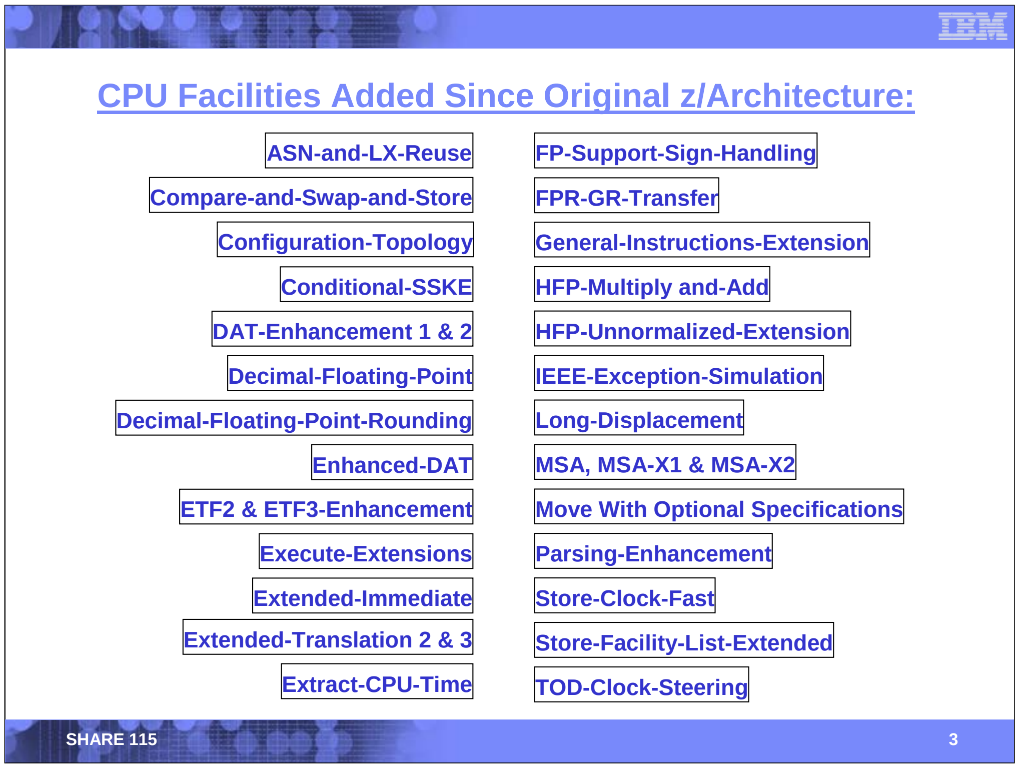
New z/Architecture Instructions that Can Save You Time & Effort

Dan Greiner
dgreiner@us.ibm.com
z/Server Architecture
SHARE 115 in Boston
2 August 2010

IBM Systems and Technology Group (STG)

Note: This is a PowerPoint presentation which contains a significant amount of animation to help illustrate the concepts described. SHARE proceedings are usually restricted to Adobe portable-document-format (.pdf) files. If you would like a copy of the original PowerPoint slide show, please see me after the session or send me an email at the address on the cover page.

The Legal Stuff



CPU Facilities Added Since Original z/Architecture:

ASN-and-LX-Reuse	FP-Support-Sign-Handling
Compare-and-Swap-and-Store	FPR-GR-Transfer
Configuration-Topology	General-Instructions-Extension
Conditional-SSKE	HFP-Multiply and-Add
DAT-Enhancement 1 & 2	HFP-Unnormalized-Extension
Decimal-Floating-Point	IEEE-Exception-Simulation
Decimal-Floating-Point-Rounding	Long-Displacement
Enhanced-DAT	MSA, MSA-X1 & MSA-X2
ETF2 & ETF3-Enhancement	Move With Optional Specifications
Execute-Extensions	Parsing-Enhancement
Extended-Immediate	Store-Clock-Fast
Extended-Translation 2 & 3	Store-Facility-List-Extended
Extract-CPU-Time	TOD-Clock-Steering

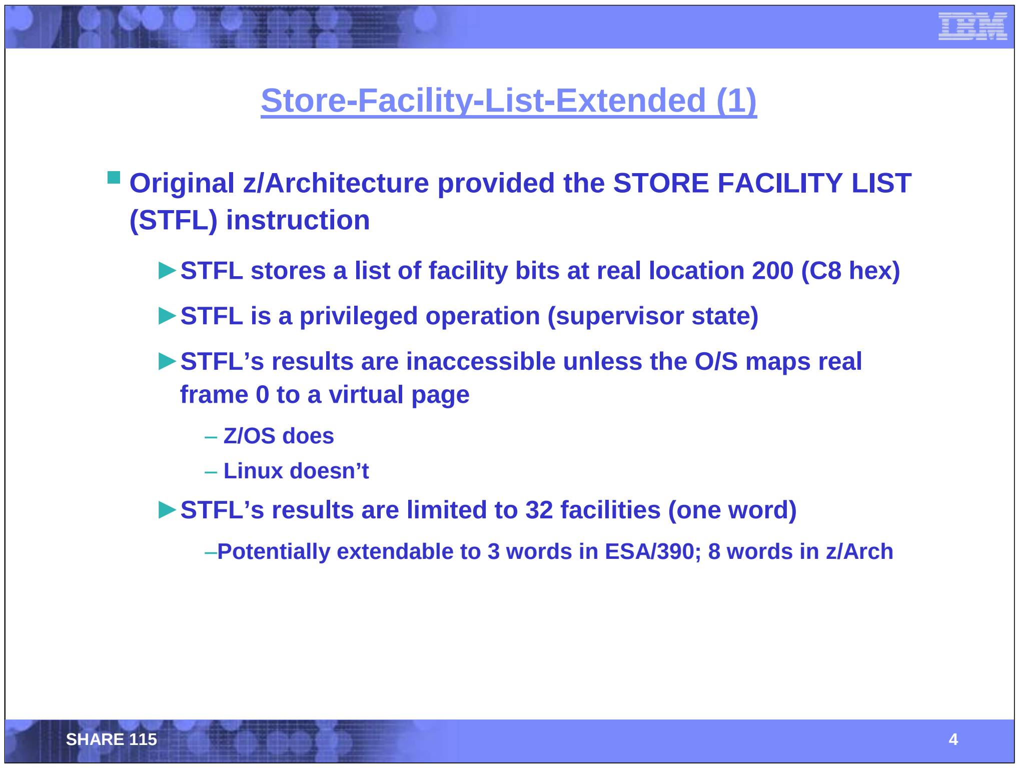
SHARE 115 3

This slide lists the major CPU facilities that have been added to z/Architecture since its introduction in 2000. Many of these facilities are targeted to improving performance, but in the period of a one-hour SHARE presentation, we will not have the time to address all of these facilities (it is also doubtful whether the attention span of the audience would endure a full presentation).

If you are interested in additional details on facilities that are not discussed in this presentation, there are two presentations from SHARE 113 that provide a review of all of the new CPU facilities:

1. Session 1290 – Additions to z/Architecture
2. Session 1291 - Additions to z/Architecture in the IBM System z10 Enterprise Class

Some material from those presentations has been incorporated into this presentation.



Store-Facility-List-Extended (1)

- Original z/Architecture provided the STORE FACILITY LIST (STFL) instruction
  - STFL stores a list of facility bits at real location 200 (C8 hex)
  - STFL is a privileged operation (supervisor state)
  - STFL's results are inaccessible unless the O/S maps real frame 0 to a virtual page
    - Z/OS does
    - Linux doesn't
  - STFL's results are limited to 32 facilities (one word)
    - Potentially extendable to 3 words in ESA/390; 8 words in z/Arch

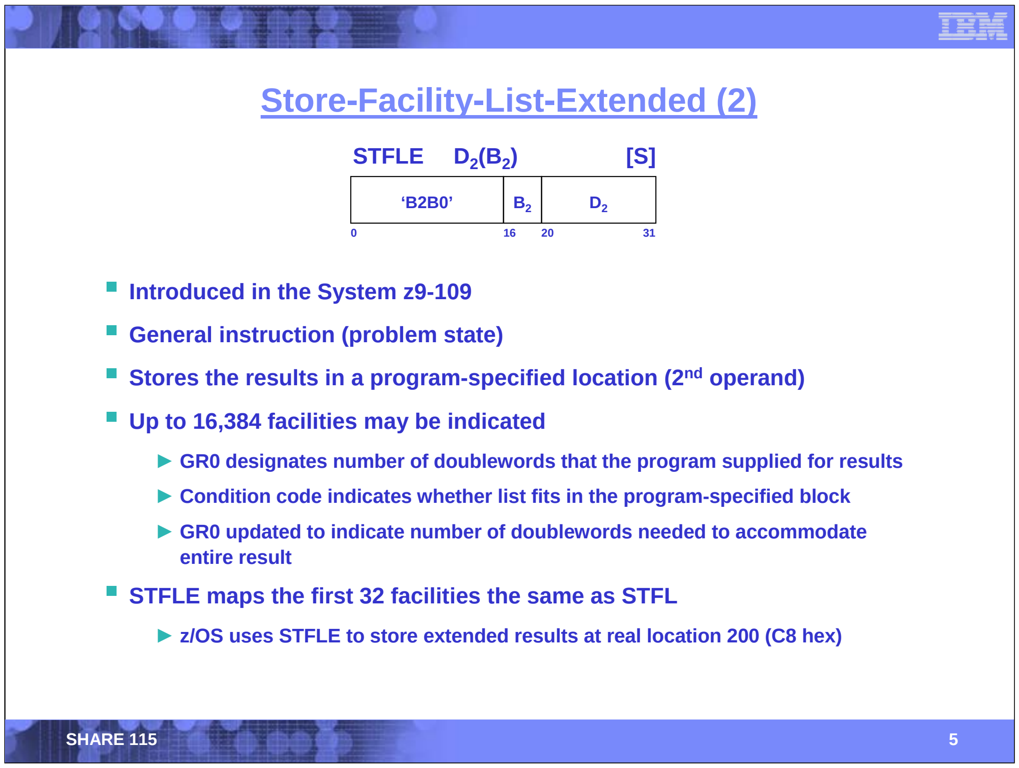
SHARE 115 4

Before discussing particular facilities, it would be worth spending some time on how to determine if a facility is installed.

The original STORE FACILITY LIST (STFL) instruction has several limitations:

- Only 32 facility bits are supported.
- The results are placed in real storage location 200 (C8 hex). Although z/OS maps real page 0 in a V=R manner, Linux does not.
- STFL is a privileged instruction.

Thus, for environments such as Linux, a costly system call is required to determine what hardware facilities are available.



