the instruction are saved in a model-dependent location that is not accessible by the program.

Regardless of whether the TND is zero, effective AR-modification, FR-modification, and program-interruption-filtering controls are computed, as follows.

- The effective A and F controls are the logical AND of the controls in the TBEGIN instruction and the respective controls in any outer TBEGIN instruction.
- The effective PIFC is the maximum of the control in the TBEGIN instruction and any outer TBEGIN instruction.



Continuing with TBEGIN processing:

The transaction nesting depth (TND) is incremented. If the TND transitions from zero to one, the CPU enters the TX mode; otherwise, the CPU remains in the TX mode.

The condition code is set to zero.

Note, when a nonconstrained transaction aborts, control is passed to the transaction-abort PSW (TAPSW), the instruction address of which points past the outermost TBEGIN instruction. The condition code in the TAPSW will either be 2 or 3. Thus, the instruction following the outermost TBEGIN instruction is expected to be a conditional branch instruction. If the CC is zero, then it means that the transaction successfully initiated, and control is expected to fall through to the next instruction. If the CC is nonzero, then it means that the transaction was aborted, and control is passed to an abort handler.

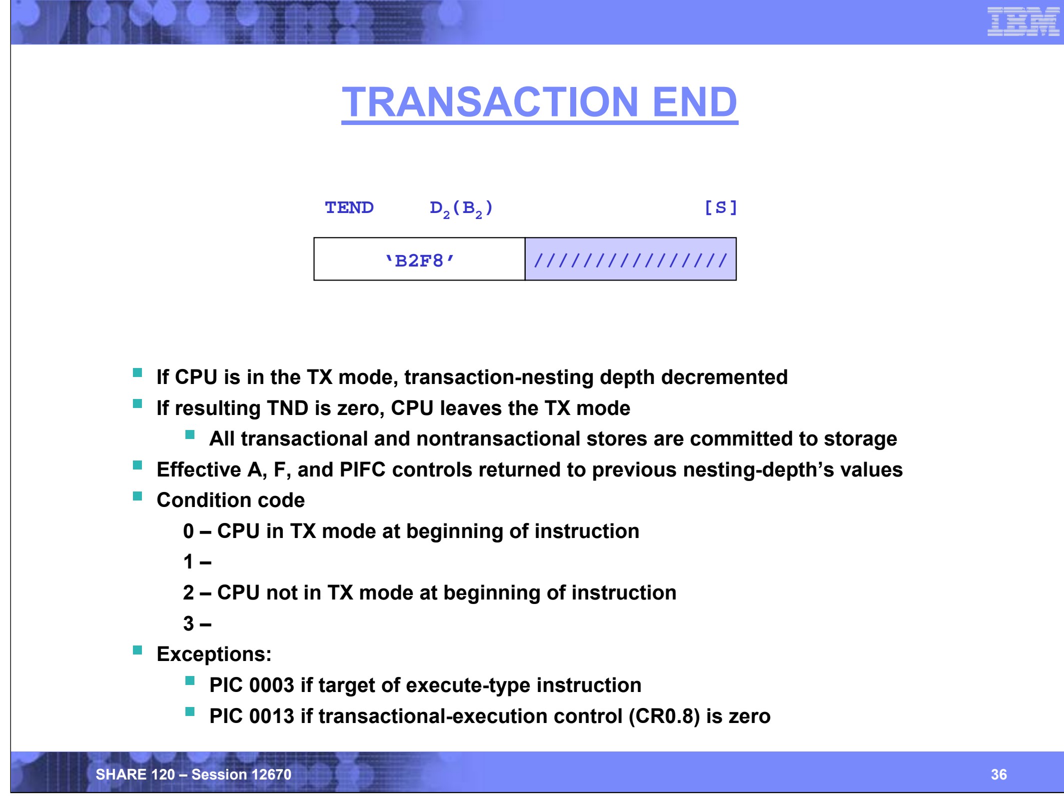
As shown on this slide, there are various exception conditions. Of note:

• There is an abort condition if the maximum nesting depth of 16 is exceeded.

• A specification exception is recognized if a reserved PIFC value is coded, or if the first-operand address is not on a doubleword boundary.

• An execute exception is recognized if the instruction is the target of an execute-type instruction.

• If attempted in the constrained TX mode, a transaction-constraint exception is recognized.



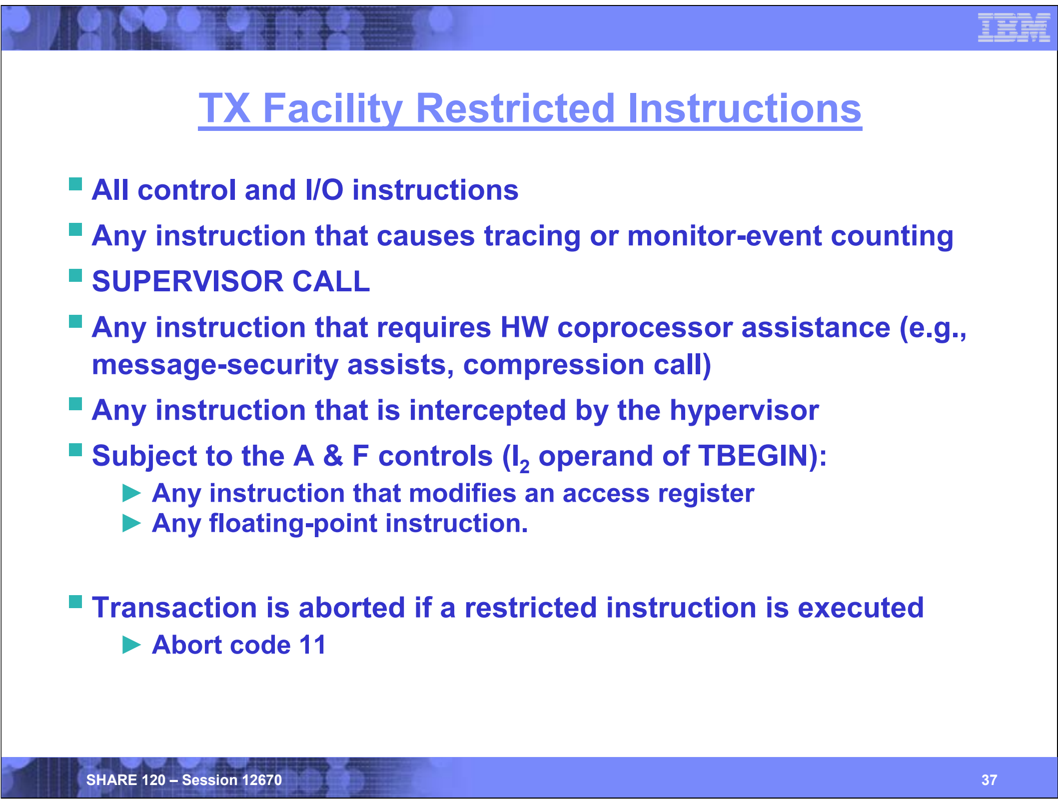
The TRANSACTION END (TEND) instruction is used to end a section of code that is executing in the TX mode. It is an S-format instruction, but has no operands.

If the CPU is in the TX mode, the transaction nesting depth (TND) is decremented.

If the resulting TND is zero, then the CPU leaves the TX mode, committing all stores to memory. Otherwise, the effective A, F, and PIFC control return to the value of the previous nesting-depth values.

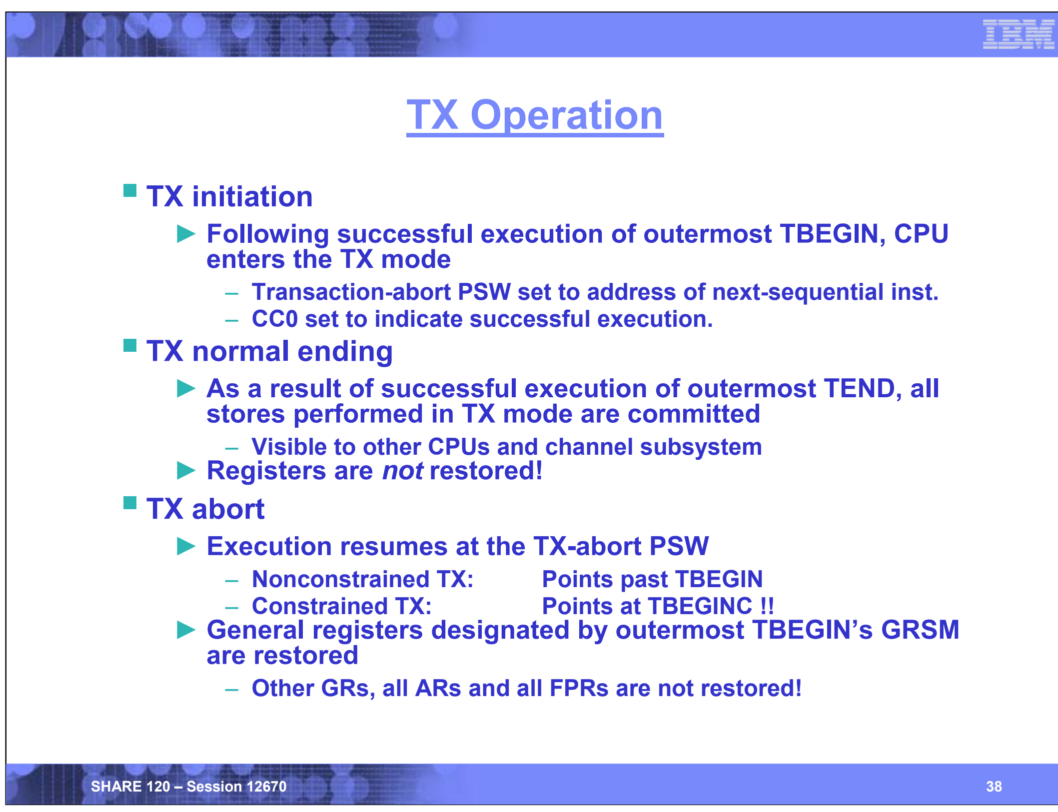
TEND may be executed even if the CPU is not in the TX mode. The condition code indicates whether the CPU was in the TX mode (0) or not (2).

As with certain other TX-facility instructions, TEND may not be the target of an execute-type instruction.



The TX facility imposes numerous restrictions on the instructions that can be executed when the CPU is in the TX mode. Each of these are enumerated on this slide.

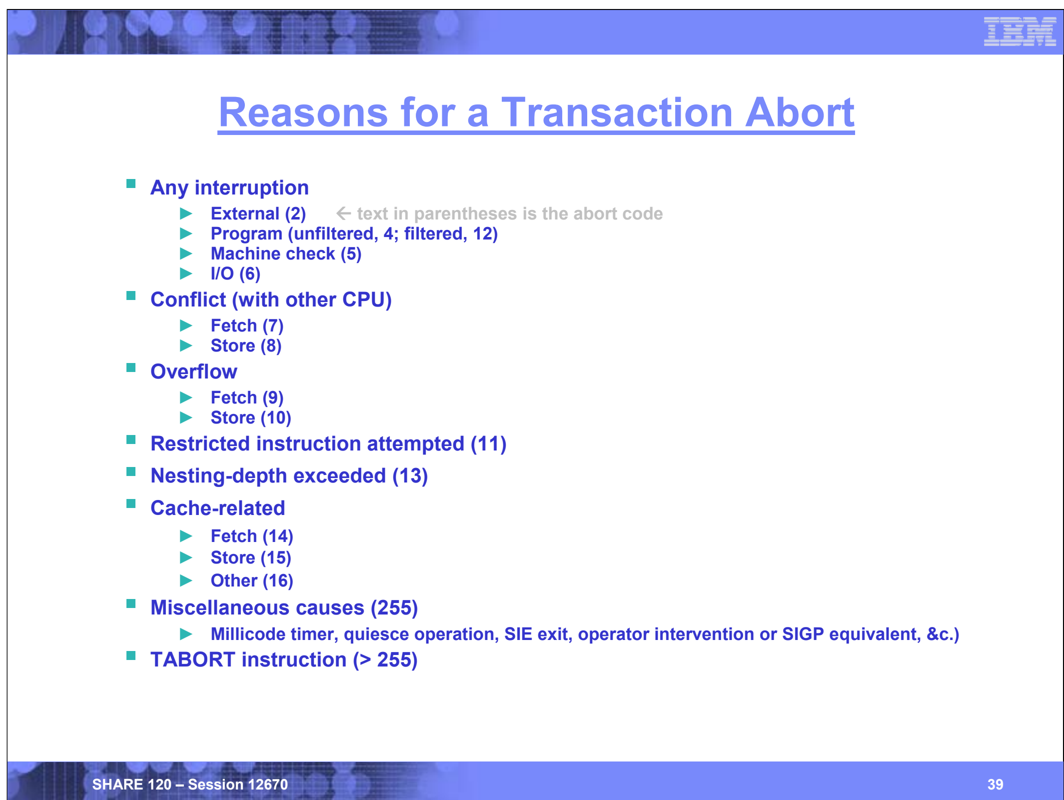
If a restricted instruction is attempted, the transaction is aborted with abort code 11 (see later slides for a complete list of abort codes).



This slide reviews the operation of the CPU while in the TX mode. Of particular note:

When a transaction completes normally, there is no restoration of general registers, access registers, or floating-point registers.

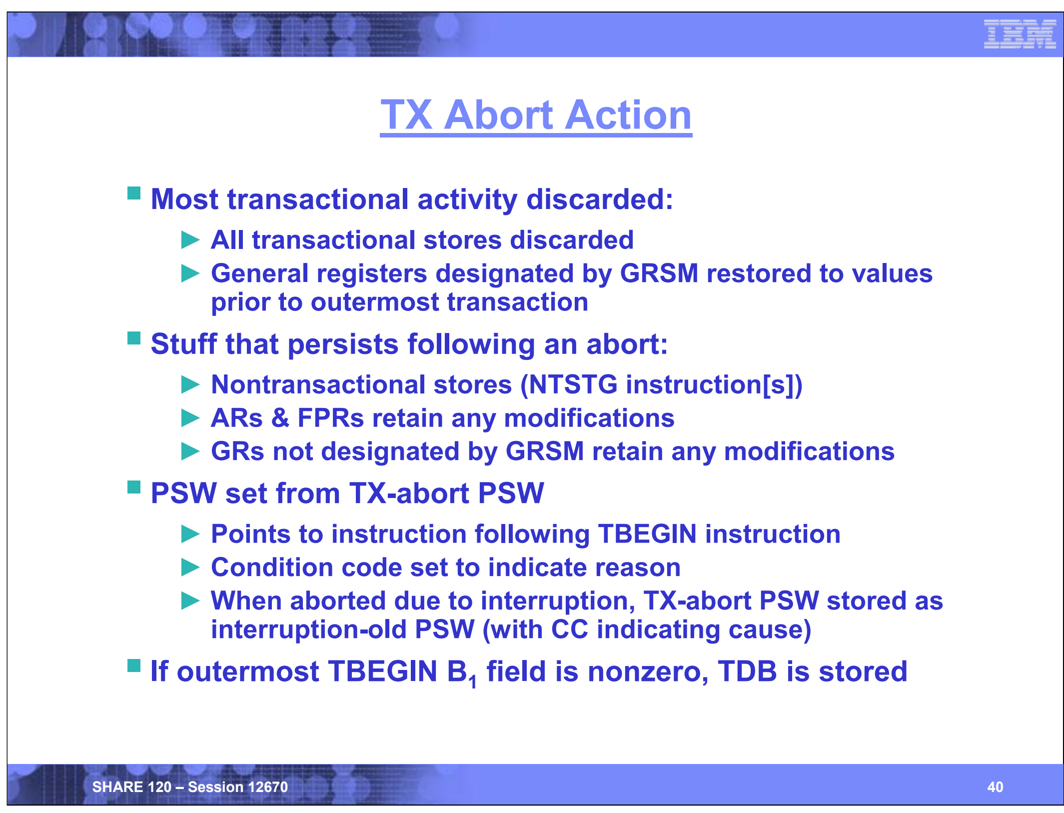
When a transaction aborts, registers designated by the GRSM field of the outermost TBEGIN instruction are restored; all other GRs, and all ARs and FRs and the floating-point control register (FPCR) are not restored.



This slide enumerates the various reasons that a transaction may be aborted. The numbers shown in the parentheses are the abort code.

Conflict conditions (codes 7, 8, and 14-16) represent cases where other CPUs have attempted to fetch from or store into locations that the transaction has stored into or fetched from, respectively.

Note that a transaction may be aborted due to any interruption, including an external interruption (such as a time-slice ending) or an I/O interruption. Thus, it is never assured that a transaction will complete on its first execution. Examples of re-driving an aborted transaction are shown below.



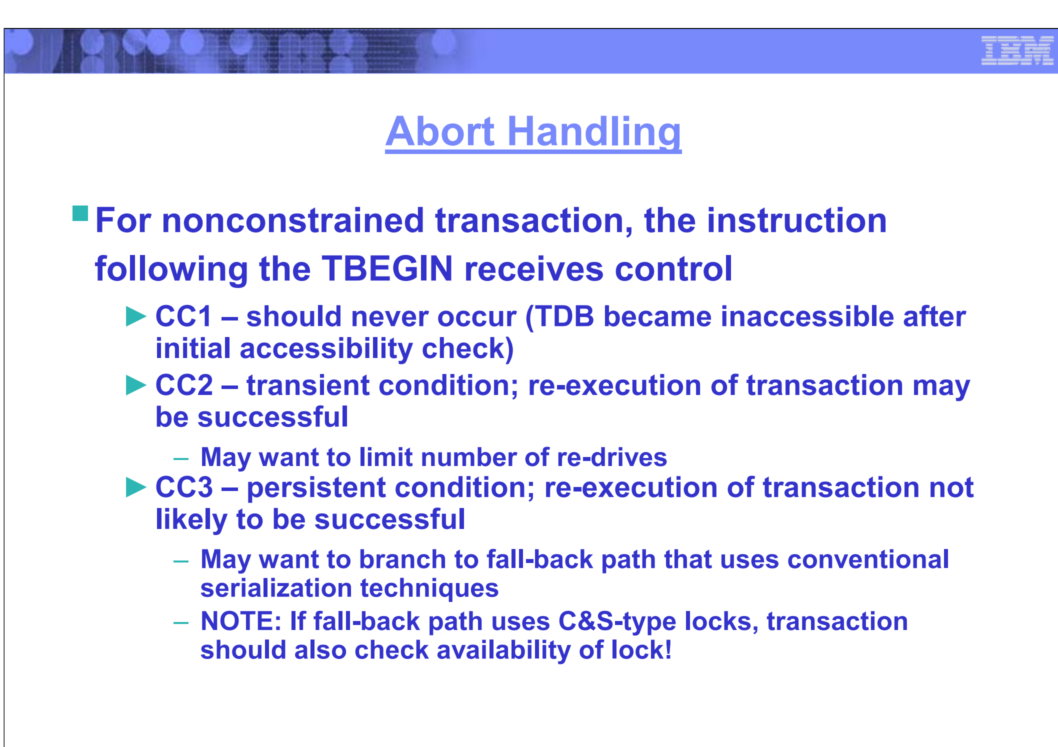
When a transaction is aborted, all transactional stores are discarded … not just as observed by other CPUs, but by the CPU executing the transaction as well. Thus, it's as if the stores never occurred. (There may be some lingering hints of stores having occurred, such as change bits remaining set in the storage keys, but as far as an application is concerned, there's no evidence.)

Additionally, any general register pairs that are specified in the general register save mask are restored to their pre-TX-mode contents. If all eight register pairs (that is, all 16 GPRs) are specified, then all contents of the registers are restored. Note, saving and restoring registers does consume CPU cycles, so specifying the minimum register set is recommended.

Things that are retained following a transaction's abort include any nontransactional stores made by the NTSTG instruction. Also, any ARs or FRs that were modified by the transaction retain their changes.

The transaction-abort PSW that was set by the outermost TBEGIN instruction, with a condition code set to indicate the severity of the abort, becomes effective. Thus, the instruction following the TBEGIN receives control following the abort of a nonconstrained transaction (or the TBEGINC instruction receives control following the abort of a constrained transaction … more on this later).

Finally, if the B_1 field of the outermost TBEGIN instruction was nonzero, then a *program-specified* TDB is stored. Note, if the transaction is aborted due to program interruption conditions, then a program-interruption TDB is also saved in the prefix area of low storage.