Store-Facility-List-Extended (2)

STFLE $D_2(B_2)$ [S]

'B2B0'	B_2	D_2
0	16	20 … 31

- Introduced in the System z9-109
- General instruction (problem state)
- Stores the results in a program-specified location (2nd operand)
- Up to 16,384 facilities may be indicated
  - GR0 designates number of doublewords that the program supplied for results
  - Condition code indicates whether list fits in the program-specified block
  - GR0 updated to indicate number of doublewords needed to accommodate entire result
- STFLE maps the first 32 facilities the same as STFL
  - z/OS uses STFLE to store extended results at real location 200 (C8 hex)

SHARE 115 5

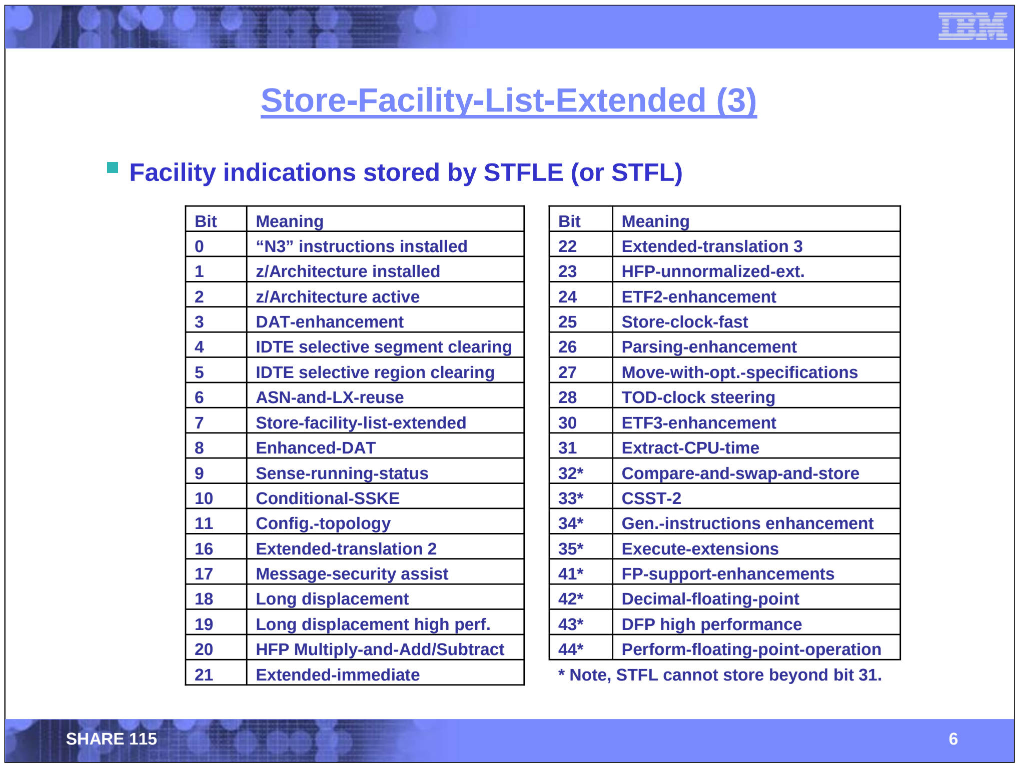
▪STFLE is a general instruction, thus any application can execute it.

▪The results are stored in a program-specified location.

▪Up to 16K of facility indications may be indicated (256 doublewords; 1 bit per facility indication).

▪The first 32 facility indications are identical to that provided by STFL.

z/OS continues to store facility indications at location 200, but now it uses STFLE instead of STFL. Thus facilities 32 and above may be indicated in real locations 204 and up, such that the z/OS application does not need STFLE at all.



Store-Facility-List-Extended (3)

- Facility indications stored by STFLE (or STFL)

Bit	Meaning
0	“N3” instructions installed
1	z/Architecture installed
2	z/Architecture active
3	DAT-enhancement
4	IDTE selective segment clearing
5	IDTE selective region clearing
6	ASN-and-LX-reuse
7	Store-facility-list-extended
8	Enhanced-DAT
9	Sense-running-status
10	Conditional-SSKE
11	Config.-topology
16	Extended-translation 2
17	Message-security assist
18	Long displacement
19	Long displacement high perf.
20	HFP Multiply-and-Add/Subtract
21	Extended-immediate
22	Extended-translation 3
23	HFP-unnormalized-ext.
24	ETF2-enhancement
25	Store-clock-fast
26	Parsing-enhancement
27	Move-with-opt.-specifications
28	TOD-clock steering
30	ETF3-enhancement
31	Extract-CPU-time
32*	Compare-and-swap-and-store
33*	CSST-2
34*	Gen.-instructions enhancement
35*	Execute-extensions
41*	FP-support-enhancements
42*	Decimal-floating-point
43*	DFP high performance
44*	Perform-floating-point-operation

* Note, STFL cannot store beyond bit 31.

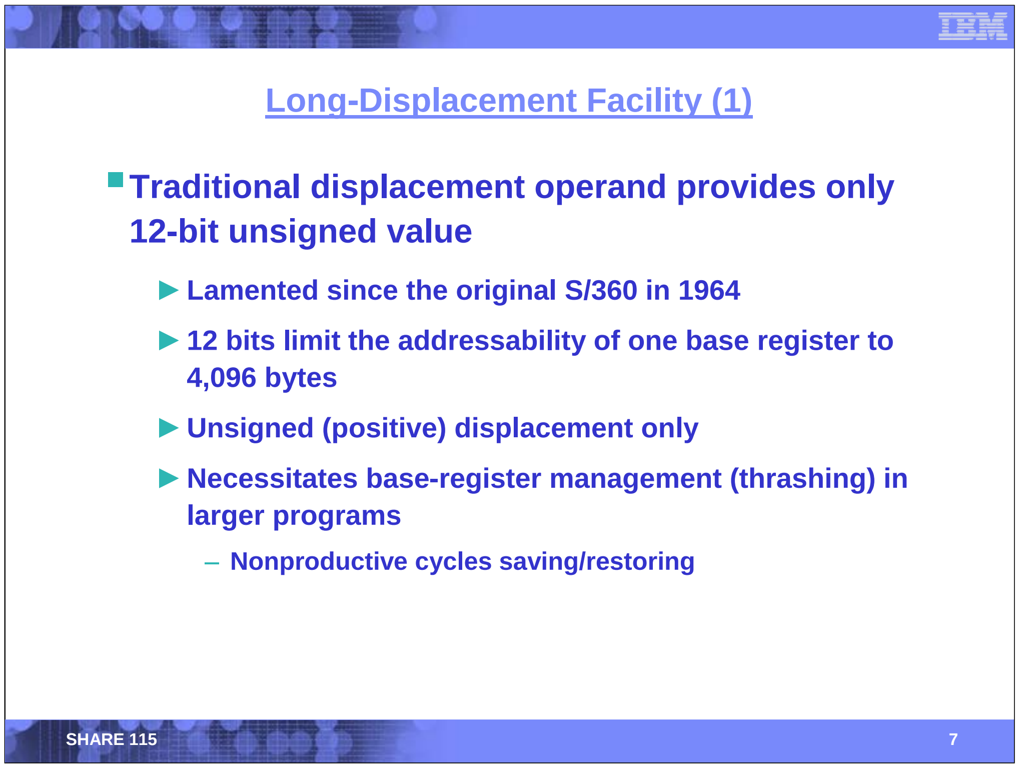
SHARE 115 6

This slide enumerates the currently-defined facility bits, as stored by the STFL or STFLE instructions. Note, STFL is now deprecated, as it can only store the first 32 facility indications.

A program that needs to examine facility indications should execute STFLE (or STFL) only once! Subsequent examination of the facility indications should be done via bit testing instructions such as TEST UNDER MASK (TM), using the program’s own copy of the bits stored by STFLE.

In the case of z/OS, the operating system uses STFLE to store the facility indications beginning at real location 200 (C8 hex) – the same place that STFL stores the first 32 facility indications. z/OS maps this location to the equivalent virtual address in address spaces from which instructions can be executed (i.e., primary or home).

Obviously, if your program is going to be run only on a processor in which a facility is known to be installed, then you do not need to test for its presence.



Long-Displacement Facility (1)

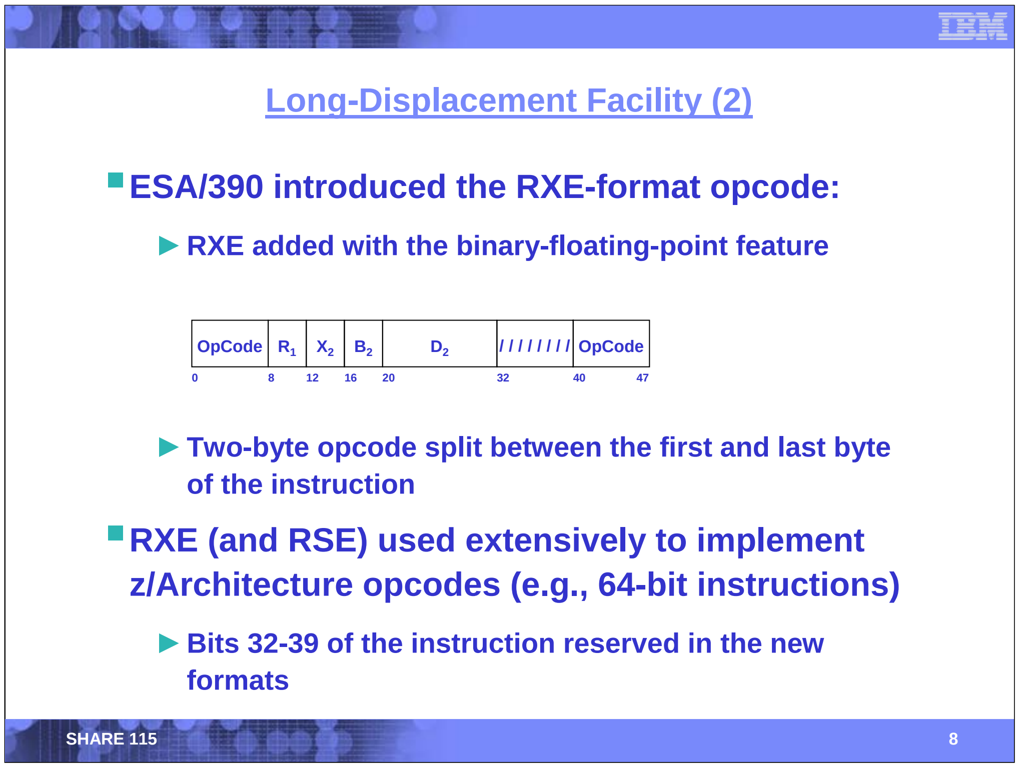
- Traditional displacement operand provides only 12-bit unsigned value
  - Lamented since the original S/360 in 1964
  - 12 bits limit the addressability of one base register to 4,096 bytes
  - Unsigned (positive) displacement only
  - Necessitates base-register management (thrashing) in larger programs
    - Nonproductive cycles saving/restoring

SHARE 115 7

Ever since programmers started writing assembler-language code for the System 360, a common complaint has been that the 12-bit unsigned displacement provided by common storage-accessing instructions is insufficient. The 12-bit displacement field allows only a positive offset of up to 4,095 bytes.