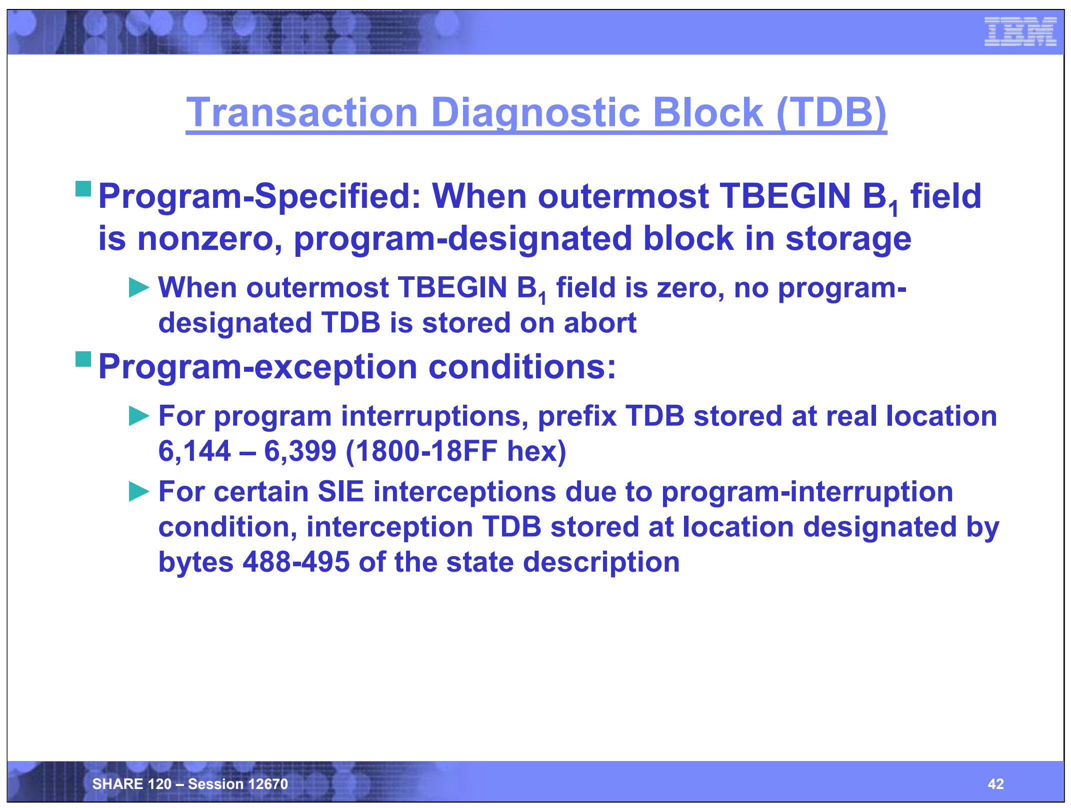
The condition code that is set in the transaction-abort PSW is always a nonzero value. Thus, the instruction following the outermost TBEGIN can determine whether it is being executed due to the successful execution of TBEGIN, or if it is the result of a transaction being aborted.

CC1 represents an extremely rare condition that should never occur. This indicates that the program-specified TDB (the accessibility of which is checked during the execution of the outermost TBEGIN) has become inaccessible during the execution of the transaction. This can only happen due to unexpected key changes made by the operating system – and should never occur.

CC2 represents a transient condition such as a conflict with another CPU. This condition is likely to be temporary, thus a repeated attempt at executing the transaction is likely to produce a successful completion.

CC3 represents a persistent condition, such as having exceeded the nesting depth or encountering an operation exception. Without program intervention, these conditions are not likely to go away on their own. Thus repeated attempts at executing the transaction are likely to continue to abort.

One special condition is worth mentioning. If TX is being used for lock elision (that is, to avoid using a lock), but the fall-back code path uses a lock word, then the transactional execution code path should test the accessibility of the lock to ensure that it’s free. That way, the TX code path can co-exist with any non-TX code paths that attempt to serialize on the same resources.



The transaction diagnostic block is a 256-byte area in memory. There are three types of TDBs, of which zero, one, or two may be stored when a transaction aborts.

- When the transaction diagnostic-block address (TDBA) is valid – that is, when the B_1 field of the outermost TBEGIN instruction is nonzero – then a program-designated TDB is stored if a transaction is aborted.
- For program interruptions, a TDB exists in the prefix area at locations 1800-18FF hex. A subset of the TDB information is saved in the prefix-area TDB (only a subset is saved, because some information in the TDB such as the program-interruption ID is already saved in other prefix-area locations).
- For conditions that cause the CPU to leave the interpretive-execution state (called SIE interceptions), an interception TDB is stored at a location designated by the hypervisor (that is, LPAR or z/VM).

TDB Contents:

Offset				
0	Format	Flags	Reserved	Trans Nest. Depth
8	Transaction Abort Code			
16	Conflict Token			
24	Aborted Transaction Instruction Address			
32	EAID	DXC	Reserved	Program Interruption ID
40	Translation Exception ID			
48	Breaking-Event Address			
56	Reserved			
112	Model-Dependent Diagnostic Information			
128 – 248	General Registers			

SHARE 120 – Session 12670 43

The contents of the TDB are shown here.

• Format: When zero, the remaining fields of the TDB are unpredictable. When one, the fields are as shown in this slide.

• Flags: Bit 0 indicates that the conflict token (bytes 16-23) is valid. Bit 1 indicates that the CPU was in the constrained TX mode when the abort occurred.

• TND: The transaction-nesting depth at the time of the abort.

• Transaction Abort Code: The CPU-generated code or second operand of the TABORT instruction.

• Conflict Token: The logical address at which a conflict was detected. This field is valid only if flag bit 1 is one.

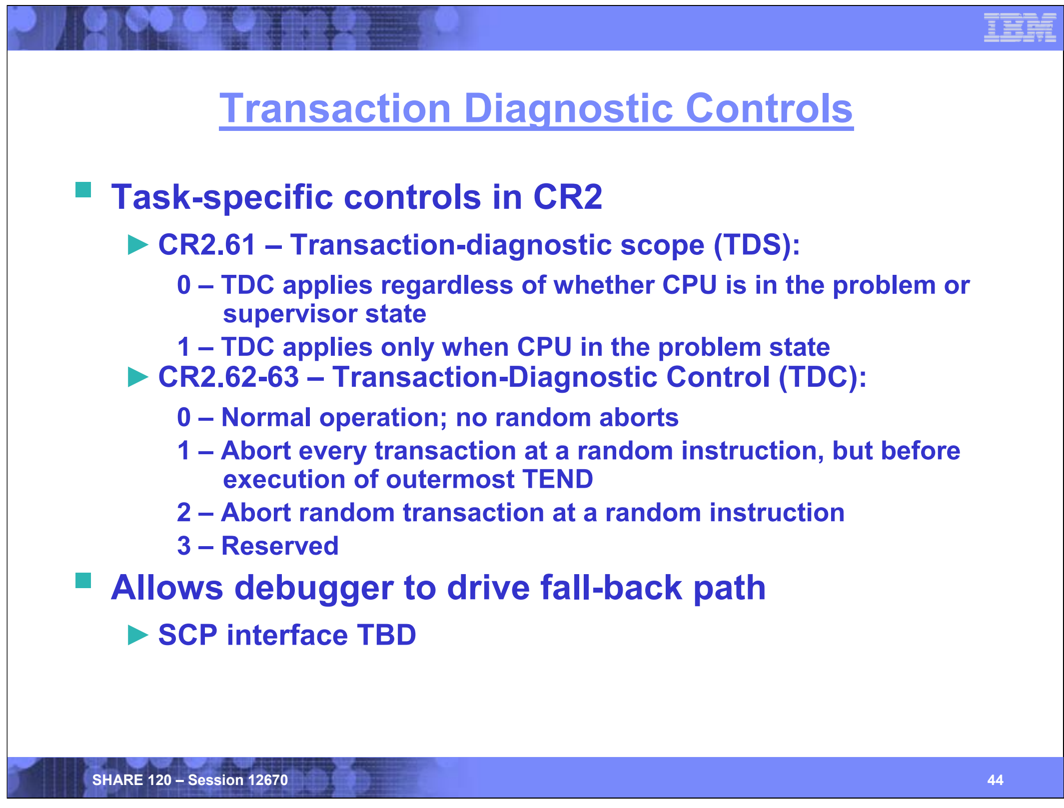
• Aborted-Transaction Instruction Address: The instruction address at which the abort was detected.

• EAID, DXC, program-interruption ID, translation-exception ID, and breaking-event address are only stored for transactions that are aborted due to program interruptions, and some of these fields are only stored for certain program interruptions. See the *z/Architecture Principles of Operation* (SA22-7832-09) for details on these fields.

• Model-Dependent Diagnostic Information: IBM internal-use diagnostic information.

• General Registers: The contents of all 16 general registers at the time of the abort.

.



Control register 2 contains information that is unique to a task (sometimes called process or dispatchable unit). z/OS alters CRs for each task that is dispatched.

Two new fields are added to CR2 in support of debugging a transaction:

Bits 62-63 contain a transaction diagnostic control (TDC).

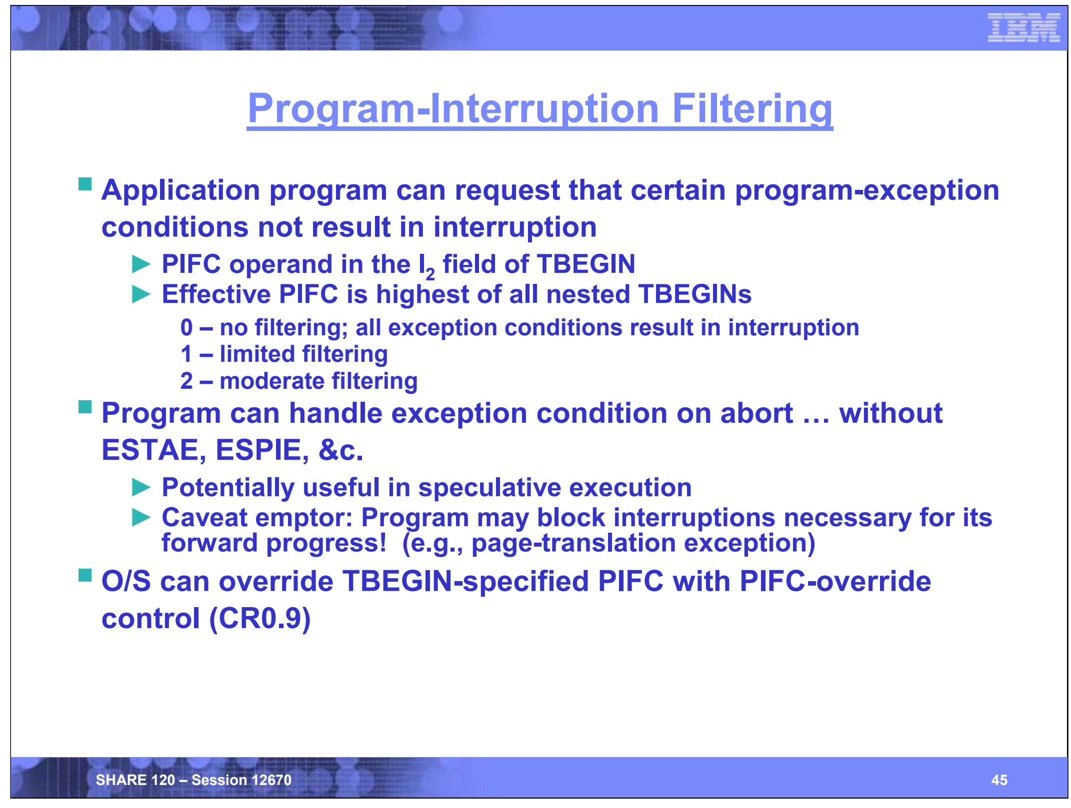
0 - means that transactional execution will not be randomly aborted … at least, not due to the TDC.

1 - means that every transaction will be aborted at a random instruction, but before the TEND instruction.

2 – means that random transactions are aborted at random instructions.

The TDC allows a debugger to deliberately cause a transaction to be aborted, thus allowing the testing of the abort-handler fall-back code path (that is, the path branched to by a nonzero condition code in the instruction following the TBEGIN).

Bit 61 contains the transaction diagnostic scope (TDS). This bit controls the effectiveness of the TDC (in bits 62-63). When the TDS is zero, the TDC applies to both the supervisor and problem states; when the TDS is one, the TDC applies only to the problem state.

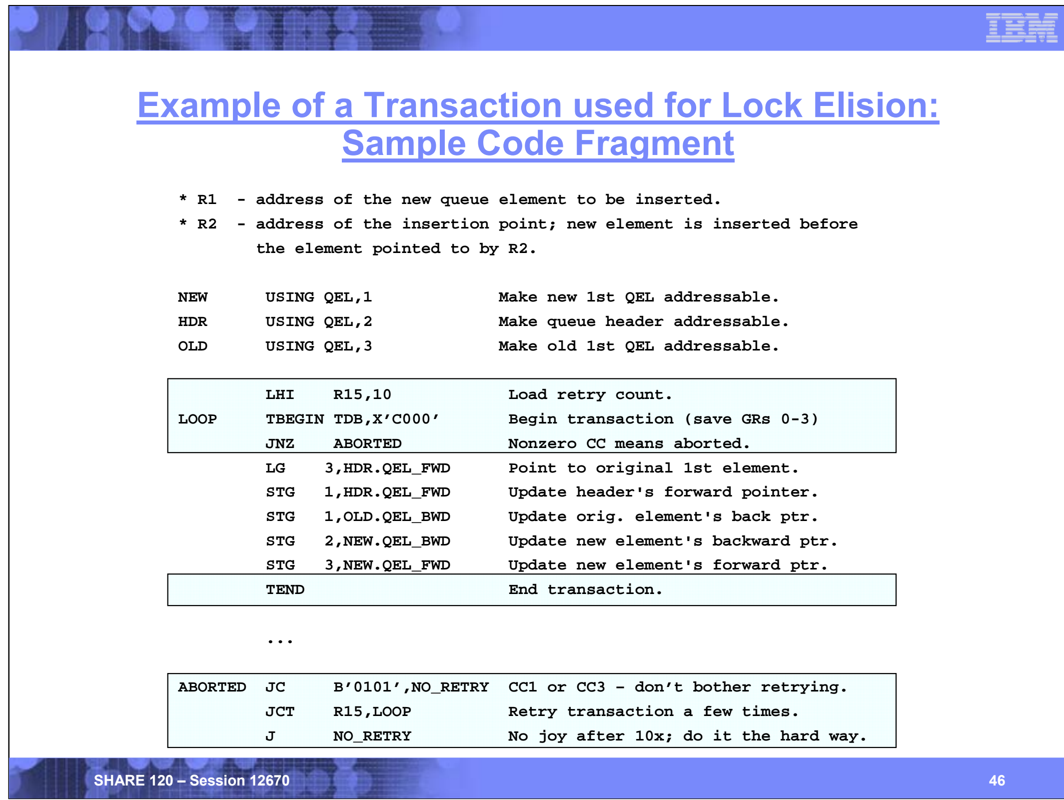


One of the very powerful features of transactional execution is program-interruption filtering. By means of the program-interruption-filtering control (PIFC, bits 14-15 of the I_2 field of the TBEGIN instruction), the program can request that certain classes of program-exception conditions that occur during TX not result in a program interruption. Rather, the transaction is simply aborted, and control is passed to the abort-handler transaction-abort PSW.

Thus, without establishing any elaborate recovery environment (such as using the z/OS ESPIE or ESTAE macros), the program can efficiently receive control if it causes an exception. This may be particularly useful in a speculative-execution environment, for example, the Java null-checking scenario. If an access exception is recognized dereferencing a pointer, then the transaction aborts, and the program can try a more conservative code path instead.

However, the program can cause program exceptions to be filtered that are otherwise necessary for it to make progress. For example, if page-translation exceptions are filtered, then the OS may not see the exception and cause the page frame to be migrated in from auxiliary storage. Therefore, a transaction's fall-back path may need to reference storage locations that cause protection or translation exceptions in order to allow the OS to resolve the exceptions.

The OS has the ability to completely override program-interruption filtering by means of the program-interruption filtering override – bit 9 of control register 0.



This assembler program fragment illustrates the same code sequence as originally shown on slide 17.

In this example, the SETLOCK macro instructions are replaced with TBEGIN and TEND instructions, as shown in the first two highlighted areas. The code fragment uses general register 15 as an abort counter.

If the transaction is aborted, the code branches to the label ABORTED, where a determination is made as to whether the transaction should be re-driven. For condition codes 1 or 3, there is little chance of recovery, so the code is not reattempted (it branches to NO_RETRY). However, if the condition code is 2, the count in general register 15 is decremented, and, if nonzero, the transaction is attempted again.

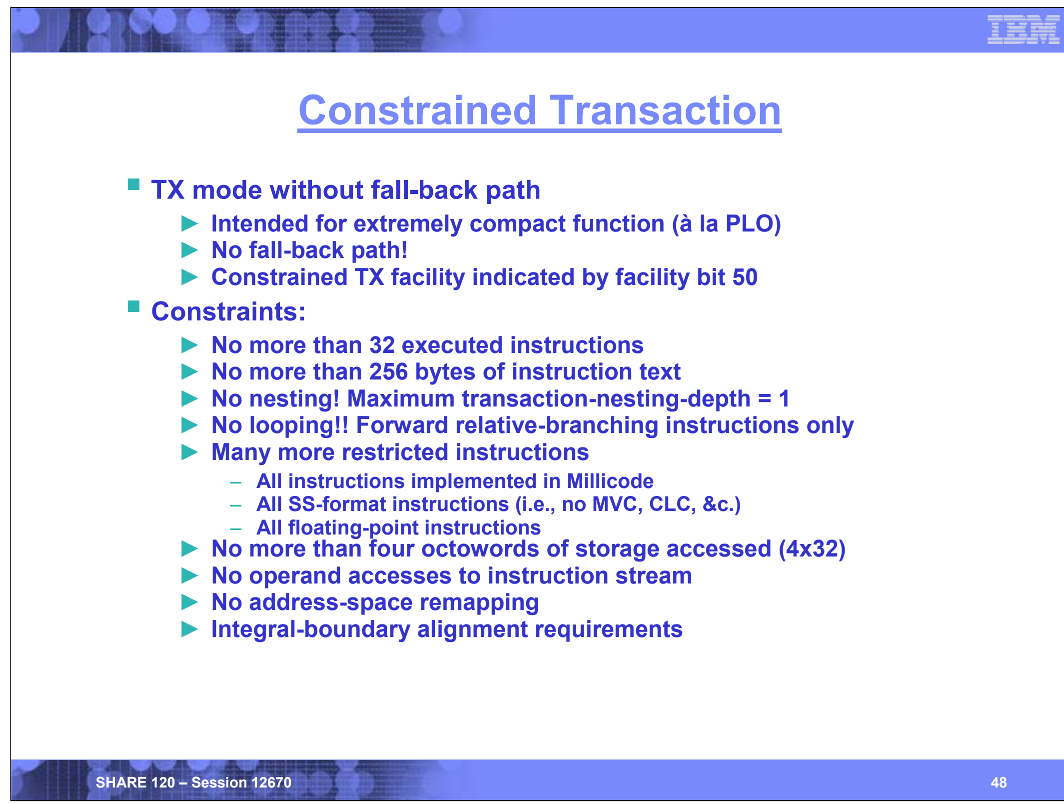
General Comments

- If the transaction is used for lock elision, and the fall-back path uses a lock, the transaction must (at least) fetch the lock word to see that it's available.
  - Ensures that the transaction aborts if another CPU accesses the lock non-transactionally.
- Coding of both transactional and fall-back path adds to the complexity of the code
  - Hence, constrained transactions for small updates (see next slides)
- If transactions are nested, outermost transaction must account for unanticipated abort conditions that occur in the inner transaction
  - E.g., modified, but unrestored GRs, ARs, FPRs

As noted earlier, a transaction that is used for lock elision – but has a fall-back path or other code paths that use a conventional lock – should check to see if the lock is available, and if not, end and branch to an abort handler. This ensures that the transactional and conventional code can successfully co-exist.

Also, coding of a transaction may be quite simple, but having both transactional and fall-back code paths may be more complicated and require additional testing. Constrained transactions, to be discussed shortly, address this limitation by eliminating the fall-back path completely.