This means that for larger programs, a single base register is inadequate, necessitating a second (or third [or fourth]) register be committed as a program base. This issue affects not only the code, but also any data areas that may exceed 4K.

The 4K limitation puts pressure on the 16 general-purpose registers. For some programs, a significant amount of code – and execution cycles – are devoted to register management. Compared to an architecture that has more registers, the register-management operations are unproductive cycles, wasted to accommodate a limited architecture.



Long-Displacement Facility (2)

- ESA/390 introduced the RXE-format opcode:
  - RXE added with the binary-floating-point feature

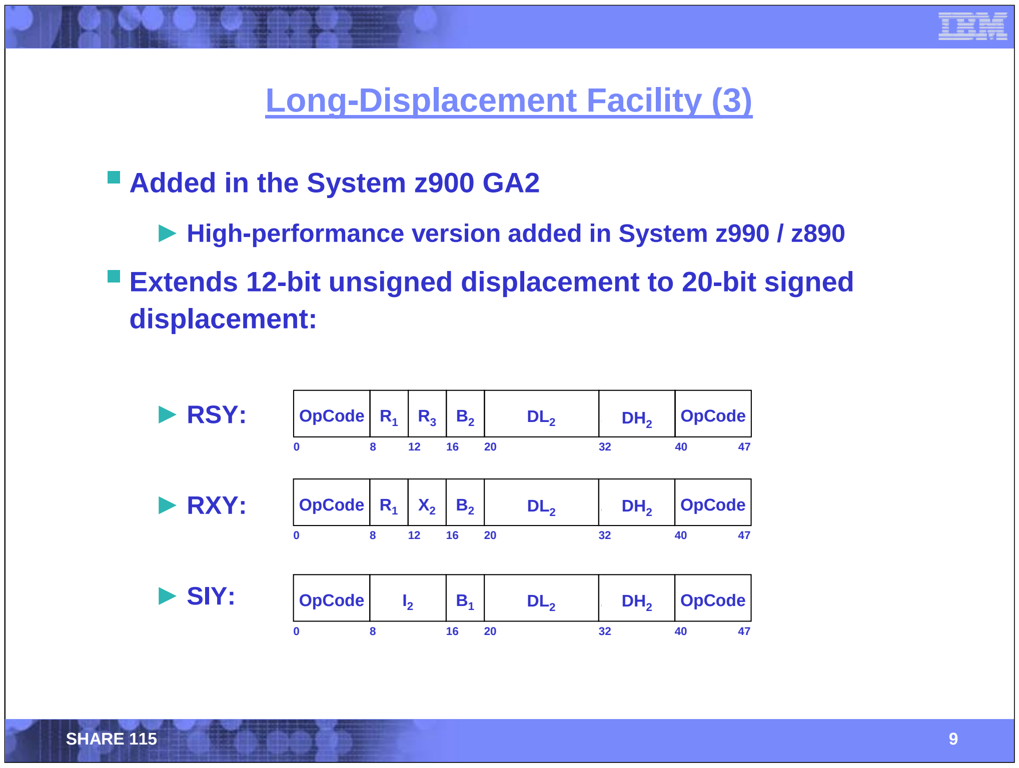
OpCode	R_1	X_2	B_2	D_2	////////	OpCode
0	8	12	16	20	32	40 47

  - Two-byte opcode split between the first and last byte of the instruction
- RXE (and RSE) used extensively to implement z/Architecture opcodes (e.g., 64-bit instructions)
  - Bits 32-39 of the instruction reserved in the new formats

SHARE 115 8

The binary floating-point facility in ESA/390 introduced the RXE instruction format. Bits 8-31 of the RXE format provide the same register, base, index, and displacement fields as the RX format, however the opcode is 16 bits – split between the first and last bytes of the instruction. Bits 32-39 of the instruction are reserved.

With the advent of z/Architecture, the RS instruction format was similarly extended to form the RSE format. The RSE and RXE instruction formats were used extensively in implementing the new 64-bit architecture.



Long-Displacement Facility (3)

- Added in the System z900 GA2
  - High-performance version added in System z990 / z890
- Extends 12-bit unsigned displacement to 20-bit signed displacement:
  - RSY:

OpCode	R_1	R_3	B_2	DL_2	DH_2	OpCode
0	8	12	16	20	32	40 47

  - RXY:

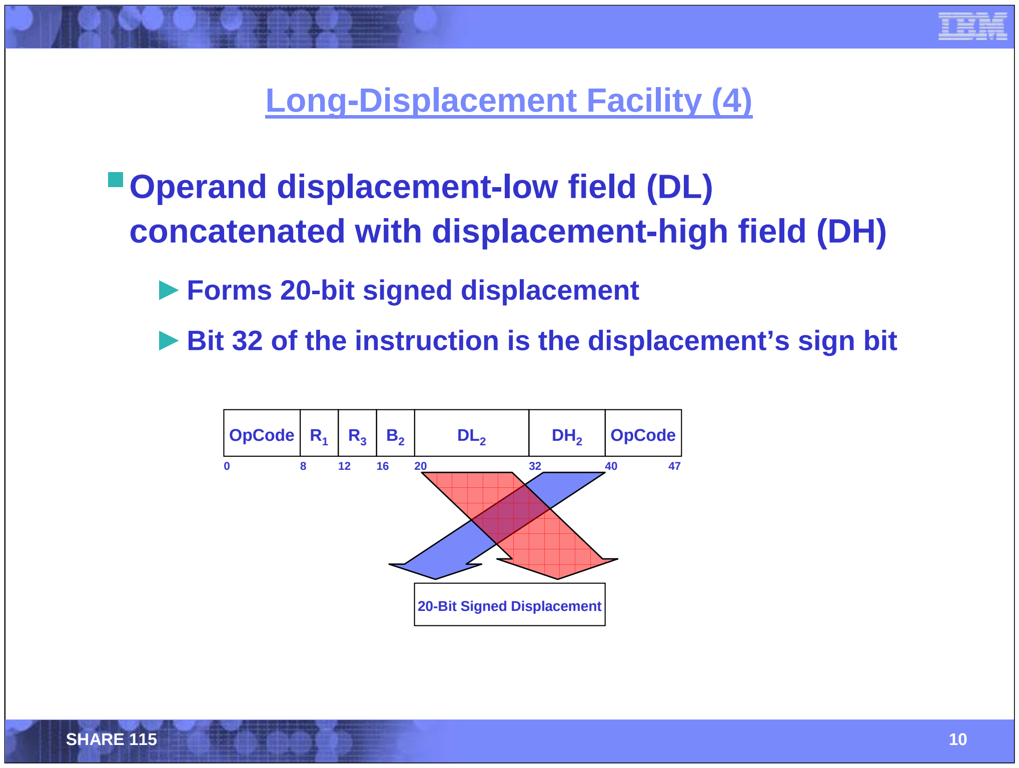
OpCode	R_1	X_2	B_2	DL_2	DH_2	OpCode
0	8	12	16	20	32	40 47

  - SIY:

OpCode	I_2	B_1	DL_2	DH_2	OpCode
0	8	16	20	32	40 47

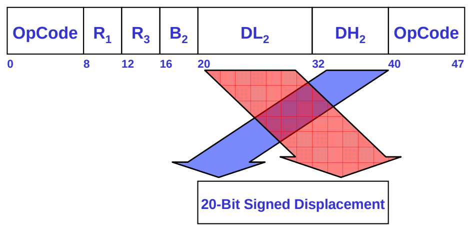
SHARE 115 9

The long-displacement facility builds upon the RSE and RXE instruction formats introduced in z/Architecture. The new RSY and RXY instruction formats have all of the same fields as the RSE and RXE instructions, but with an additional field occupying the previously-reserved bits 32-39. A new SIY format, a long-displacement analog to the SI format, is also introduced.



Long-Displacement Facility (4)

- Operand displacement-low field (DL) concatenated with displacement-high field (DH)
  - Forms 20-bit signed displacement
  - Bit 32 of the instruction is the displacement's sign bit

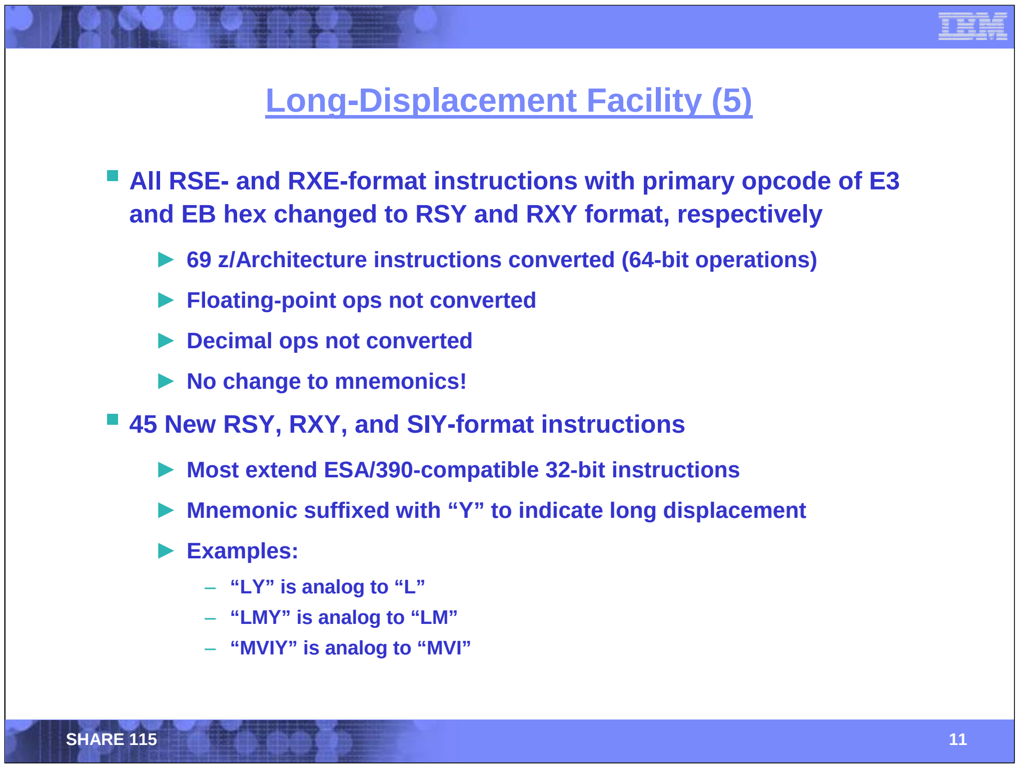


Prior to the long-displacement facility, the displacement field in an instruction was a 12-bit unsigned field, providing a displacement range from 0-4,095 bytes.