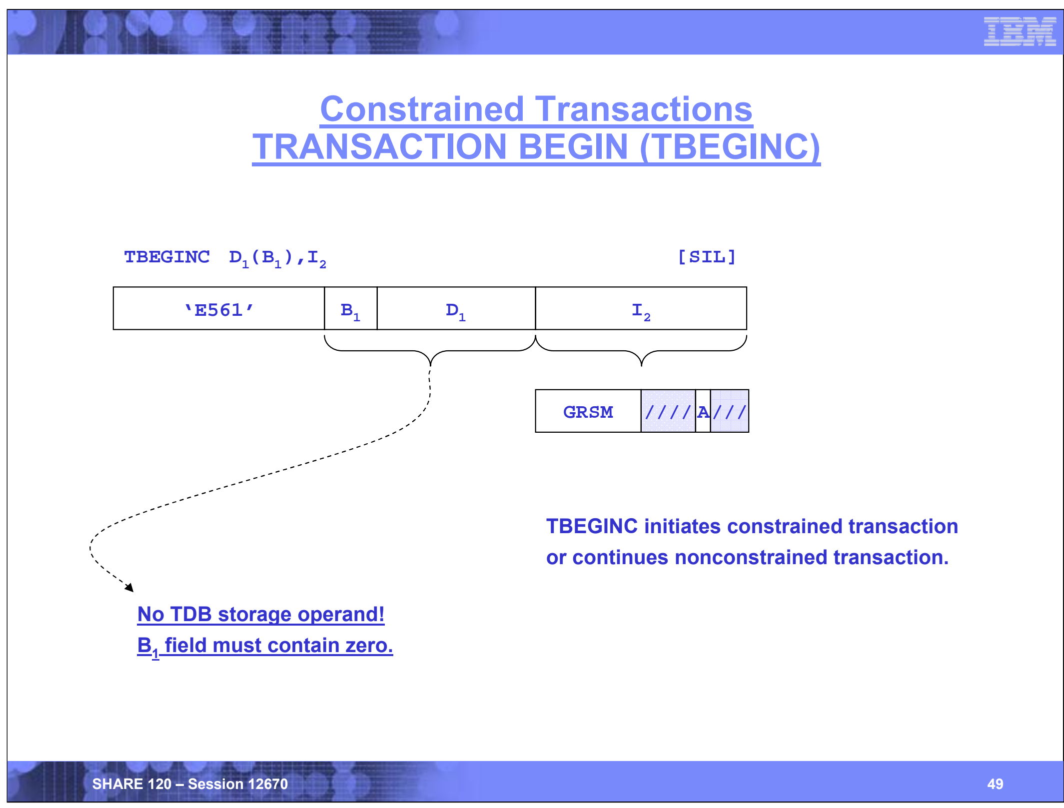
When transactions are nested, an outermost transaction might encounter aborts due to unanticipated conditions discovered in an inner transaction. For example, an outermost transaction might set the GRSM to save registers 0-3, saving only those registers that it modified. However, if this code called some other function – perhaps a library routine – that modified other registers, and then the transaction aborted, the program may be unprepared to deal with other changed registers that were altered by the called program.



Coding a fall-back code path introduces a fair amount of complexity into transactional execution. The constrained transaction minimizes that complexity by eliminating the need for a fall-back path. However, the constrained transaction has significant additional restrictions (constraints), as enumerated on this slide.

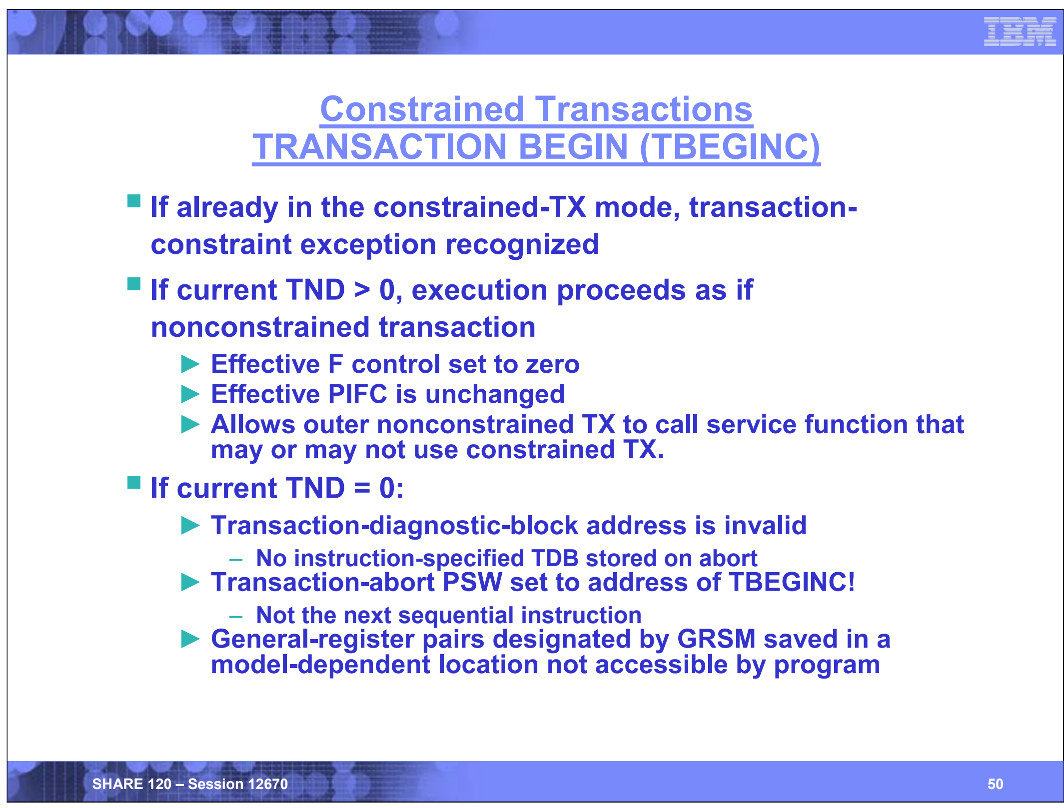
Even though a constrained transaction may initially abort, it is assured of eventual completion.

Thus far, when we have discussed transactional execution, we have been discussing the *nonconstrained* TX mode (as initiated by a TBEGIN instruction). The following discussion describes an additional *constrained* TX mode (as initiated by the TBEGINC instruction).



This slide illustrates the TBEGINC instruction, a variant of TRANSACTION BEGIN that is used to initiate the constrained transaction. TBEGINC is similar to TBEGIN, except as follows:

1. There is no abort handler!
2. Since there is no abort handler, there is no need for a program-specified transaction-diagnostic block (TDB). The B_1 field of the TBEGINC instruction must contain zero!
3. The controls in the I_2 field are limited: there are no floating-point control (F) or program-interruption-filtering control (PIFC) bits.



Execution of TBEGINC is as follows.

1. Nesting within a constrained transaction is not permitted. If the CPU is already in the constrained TX mode, then a transaction-constraint program interruption is recognized (program interruption code 0018 hex).
2. If the transaction nesting depth is already greater than zero (meaning the CPU is in the nonconstrained TX mode), then execution simply proceeds as if this was a nonconstrained transaction. In this case, the effective F control is zeroed, and the effective PIFC remains unchanged. This allows an outer, nonconstrained transaction to call a service function that may or may not use constrained TX mode.
3. If the current TND is zero, then:
    1. The TDBA is marked as invalid (since there is no abort handler, there is no need for a transaction diagnostic block)
    2. The transaction-abort PSW is set to point directly at the TBEGINC instruction! This means that if the transaction is aborted, it will be re-driven – without attempting to branch to an abort handler.
    3. Any general registers specified by the GRSM are saved.

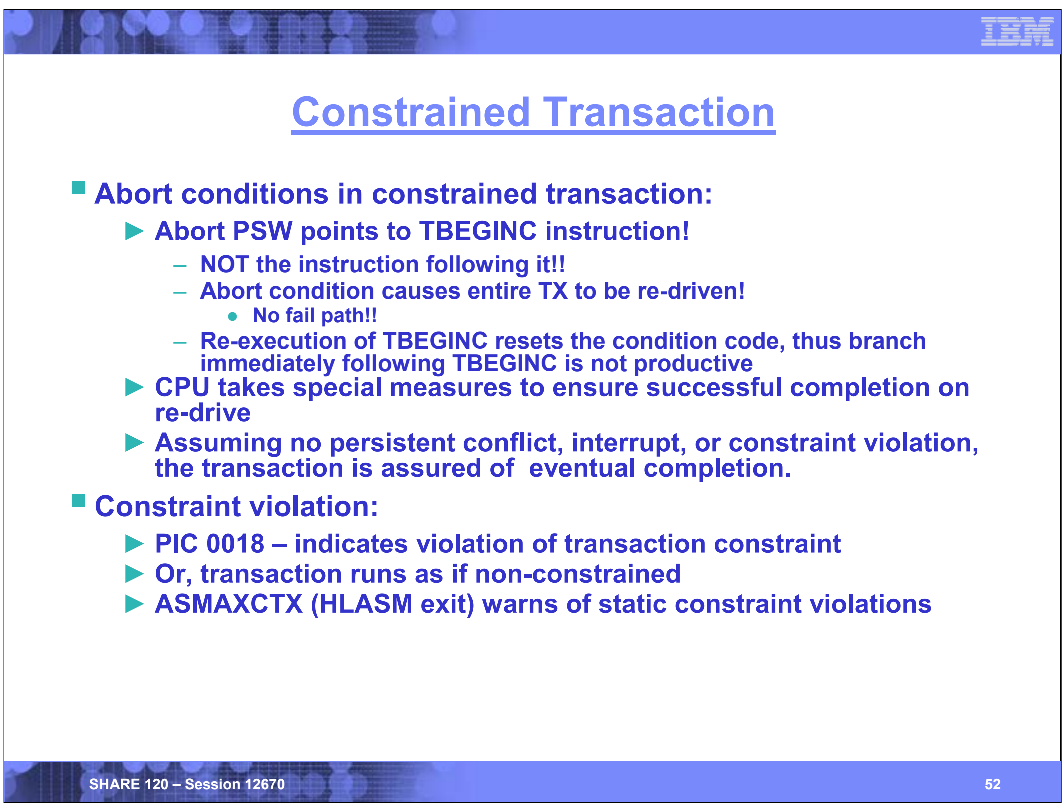
Constrained Transactions
TRANSACTION BEGIN (TBEGINC)

- Effective A = TBEGINC A & any outer A
- TND incremented
  - If TND transitions from 0 to 1, CPU enters the constrained TX mode
  - Otherwise, CPU remains in the nonconstrained TX mode
- Instruction completes with CC0
- Exceptions:
  - Abort code 13 if nesting depth exceeded
  - PIC 0006 is B_1 field is nonzero
  - PIC 0013 if transactional-execution control (CR0.8) is zero
  - PIC 0018 if issued in constrained TX mode

Continuing with TBEGINC execution:

1. The effective A control (allowing AR modification) is set to the control on the TBEGINC instruction logically ANDed with any outer value of A.
2. The transaction nesting depth is incremented. If the result is 1, then the CPU enters the constrained TX mode; otherwise, the CPU remains in the nonconstrained TX mode.
3. The instruction completes with CC0.

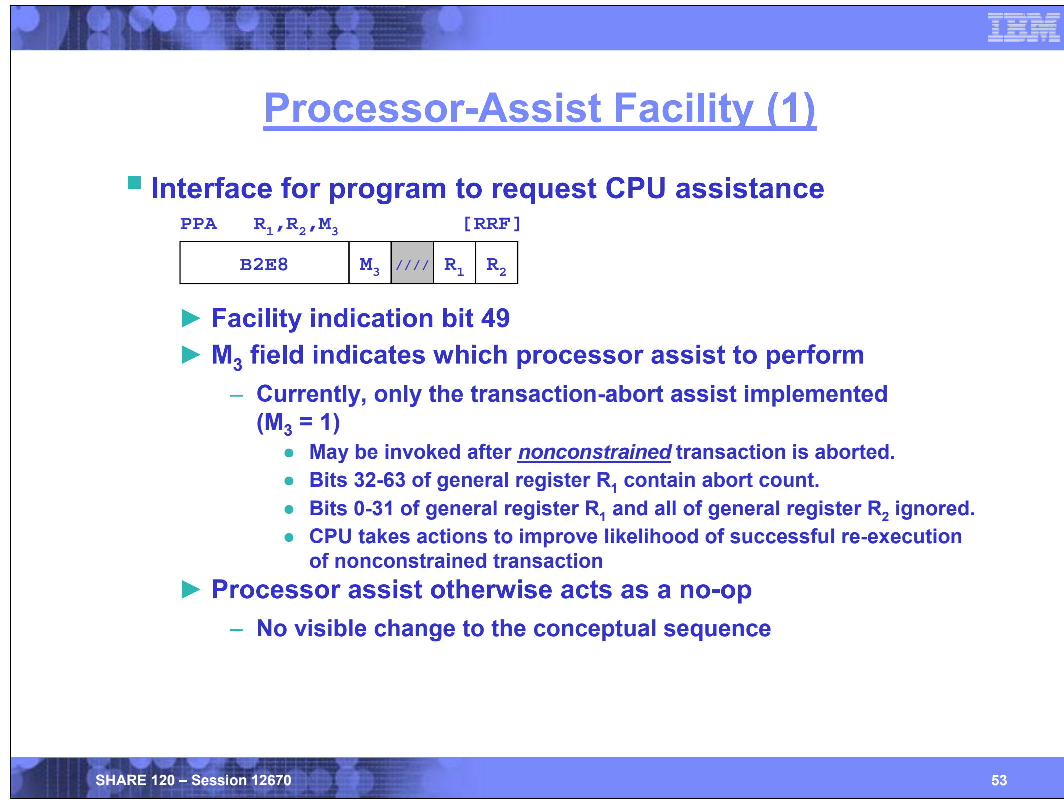
Various exception conditions may be recognized, as listed on this slide. Note, although a constrained transaction may only have a nesting depth of 1, the instruction recognizes an abort if the maximum TND is exceeded. This can occur if the CPU is already in the nonconstrained mode when TBEGINC is executed.



The astute reader may ponder, what prevents my CPU from looping forever in a constrained TX … aborting and redriving forever?

If any of the constraints are violated, this is possible. So, the constrained transaction must be carefully coded. HLASM provides an exit, ASMAXCTX, that helps in identifying constraint violations.

However, in the absence of constraint violations, there may still be causes for an abort, for example, if multiple CPUs repeatedly access the same storage locations. The CPU takes special measures to ensure that the redrive of an aborted constrained transaction will eventually complete.



What about nonconstrained transactions? Is there any CPU tricks that can help them to complete successfully?

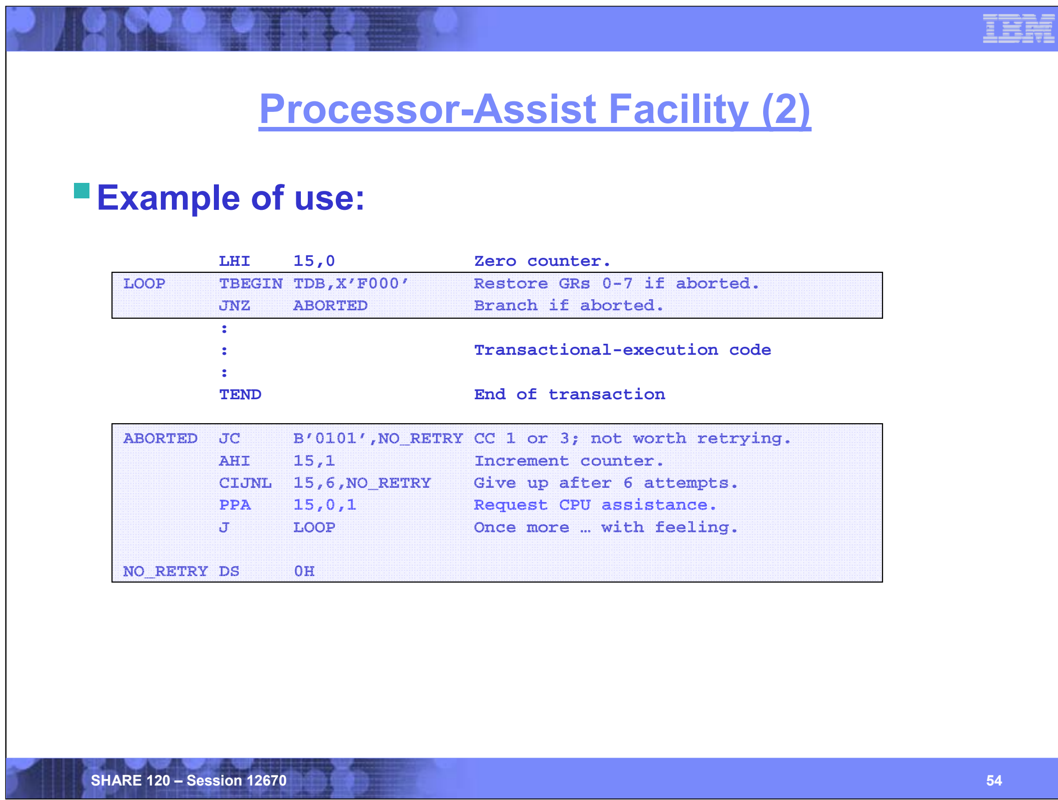
The processor-assist facility provides an instruction that can be used following the abort of a nonconstrained transaction. It provides a hint to the CPU as to how many times the transaction has aborted, and the CPU may take certain measures to help ensure that a subsequent execution of the transaction will succeed.

PPA is an RRF-format instruction designed to accommodate future assist functions; initially, only the transaction-abort assist is implemented, as indicated by an M_3 value of 1.

Bits 32-63 of the general register designated by the R_1 field contain a program-specified count of how many times a nonconstrained transaction has been aborted; initially, bits 32-63 of this register should contain zero. Bits 0-31 of general register R_1 and all of general register R_2 are ignored.

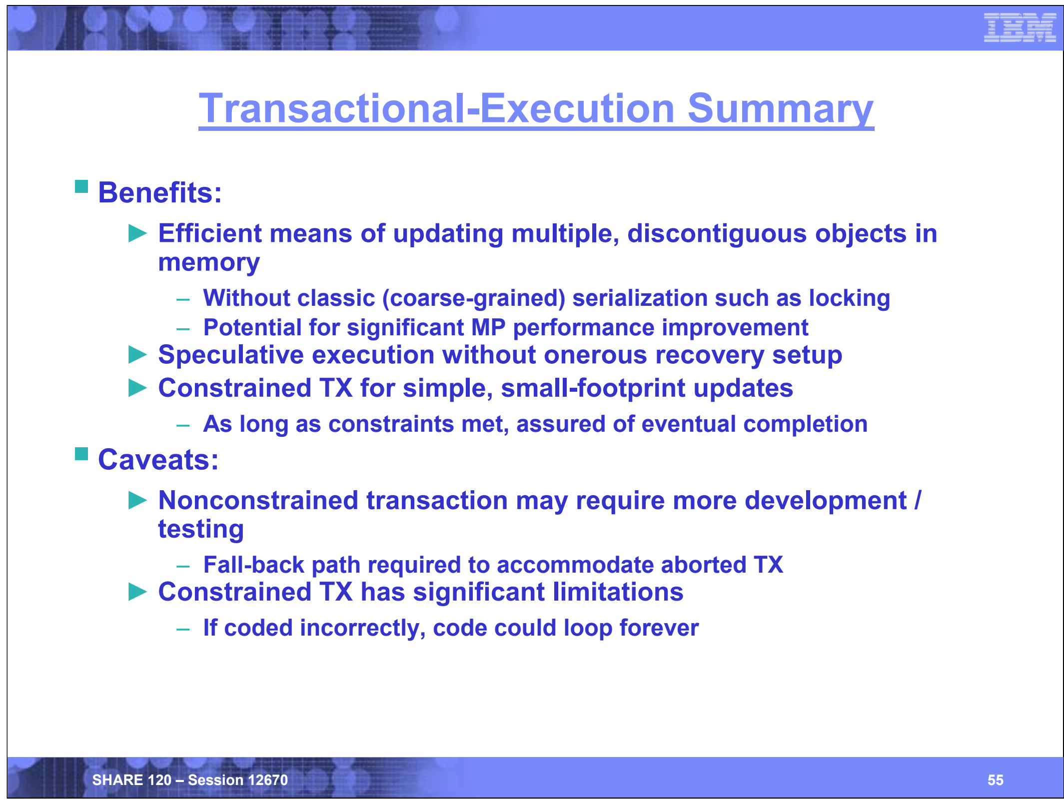
Other than the actions described above, PPA acts as a no-operation.

Note, the program should not attempt to “cheat” the recovery action by specifying a higher abort count than has actually occurred. A higher count does not necessarily provide a higher chance of successful redrive, and it may hurt program performance.