The new formats contain a 20-bit **signed** displacement, thus providing a positive or negative displacement of 512K.

Bits 32-39 of the instruction form the displacement high (DH) field that provides the most-significant bits of the displacement. Bit 32 is the sign bit.

The DH field, concatenated with the displacement low field (DL, that is, the classic 12-bit displacement in bits 20-31) form the 20-bit signed value.



Long-Displacement Facility (5)

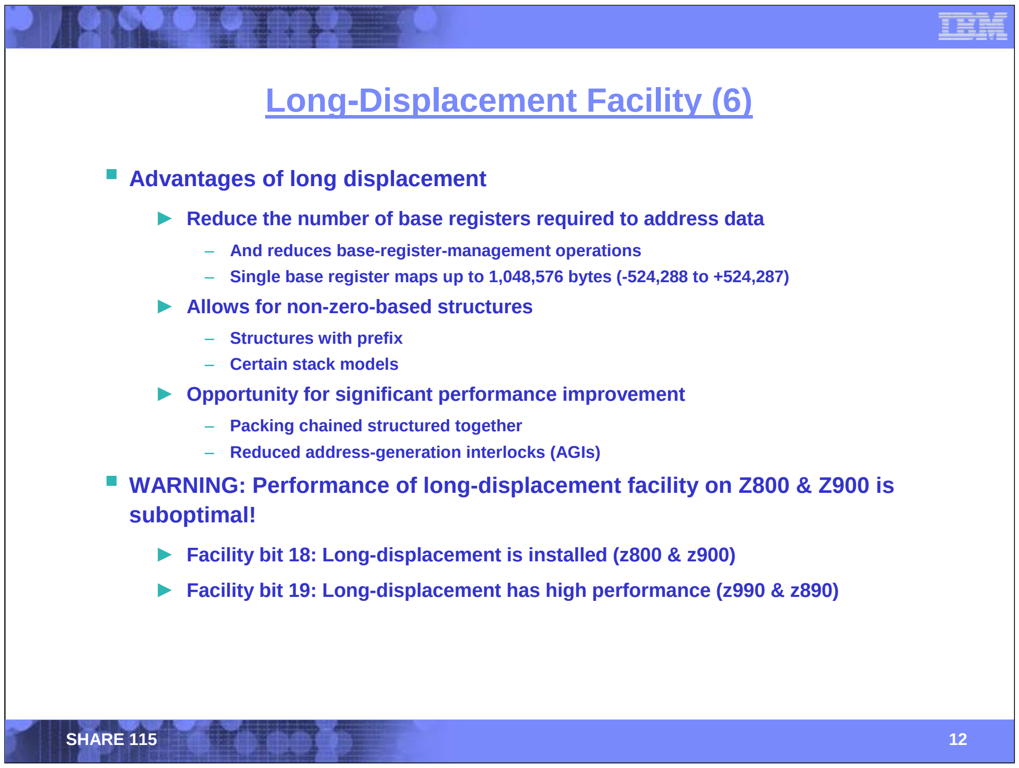
- All RSE- and RXE-format instructions with primary opcode of E3 and EB hex changed to RSY and RXY format, respectively
  - 69 z/Architecture instructions converted (64-bit operations)
  - Floating-point ops not converted
  - Decimal ops not converted
  - No change to mnemonics!
- 45 New RSY, RXY, and SIY-format instructions
  - Most extend ESA/390-compatible 32-bit instructions
  - Mnemonic suffixed with “Y” to indicate long displacement
  - Examples:
    - “LY” is analog to “L”
    - “LMY” is analog to “LM”
    - “MVIY” is analog to “MVI”

SHARE 115 11

A significant number of the z/Architecture RSE and RXE instructions (that is, those that provided 64-bit support) were converted to long displacement (RSY and RXY format). Instruction-level compatibility for programs developed using 12-bit displacements is assured, since HLASM will generate zeros for the reserved fields.

Decimal and floating-point operations were not converted to long displacement.

New instructions were defined to provide long-displacement analogs for most of the 32-bit RS, RX, and SI instructions (that is, those ported to z/Architecture from ESA/390). The letter “Y” was appended to the mnemonic to indicate the long-displacement form. For example, the 32-bit LOAD includes both L and LY.



Long-Displacement Facility (6)

- Advantages of long displacement
  - Reduce the number of base registers required to address data
    - And reduces base-register-management operations
    - Single base register maps up to 1,048,576 bytes (-524,288 to +524,287)
  - Allows for non-zero-based structures
    - Structures with prefix
    - Certain stack models
  - Opportunity for significant performance improvement
    - Packing chained structured together
    - Reduced address-generation interlocks (AGIs)
- WARNING: Performance of long-displacement facility on Z800 & Z900 is suboptimal!
  - Facility bit 18: Long-displacement is installed (z800 & z900)
  - Facility bit 19: Long-displacement has high performance (z990 & z890)

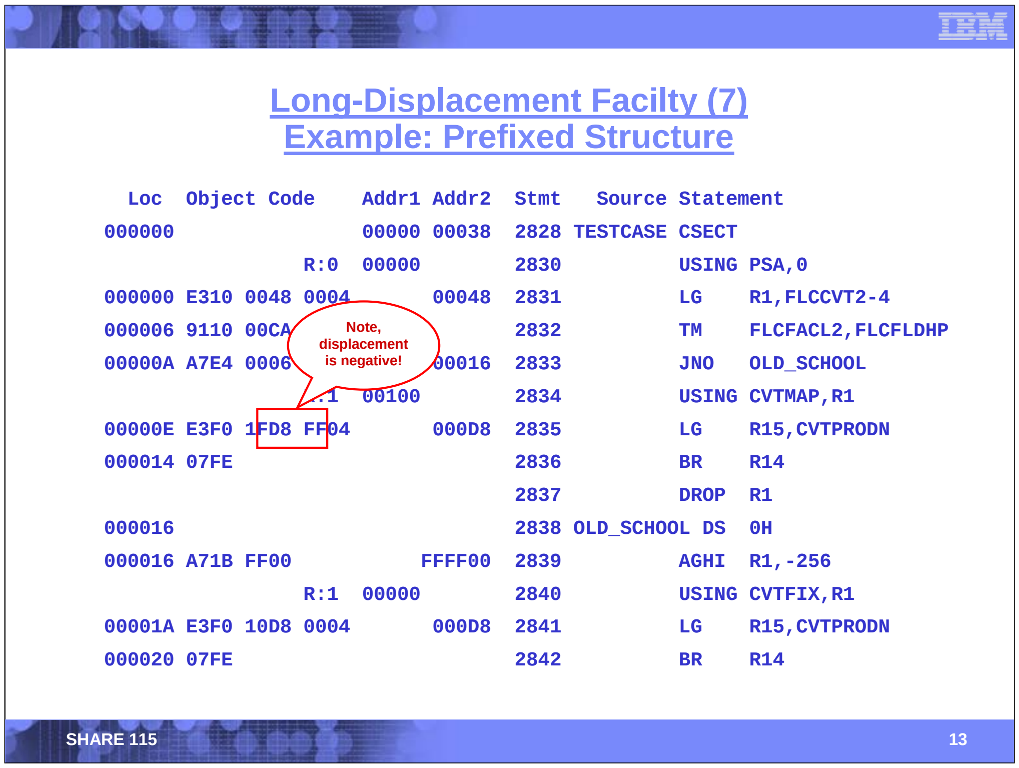
SHARE 115 12

There are numerous potential advantages to using the long-displacement facility:

•The 12-bit displacement has been the bane of assembler programmers not long after the introduction of the S/360. Code that follows chains of pointers from a base structure to extension controls blocks can (conceivably) be redesigned to consolidate such linked control structures.

•Certain control structures are not zero based. The long displacement provides an easy means of pointing a register at the nominal base on the structure, allowing the prefix portion to be referenced using a negative displacement.

However there are some potential drawbacks. The facility was first introduced with the z990 processors, where all of the facility is implemented in hardware. However, the facility was retrofit to the z800 and z900 systems, where it is implemented in Millicode. There are two separate facility indications for long displacement: the first indicates the presence of the facility, and the second indicates high performance.