This slide shows an assembler program's use of the PPA instruction. The first block of highlighted code shows the transactional-execution code; prior to this block, general register 15 is zeroed.

In the ABORTED section of code, we again check the condition code to see if there is a chance of recovery; if not, the code branches to NO_RETRY. Next, the count in general register 15 is incremented, and if it has exceeded a threshold (six, in this case), we also branch to NO_RETRY. Otherwise, the PPA instruction is executed, as shown in the second section of highlighted code, and the program branches back to attempt to re-execute the transaction.

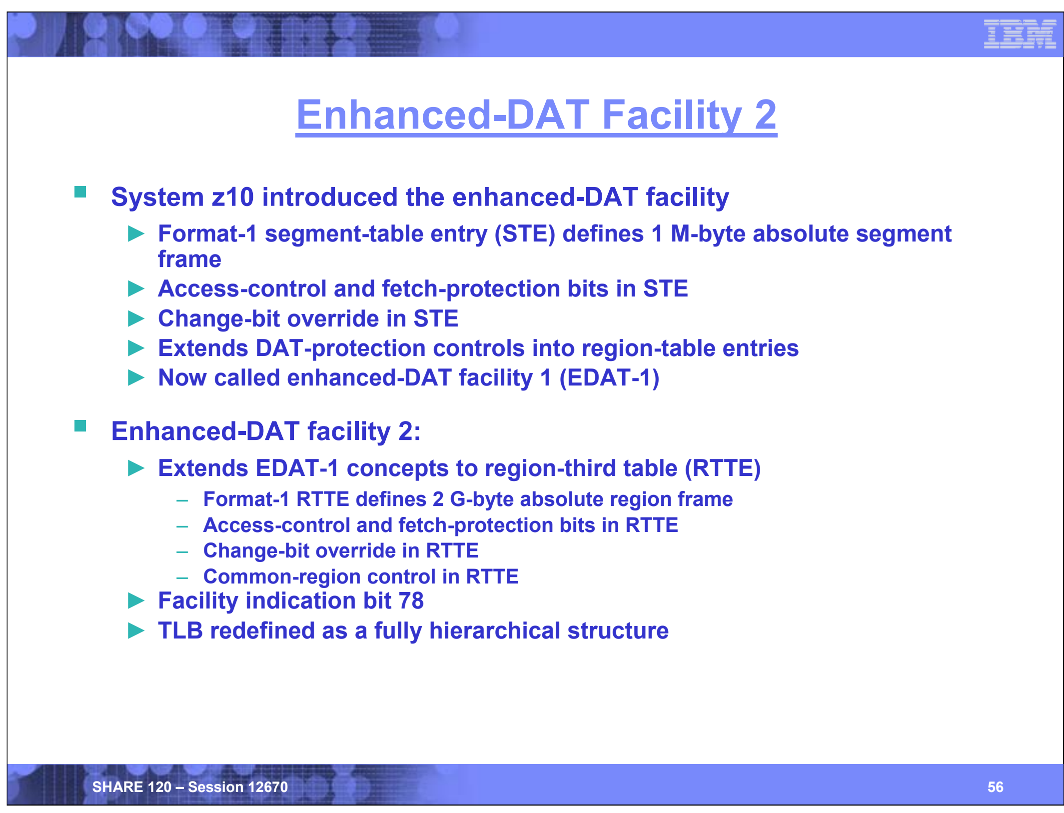


Transactional execution has enormous potential in reducing overhead associated with classic coarse-grained serialization such as locking. Already IBM lab tests have shown extremely promising improvement in certain workloads that would otherwise suffer MP effects with more processors.

Transactional execution also holds promise in implementing speculative forms of execution, and may simplify many coding paths that would otherwise need cumbersome recovery environments (like ESTAE and ESPIE).

For quick multiple-location updates with a small memory footprint, the constrained transaction provides an effective means of high-performance processing.

However, there may be additional development required to develop and test nonconstrained transactions, and constrained transactions have very stringent coding requirements.

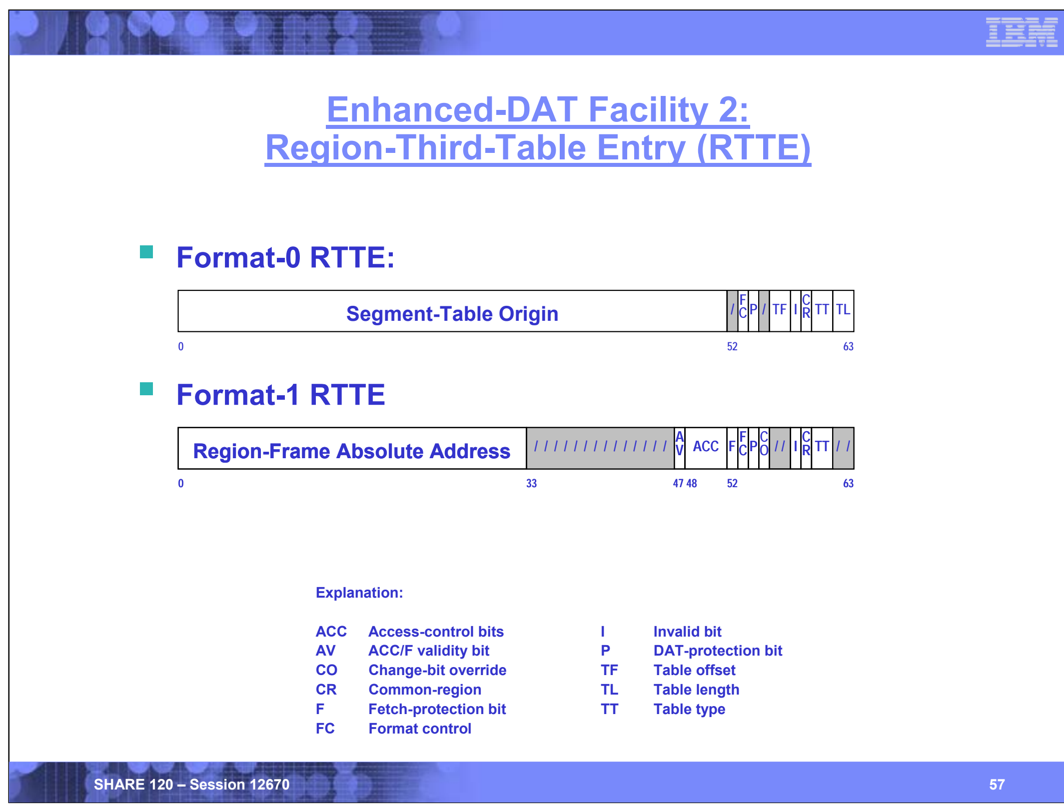


The System z10 introduced the enhanced-DAT (EDAT) facility, now called the enhanced-DAT facility 1 (or just EDAT-1, for short). The features introduced by this facility are enumerated on this slide, however the one of most attention is the 1 megabyte segment-frames (often – but inaccurately – called large pages).

Enhanced-DAT facility 2 builds upon EDAT-1, providing a super-large 2 G-byte region frame. Similar to the changes made by EDAT-1 to the segment-table entry, EDAT-2 adds new controls to the region-third table entry, providing the format control, access- and fetch-protection controls and corresponding validity indication, change-bit override, and common-region controls.

Facility indication bit 78 designates the presence of the EDAT facility 2.

Additionally, with the introduction of this facility, the of the translation lookaside buffer (TLB) is redefined to be a completely hierarchical structure. While retaining compatibility with the former TLB structure consisting of page-table entries (PTEs) and common-region-and-segment-table entries (CRSTEs), the new structure allows for more flexible future design.

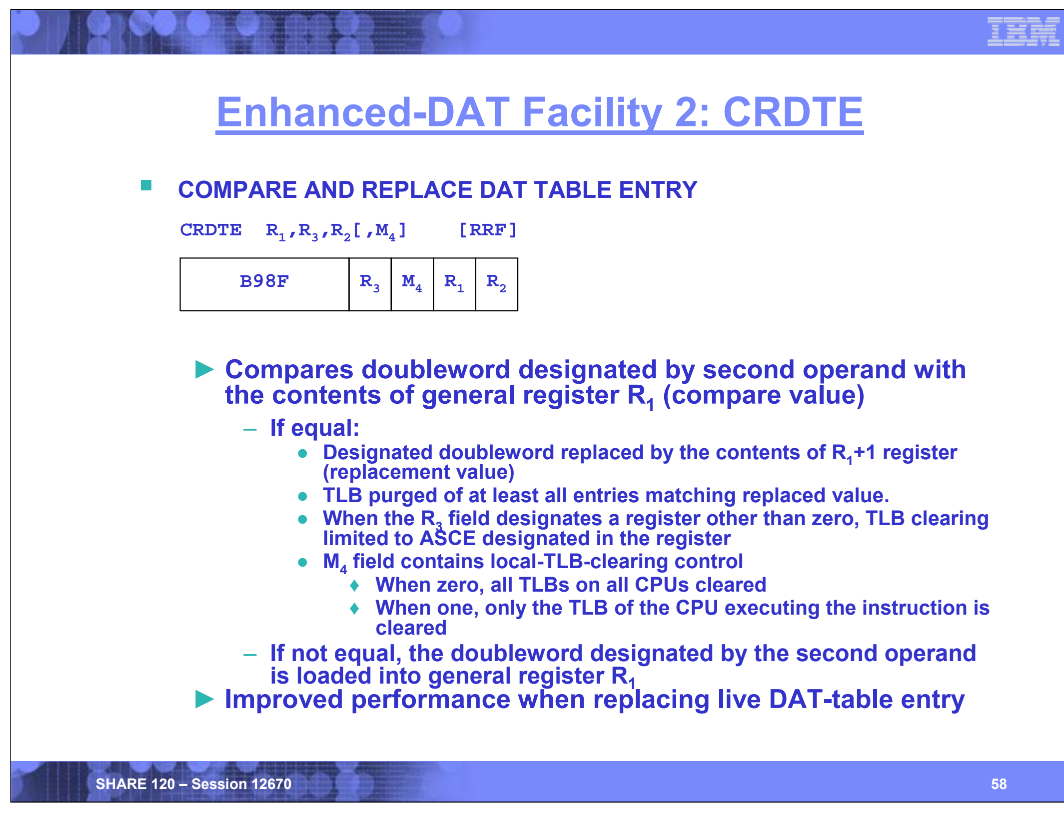


This slide illustrates the changes to the region-third-table entry (RTTE) with EDAT-2.