Long-Displacement Facilty (7)
Example: Prefixed Structure

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement
000000                00000 00038  2828 TESTCASE CSECT
                 R:0  00000        2830          USING PSA,0
000000 E310 0048 0004       00048  2831          LG    R1,FLCCVT2-4
000006 9110 00CA                   2832          TM    FLCFACL2,FLCFLDHP
00000A A7E4 0006            00016  2833          JNO   OLD_SCHOOL
                 R:1  00100        2834          USING CVTMAP,R1
00000E E3F0 1FD8 FF04       000D8  2835          LG    R15,CVTPRODN
000014 07FE                        2836          BR    R14
                                   2837          DROP  R1
000016                             2838 OLD_SCHOOL DS  0H
000016 A71B FF00           FFFF00  2839          AGHI  R1,-256
                 R:1  00000        2840          USING CVTFIX,R1
00001A E3F0 10D8 0004       000D8  2841          LG    R15,CVTPRODN
000020 07FE                        2842          BR    R14
```

SHARE 115 13

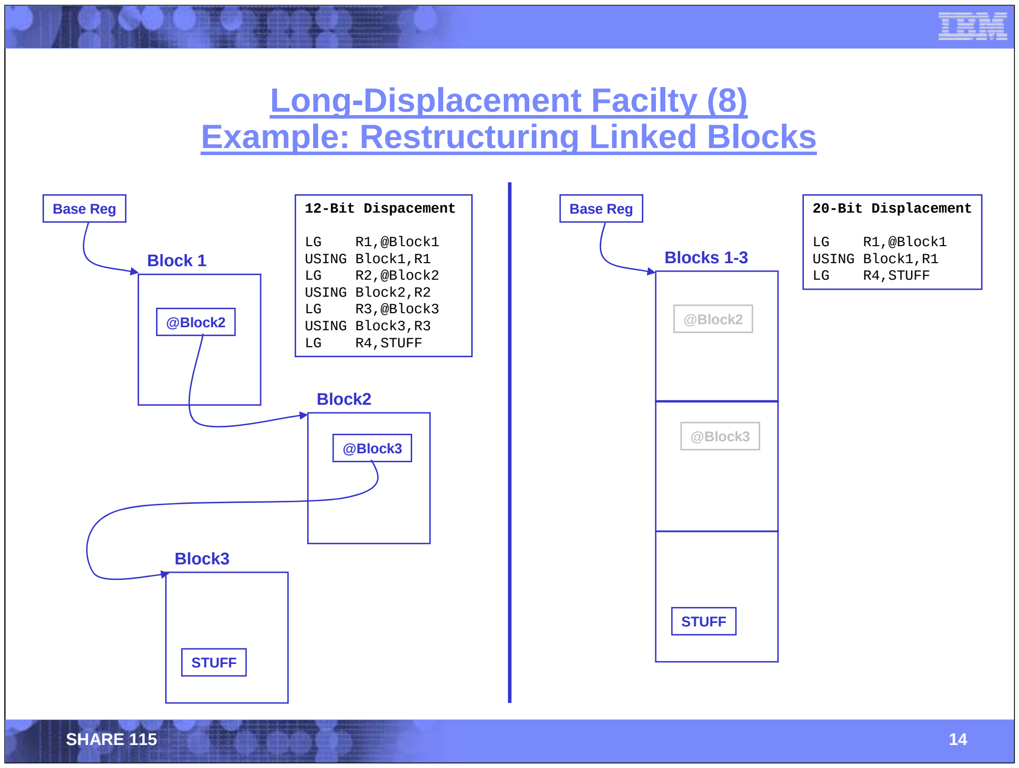
This slide illustrates one possible means of exploiting the long-displacement facility to provide addressability to the prefix area of a structure, without reloading a pointer. The example is CVTPRODN field from the prefixed portion of the z/OS communications vector table (CVT). The prefixed portion appears at an address lower than the base pointer to the CVT.

In z/OS, various locations in the first page contain a pointer to the CVT. The most commonly used pointer is the fullword at location 16 (10 hex). This example uses an alternate fullword pointer (FLCCVT2) at location 76 (4C hex), however the code actually performs a 64-bit LOAD (LG) from location 72 (48 hex), as z/OS sets locations 72-75 to zeros. On some processors, this technique may be a cycle faster than using LOAD LOGICAL THIRTY ONE BITS (LLGT).

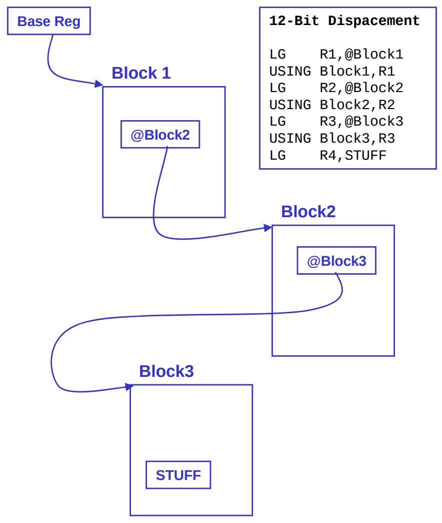
The TEST UNDER MASK instruction tests the facility bits that z/OS stores beginning at location 200. If the high-performance long-displacement facility is not installed, then the code branches to the ADD HALFWORD IMMEDIATE (AGHI) instruction which backs up the CVT pointer to its prefix. The subsequent LOAD (LG) instruction at statement 2,841 is based on the prefix origin.

If the high-performance long-displacement facility is installed, the branch is not taken. No adjustment of the CVT pointer is required. The LOAD (LG) instruction at statement 2,835 causes a negative displacement to be generated.

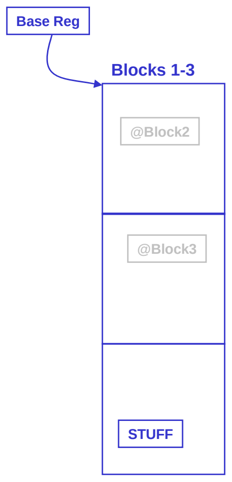
This example is for illustrative purposes only; the overhead of testing for the long-displacement facility exceeds that saved by not having to adjust the CVT pointer. However, in cases where multiple accesses to a negatively displaced field occur, additional savings may be realized.



Long-Displacement Facilty (8)
Example: Restructuring Linked Blocks



```
12-Bit Dispacement

LG    R1,@Block1
USING Block1,R1
LG    R2,@Block2
USING Block2,R2
LG    R3,@Block3
USING Block3,R3
LG    R4,STUFF
```



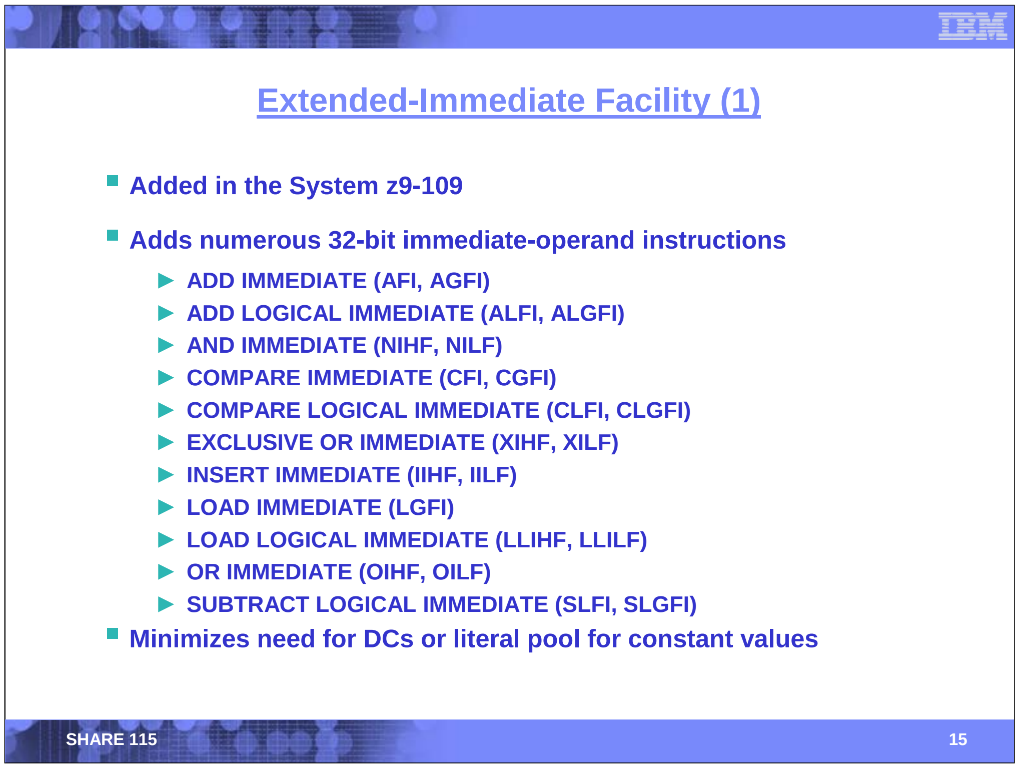
```
20-Bit Displacement

LG    R1,@Block1
USING Block1,R1
LG    R4,STUFF
```

SHARE 115 14

This slide shows how linked structures might be rearranged to exploit the long-displacement facility. Prior to the long-displacement facility, a structure that was greater than 4K in size required multiple base registers to address it. In some cases, as shown on the left, the structure would be broken into discontiguous sections, with pointers to the subsequent portions. Regardless of whether the structure was contiguous or not, access to a portion of data that was not addressable by the structure's base pointer required subsequent loading of registers to provide addressability.

The long displacement facility provides a means by which such structures may be coalesced back into a single, contiguous structure that can be addressed by a single base register. In the example shown on the left of the slide, access to the data object STUFF requires two extra loads. In the long-displacement example shown on the right, access to STUFF requires only the initial load of the pointer to the block.

Remember that the long displacement is signed; the 20-bit displacement field provides a positive or negative displacement of 512K.



Extended-Immediate Facility (1)

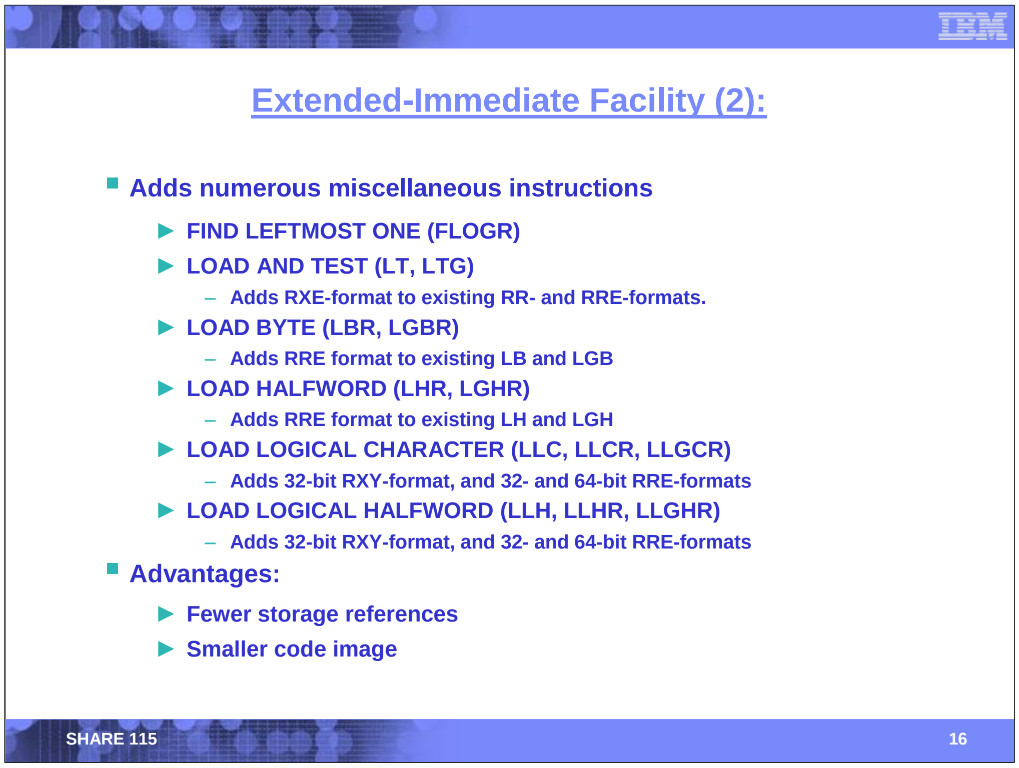
- Added in the System z9-109
- Adds numerous 32-bit immediate-operand instructions
  - ADD IMMEDIATE (AFI, AGFI)
  - ADD LOGICAL IMMEDIATE (ALFI, ALGFI)
  - AND IMMEDIATE (NIHF, NILF)
  - COMPARE IMMEDIATE (CFI, CGFI)
  - COMPARE LOGICAL IMMEDIATE (CLFI, CLGFI)
  - EXCLUSIVE OR IMMEDIATE (XIHF, XILF)
  - INSERT IMMEDIATE (IIHF, IILF)
  - LOAD IMMEDIATE (LGFI)
  - LOAD LOGICAL IMMEDIATE (LLIHF, LLILF)
  - OR IMMEDIATE (OIHF, OILF)
  - SUBTRACT LOGICAL IMMEDIATE (SLFI, SLGFI)
- Minimizes need for DCs or literal pool for constant values

SHARE 115 15

The *immediate-and-relative-instruction facility* (circa 1996) introduced a number of instructions with 16-bit immediate operands, for example LOAD HALFWORD IMMEDIATE. These ESA/390 instructions became part of the base z/Architecture.

The **extended-immediate facility** adds several instructions with 32-bit immediate fields, performing the basic arithmetic, logical, and comparison functions enumerated on this slide. One advantage of having immediate operands is that once the instruction is fetched, there is no separate fetch required for the immediate operand.

Note: The base z/Architecture provided 16-bit versions of AND IMMEDIATE and OR IMMEDIATE, but not EXCLUSIVE OR IMMEDIATE. The extended-immediate facility provides 32-bit versions of all of these instructions (no 16-bit exclusive-OR operation is needed).



Extended-Immediate Facility (2):

- Adds numerous miscellaneous instructions
  - FIND LEFTMOST ONE (FLOGR)
  - LOAD AND TEST (LT, LTG)
    - Adds RXE-format to existing RR- and RRE-formats.
  - LOAD BYTE (LBR, LGBR)
    - Adds RRE format to existing LB and LGB
  - LOAD HALFWORD (LHR, LGHR)
    - Adds RRE format to existing LH and LGH
  - LOAD LOGICAL CHARACTER (LLC, LLCR, LLGCR)
    - Adds 32-bit RXY-format, and 32- and 64-bit RRE-formats
  - LOAD LOGICAL HALFWORD (LLH, LLHR, LLGHR)
    - Adds 32-bit RXY-format, and 32- and 64-bit RRE-formats
- Advantages:
  - Fewer storage references
  - Smaller code image

SHARE 115 16

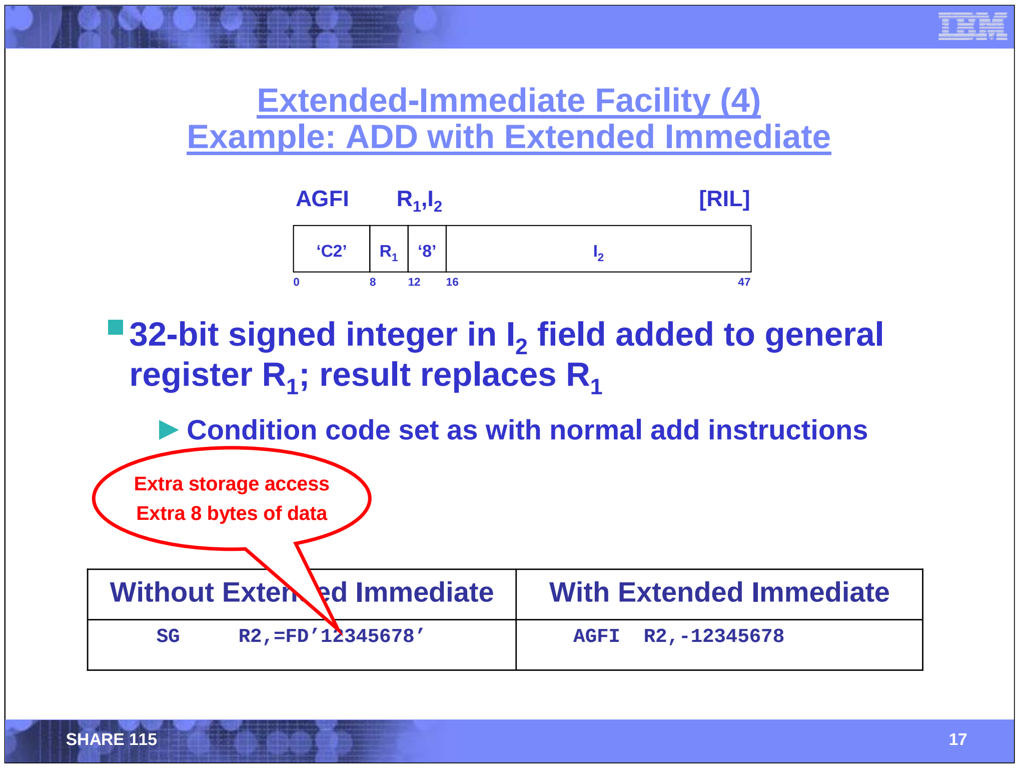
The extended-immediate facility also includes several other non-immediate-related instructions, as enumerated on this slide.

FIND LEFTMOST ONE returns the bit position of the leftmost 1 bit in a register, and the original operand with the leftmost 1 bit turned off in another register. It is particularly useful in manipulating bit maps.

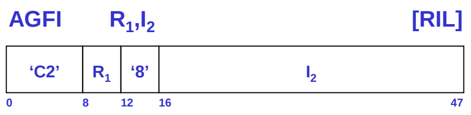
LOAD AND TEST (LT, LTG) provide for the loading from storage and testing of a value. It is similar to the combination of L/LT (or LG/LTG), but in a single instruction. Note, ICM is not truly equivalent, as it provides neither an index register nor support of 64-bit values.

The extended-immediate facility potentially improves code performance by reducing storage references (having an immediate operand fetched along with the instruction). The additional instructions listed on this slide provide additional utility in a single instruction, combining what previously might take multiple instructions to provide equivalent function.

The following slides highlight some of the instructions in the facility, showing the advantages of using them as compared to equivalent earlier instructions.



Extended-Immediate Facility (4)
Example: ADD with Extended Immediate



- 32-bit signed integer in I_2 field added to general register R_1; result replaces R_1