The RTTE is extended to include a format control in bit 53. When bit 53 is zero, the definition of the RTTE is as originally defined in z/Architecture, shown in the upper illustration (format-0 RTTE).

When the FC is one, the definition of the RTTE is as shown in the lower illustration which includes the following:

Bits 0-32	- Region-frame absolute address
Bit 47	- ACC and F bit validity indication
Bits 48-51	- Access-control bits for the region
Bit 52	- Fetch-protection bit for the region
Bit 53	- Format control
Bit 54	- Region-protected bit
Bit 55	- Change-bit override
Bit 58	- Invalid bit
Bit 59	- Common-region bit
Bits 60-61	- Table type (01 binary)



EDAT facility 2 also introduces the COMPARE AND REPLACE DAT TABLE ENTRY (CRDTE) instruction. CRDTE combines the storage-update operations of a compare-and-swap operation with the TLB purging that is associated with INVALIDATE DAT TABLE ENTRY or INVALIDATE PAGE TABLE ENTRY.

Normally, when an operating system needs to update the DAT table entry of an attached DAT table (that is, a table that may actively be used by the CPU for translation), then it must first invalidate the table entry and purge any TLB entries formed from it, make the necessary changes, and revalidate the entry. This assures that stale entries in the TLB won't accidentally be used by other CPUs, while one CPU makes the changes.

CRDTE combines these functions into a single instruction, improving the performance of such updates for all levels of DAT-table entries.

Enhanced-DAT Facility 3: CRDTE

- **CRDTE operands (similar to IDTE)**

Register	Contents
R_1	Compare Value
R_1+1	Replacement Value
R_2	Page-Table Origin (when DTT = 0) | /// /// | DTT | //
R_2	Region- or Segment-Table Origin (when DTT = 4-7) | /// //// | DTT | //
R_2+1	RFX | RSX | RTX | SX | PX | 0000 0000 0000
R_3	Region- or segment-table origin of ASCE for clearing (when R3 ≠ 0) | //// //// | DT | //
M_4	/// | LC

Explanation:

DT	Designation type	RSX	Region 2nd index
DTT	Designated table type	RTX	Region 3rd index
LC	Local-clearing control	SX	Segment index
RFX	Region 1st index	PX	Page index

The operands to CRDTE are similar to those used by COMPARE AND SWAP, INVALIDATE DAT TABLE ENTRY, and INVALIDATE PAGE TABLE ENTRY, as shown in this slide.

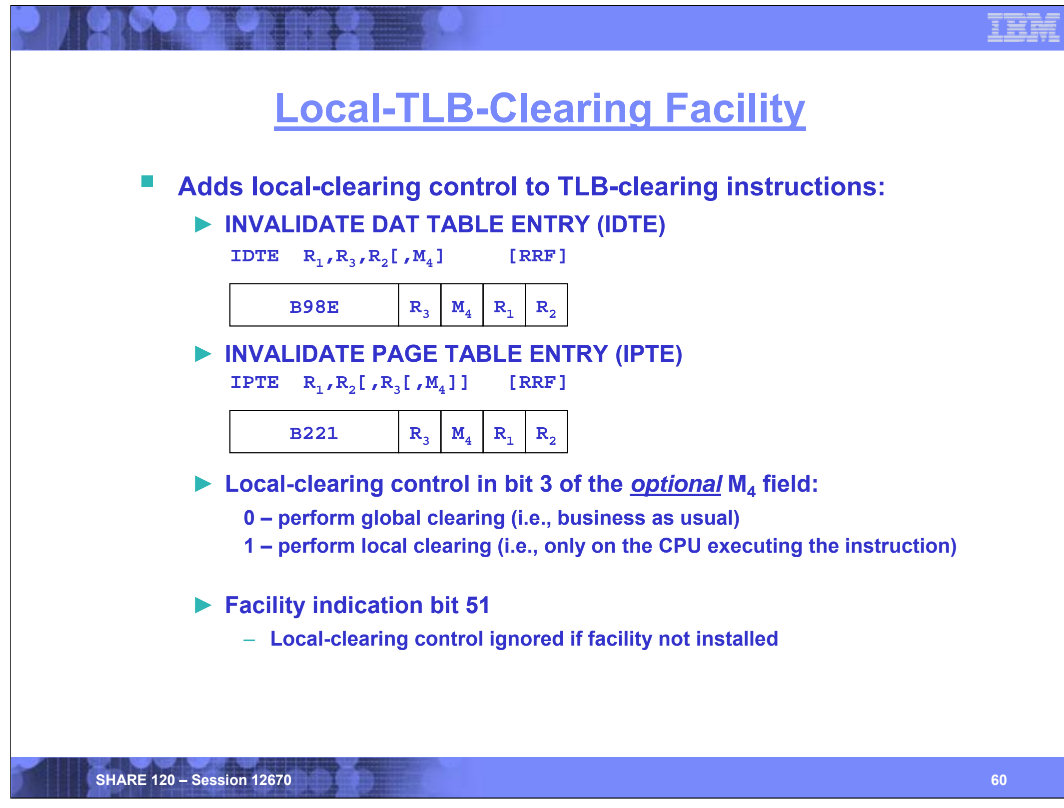
The first operand comprises an even / odd general register pair containing the compare value and the replacement value for a DAT-table entry designated by the second operand.

The second operand contains an even / odd general register pair designating the location of the DAT-table entry to be replaced. The even-numbered register contains the base address of the table, and a designated-table-type (DTT) indication. The odd-numbered register is in the form of a virtual address; based on the DTT, the appropriate portion of the virtual address (RFX, RSX, RTX, SX, or PX) is used to locate the entry in the table.

Assuming the comparison is equal, the table entry is replaced using a block-concurrent interlocked update, and the TLBs of all CPUs are cleared of at least that entry and any subordinate entries.

When the R_3 field is nonzero, the clearing can be restricted to a particular ASCE

Additionally, CRDTE contains a M_4 field containing a local-clearing control. When this bit is one, clearing is restricted to the CPU on which the instruction executes. This may avoid disruptive clearing in a uniprocessor environment or in an MP where an address space is only dispatched on a single CPU. More on local clearing in the following slides.

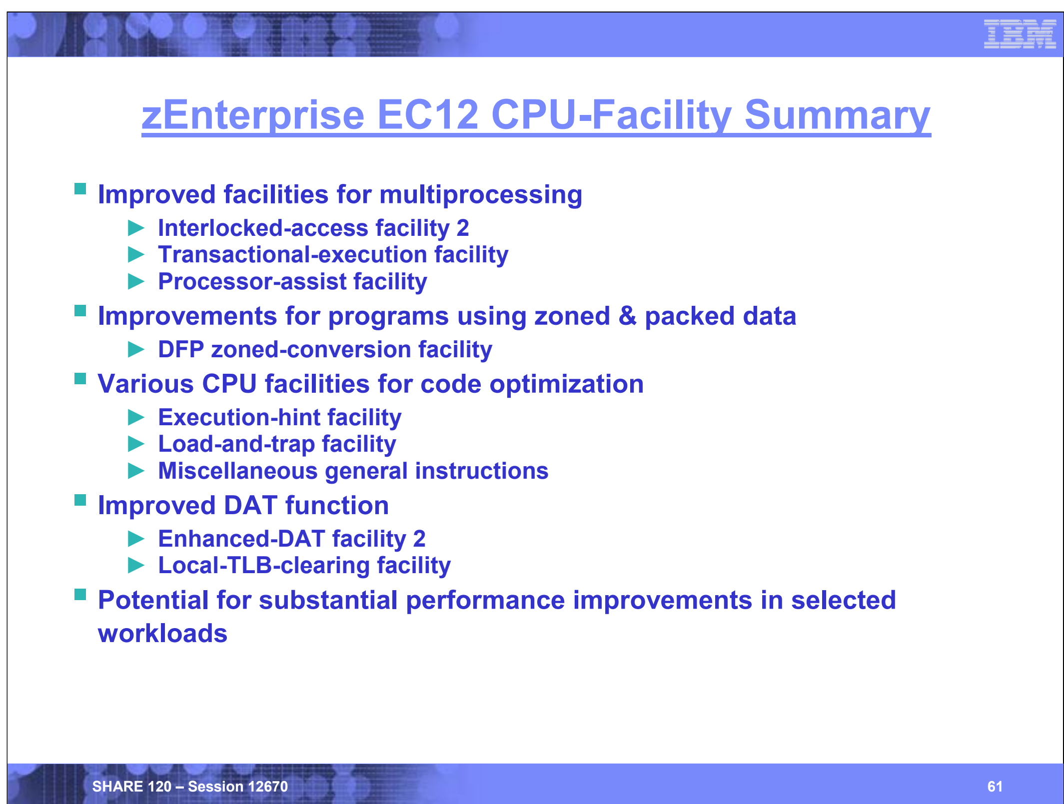


The INVALIDATE DAT TABLE ENTRY and INVALIDATE PAGE TABLE ENTRY instructions have both been enhanced to provide a local-clearing control.

In the respective instruction images shown above, this is the M_4 field of the instruction. In the assembler syntax, this field is optional, and is effectively zero if the field is not coded.

The local-clearing control operates as described on the previous slide for COMPARE AND REPLACE DAT TABLE ENTRY: if the bit is zero, global clearing occurs on all CPUs; if the bit is one, local clearing on the CPU executing the instruction occurs.

If the local-TLB-clearing facility is not installed (as indicated by facility indication bit 51 being zero), then setting the LC bit to one on either of these instructions is ignored.



The IBM zEnterprise EC12 system introduces several powerful new CPU facilities as enumerated on this slide. The most significant of these changes is the transactional-execution facility which provides a game-changing paradigm in multiprocessor serialization, and has the potential of providing significant performance improvement for selected workloads. The interlocked-access facility 2 makes it significantly easier to manipulate shared data – without the bother of using locks or compare-and-swap type of operations.

Other enhancements provide improvements in legacy workload handling (DFP zoned-conversion facility), pipeline optimization (execution hint facility and miscellaneous-general-instructions facility), data validation (load-and-trap facility), and virtual-storage management (EDAT-2).

In combination, these facilities set the stage for improved CPU throughput.

For those in the audience, (a) give yourself a pat on the back for enduring an intense hour of discussion on the zEC12, and (b) I will gladly entertain any questions now.

For those reading this from the SHARE web site, if you have further questions, you are welcome to e-mail me at dgreiner@us.ibm.com.