  - Condition code set as with normal add instructions

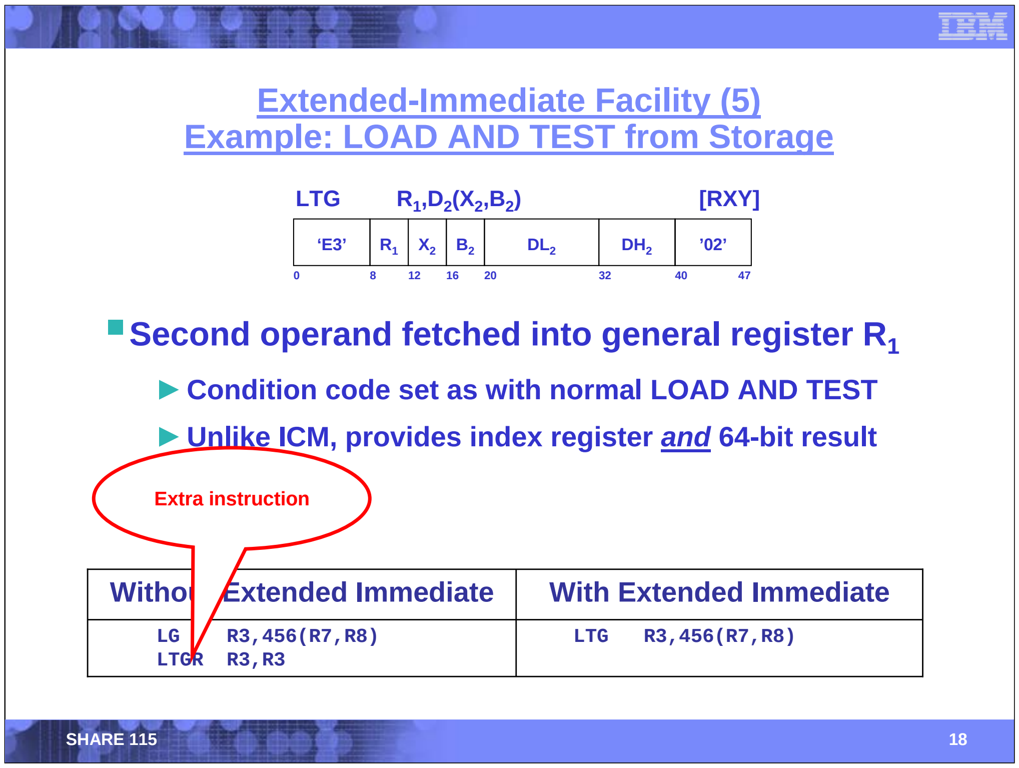
Extra storage access
Extra 8 bytes of data

Without Extended Immediate	With Extended Immediate
`SG      R2,=FD'12345678'`	`AGFI   R2,-12345678`

This slide shows the ADD IMMEDIATE (AGFI) instruction. AGFI is similar to ADD HALFWORD IMMEDIATE (AGHI), except that the second operand of AGFI is 32 bits, whereas the second operand of AGHI is a halfword.

In the comparison at the bottom of the slide, the difference is not immediately obvious – both the SUBTRACT (SG) and the AGFI are a single instruction. Because AGFI adds a signed value, it can also serve as a subtract operation by having the second operand be negative. The advantage of AGFI is that the second operand is fetched as a part of the instruction! There is additional program space required for the second-operand constant (shown on the left as a literal), and there is no need to fetch a separate second operand during execution.

Depending on the storage-access characteristics of the program, the immediate operand can have a significant performance advantage.



Extended-Immediate Facility (5)
Example: LOAD AND TEST from Storage

LTG $R_1,D_2(X_2,B_2)$ [RXY]

'E3'	R_1	X_2	B_2	DL_2	DH_2	'02'
0	8	12	16	20	32	40 47

- Second operand fetched into general register R_1
  - Condition code set as with normal LOAD AND TEST
  - Unlike ICM, provides index register ***and*** 64-bit result

Extra instruction

Without Extended Immediate	With Extended Immediate

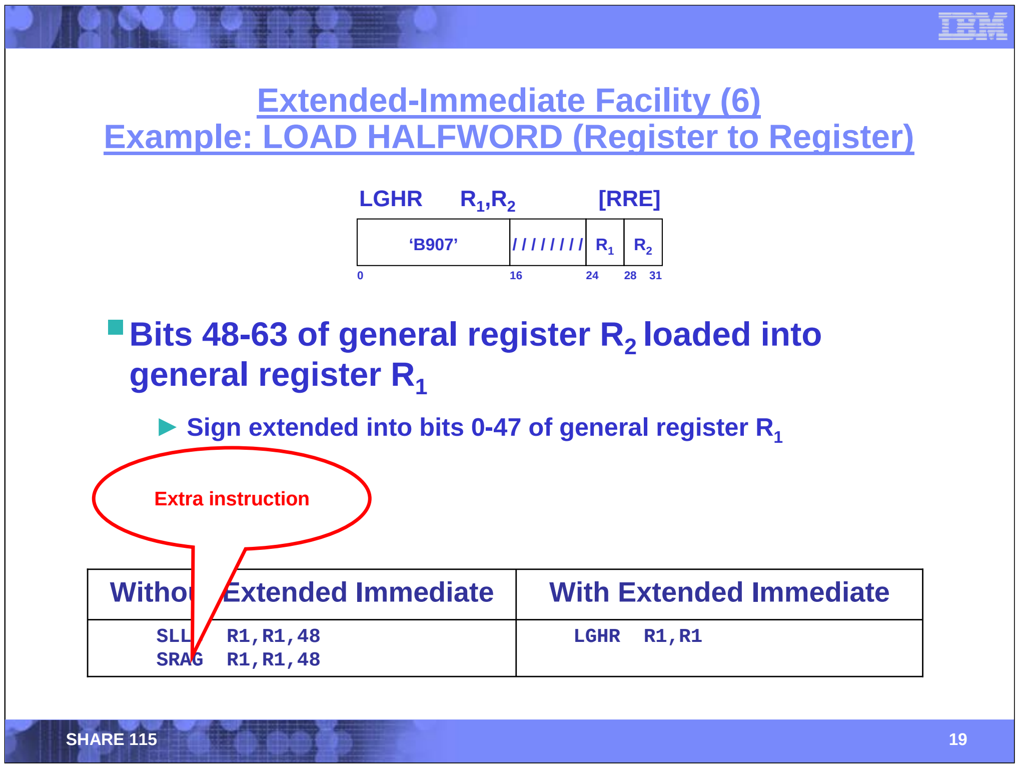
`LG R3,456(R7,R8)`
`LTGR R3,R3`
	`LTG R3,456(R7,R8)`

SHARE 115 18

The slide illustrates the LOAD AND TEST instruction with a storage operand.

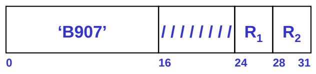
In some circumstances, the traditional LOAD/LOAD AND TEST sequence can simply be replaced with an INSERT CHARACTER UNDER MASK (ICM). However, ICM is not equivalent to the LOAD/LOAD AND TEST sequence for two reasons:

1. In this example, a 64-bit value is being loaded. ICM can only accommodate a 32-bit value.
2. ICM has no means of accommodating an index register (as is used in this example).



Extended-Immediate Facility (6)
Example: LOAD HALFWORD (Register to Register)

LGHR R_1,R_2 [RRE]



- **Bits 48-63 of general register R_2 loaded into general register R_1**
  - **Sign extended into bits 0-47 of general register R_1**

Extra instruction

Without Extended Immediate	With Extended Immediate

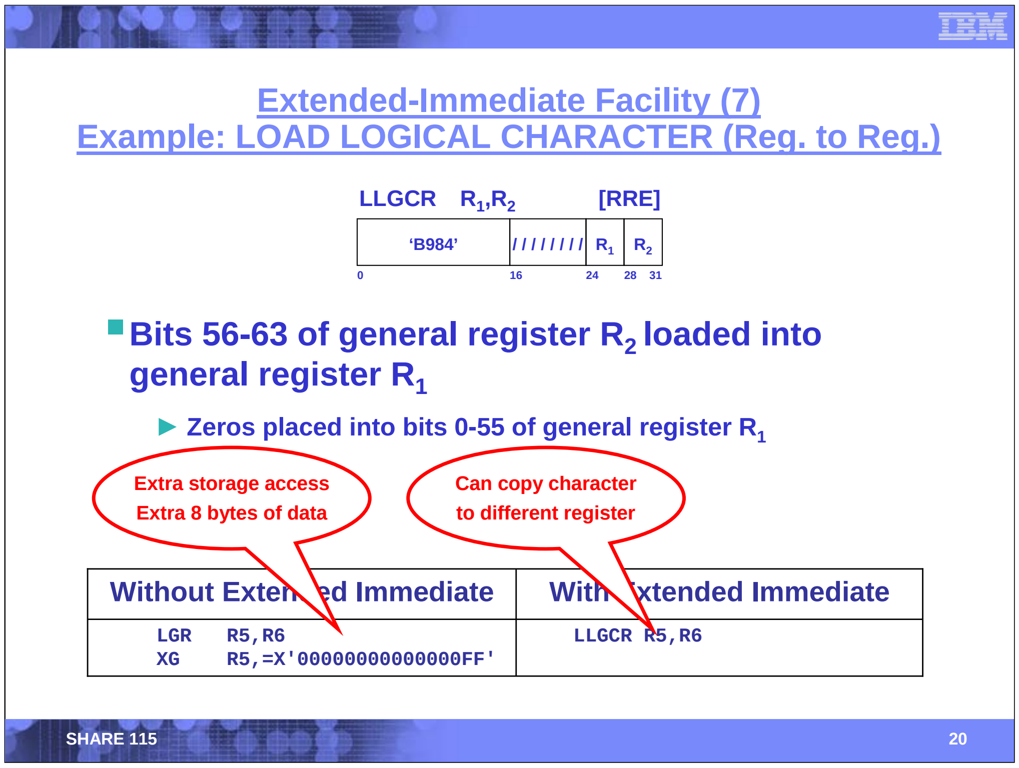
`SLLG R1,R1,48`
`SRAG R1,R1,48`
	`LGHR R1,R1`

The third example shows the isolation of bits 48-63 of a signed value in a register, where the sign is propagated to the rightmost bits.

Without the extended-immediate facility, two shift operations are required.

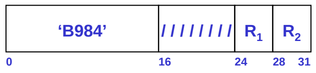
1. The first logically shifts the rightmost 16 bits to the left of the register.
2. The second arithmetically shifts the 16 bits back to the rightmost bits of the register, propagating the sign.

The register-to-register form of LOAD HALFWORD does this operation in a single instruction.



Extended-Immediate Facility (7) Example: LOAD LOGICAL CHARACTER (Reg. to Reg.)

LLGCR R_1,R_2 [RRE]



- Bits 56-63 of general register R_2 loaded into general register R_1
  - Zeros placed into bits 0-55 of general register R_1

Extra storage access
Extra 8 bytes of data

Can copy character to different register

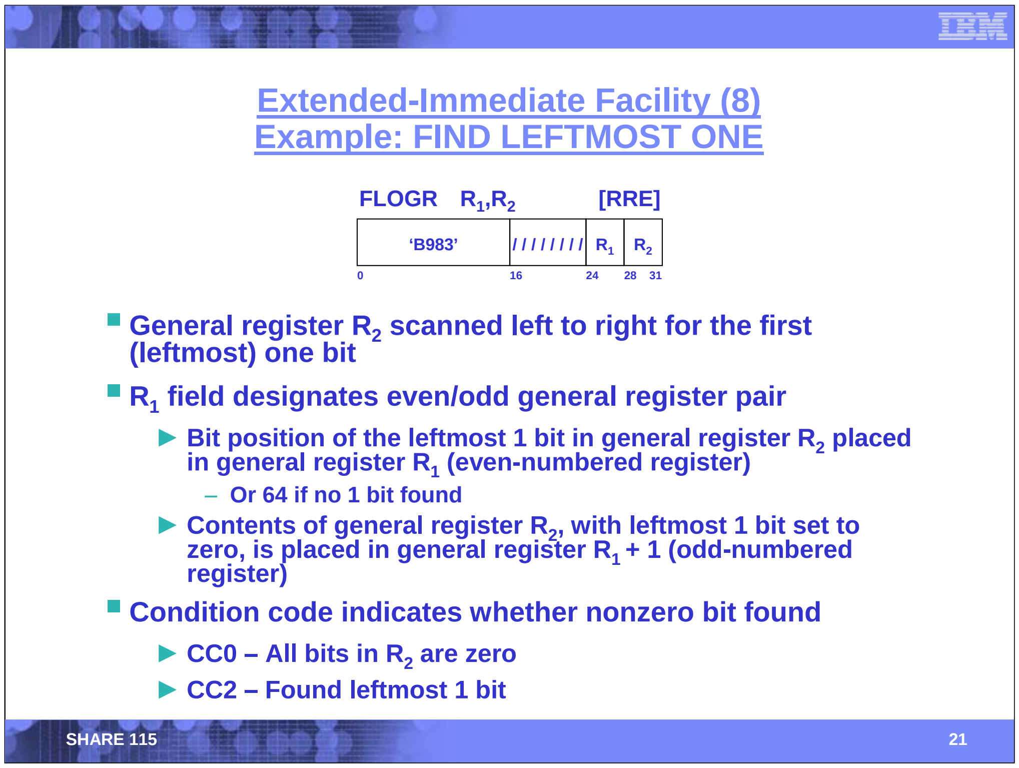
Without Extended Immediate	With Extended Immediate

`LGR   R5,R6`
`XG    R5,=X'00000000000000FF'`
	`LLGCR R5,R6`

The fourth example shows a simple isolation of bits 56-63. All other bits in the register are set to zeros.

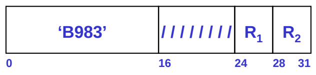
For the purposes of this example, we want to retain the original value in general register 6. In the example on the left, an extra instruction is required to perform the copy, whereas LLGCR accomplishes this feat in one instruction.

To avoid the storage reference shown in the left example, some programmers may use a pair of logical shift instructions (similar to the technique used on the previous slide). However the register-to-register LOAD LOGICAL CHARACTER instruction effects the isolation of the rightmost byte in a single instruction, without a storage reference.



Extended-Immediate Facility (8)
Example: FIND LEFTMOST ONE

FLOGR R_1,R_2 [RRE]



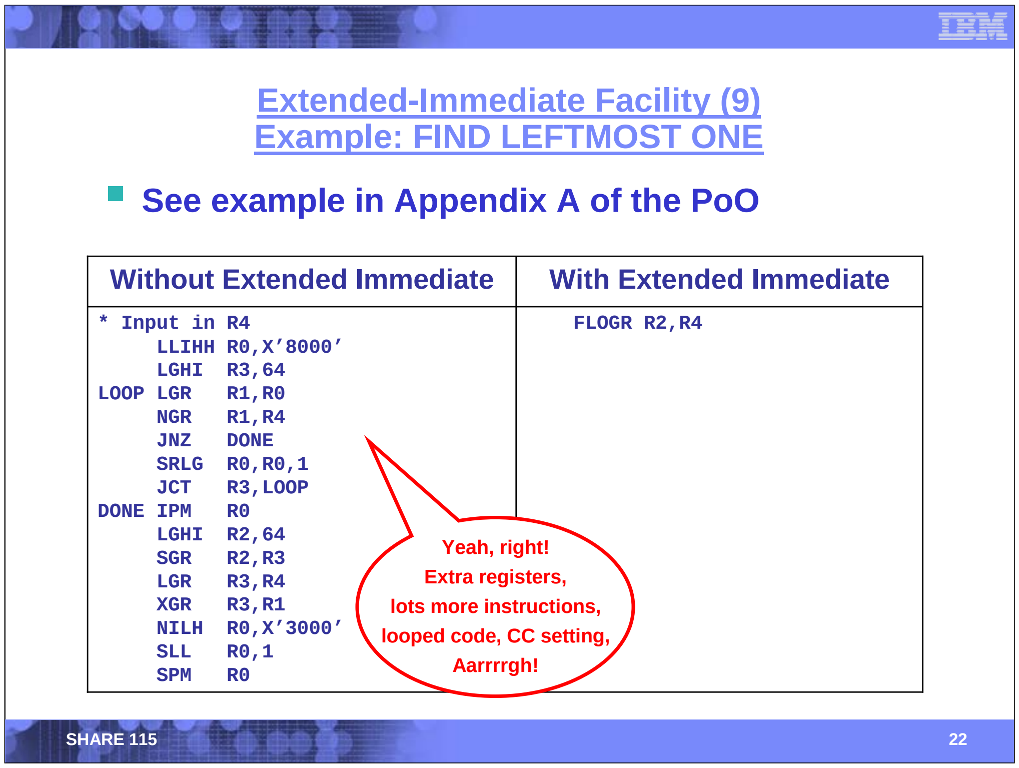
- General register R_2 scanned left to right for the first (leftmost) one bit
- R_1 field designates even/odd general register pair
  - Bit position of the leftmost 1 bit in general register R_2 placed in general register R_1 (even-numbered register)
    - Or 64 if no 1 bit found
  - Contents of general register R_2, with leftmost 1 bit set to zero, is placed in general register R_1 + 1 (odd-numbered register)
- Condition code indicates whether nonzero bit found
  - CC0 – All bits in R_2 are zero
  - CC2 – Found leftmost 1 bit

SHARE 115 21

FIND LEFTMOST ONE (FLOGR) is an extremely powerful instruction that can be used for a variety of bit-manipulation operations.

The instruction identifies the numeric bit position of the first 1 bit in the second-operand register, placing the result in the even-numbered register of the first operand. The second operand, with the leftmost one bit set to zero, is placed in the odd-numbered register of the first operand.

The *z/Architecture Principles of Operation* (SA22-7832) contains an example of the use of FLOGR in Appendix A.



Extended-Immediate Facility (9)
Example: FIND LEFTMOST ONE

- See example in Appendix A of the PoO

Without Extended Immediate	With Extended Immediate

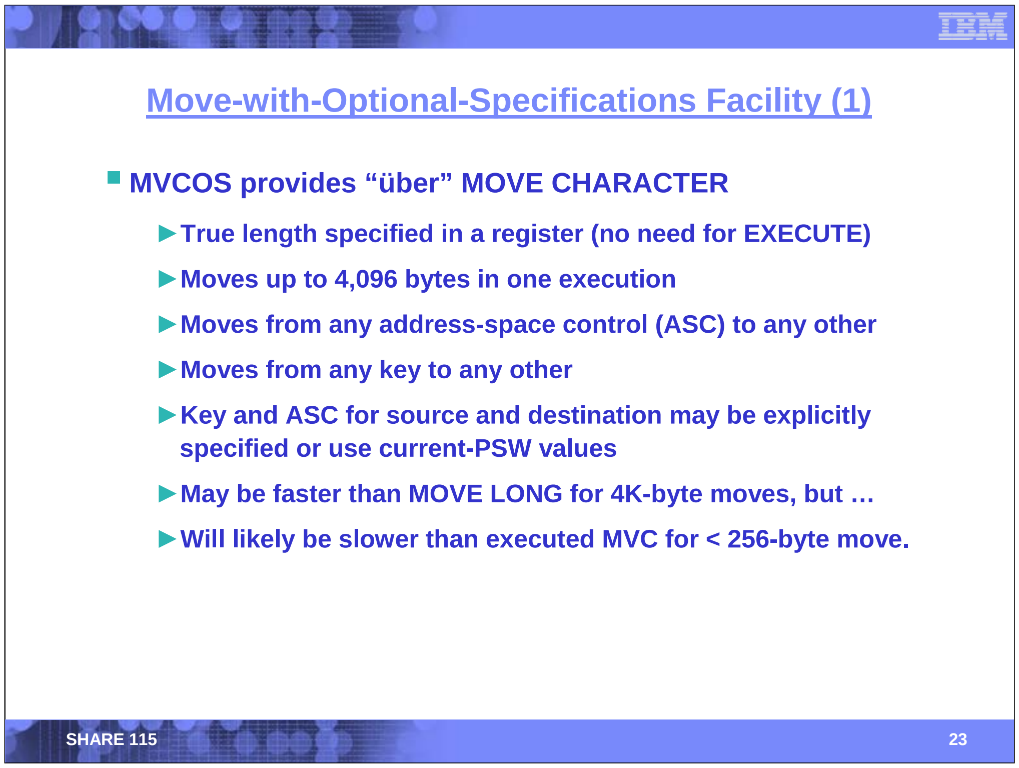
`* Input in R4`
`      LLIHH R0,X’8000’`
`      LGHI  R3,64`
`LOOP  LGR   R1,R0`
`      NGR   R1,R4`
`      JNZ   DONE`
`      SRLG  R0,R0,1`
`      JCT   R3,LOOP`
`DONE  IPM   R0`
`      LGHI  R2,64`
`      SGR   R2,R3`
`      LGR   R3,R4`
`      XGR   R3,R1`
`      NILH  R0,X’3000’`
`      SLL   R0,1`
`      SPM   R0`
	`FLOGR R2,R4`

Yeah, right!
Extra registers,
lots more instructions,
looped code, CC setting,
Aarrrrgh!

This left of this slide shows a sequence of instructions that would otherwise be required to implement the FIND LEFTMOST ONE operation shown on the right.

To perform the FLOGR function, this code on the left may need to loop up to 64 times. The final portion of the sequence is needed to set the condition code compatibly with that of FIND LEFTMOST ONE. Additionally, this sequence alters extra registers (0 and 3) that the FLOGR instruction do not.

On a System z9, FLOGR performs all these operations in 3 machine cycles!



Move-with-Optional-Specifications Facility (1)

- MVCOS provides “über” MOVE CHARACTER
  - True length specified in a register (no need for EXECUTE)
  - Moves up to 4,096 bytes in one execution
  - Moves from any address-space control (ASC) to any other
  - Moves from any key to any other
  - Key and ASC for source and destination may be explicitly specified or use current-PSW values
  - May be faster than MOVE LONG for 4K-byte moves, but …
  - Will likely be slower than executed MVC for < 256-byte move.

SHARE 115 23

MOVE WITH OPTIONAL SPECIFICATIONS provides functions similar to that of MOVE TO PRIMARY, MOVE TO SECONDARY, MOVE WITH DESTINATION KEY, MOVE WITH KEY, and MOVE WITH SOURCE KEY – and much more.

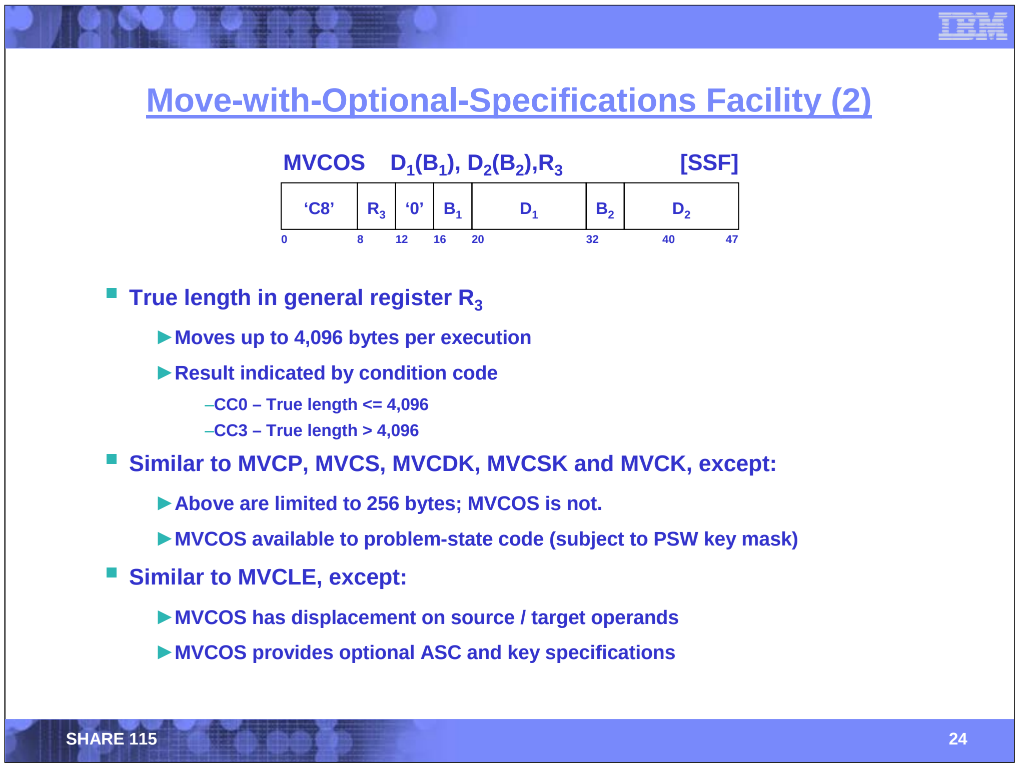
The source and target operands are both base/displacement storage operands (sorry, short displacement only). General register 1 contains a true length value; up to 4,096 bytes may be moved for each execution.

General register 0 contains operand-access controls (OAC) for both the source and target operands. The OAC contains specifications as to the storage key and address-space control (ASC) to be used for accessing each operand. The storage key may either be that of the current PSW or that specified in the OAC. Similarly, the address space control may be either that of the current PSW or that specified in the OAC. When GR0 contains zero, the source and target key/ASC values come from the PSW.

When executed in the problem state, the key specifications for both operands must be valid in the PSW key mask (CR3.32-47).

The instruction does not perform operand-overlap checking (unlike MVCL). However, because MVCOS performs several other authorization checks, its performance may be less than MVCL.

The instruction can specify any address-space control, however specification of AR-mode ASC is not particularly useful. If you already have access-register capability, then any other move instruction (e.g., MVC) can be used.



Move-with-Optional-Specifications Facility (2)

MVCOS $D_1(B_1)$, $D_2(B_2)$,R_3 [SSF]

'C8'	R_3	'0'	B_1	D_1	B_2	D_2
0	8	12	16	20	32	40 … 47

- True length in general register R_3
  - ▶Moves up to 4,096 bytes per execution
  - ▶Result indicated by condition code
    - –CC0 – True length <= 4,096
    - –CC3 – True length > 4,096
- Similar to MVCP, MVCS, MVCDK, MVCSK and MVCK, except:
  - ▶Above are limited to 256 bytes; MVCOS is not.
  - ▶MVCOS available to problem-state code (subject to PSW key mask)
- Similar to MVCLE, except:
  - ▶MVCOS has displacement on source / target operands
  - ▶MVCOS provides optional ASC and key specifications

SHARE 115 24

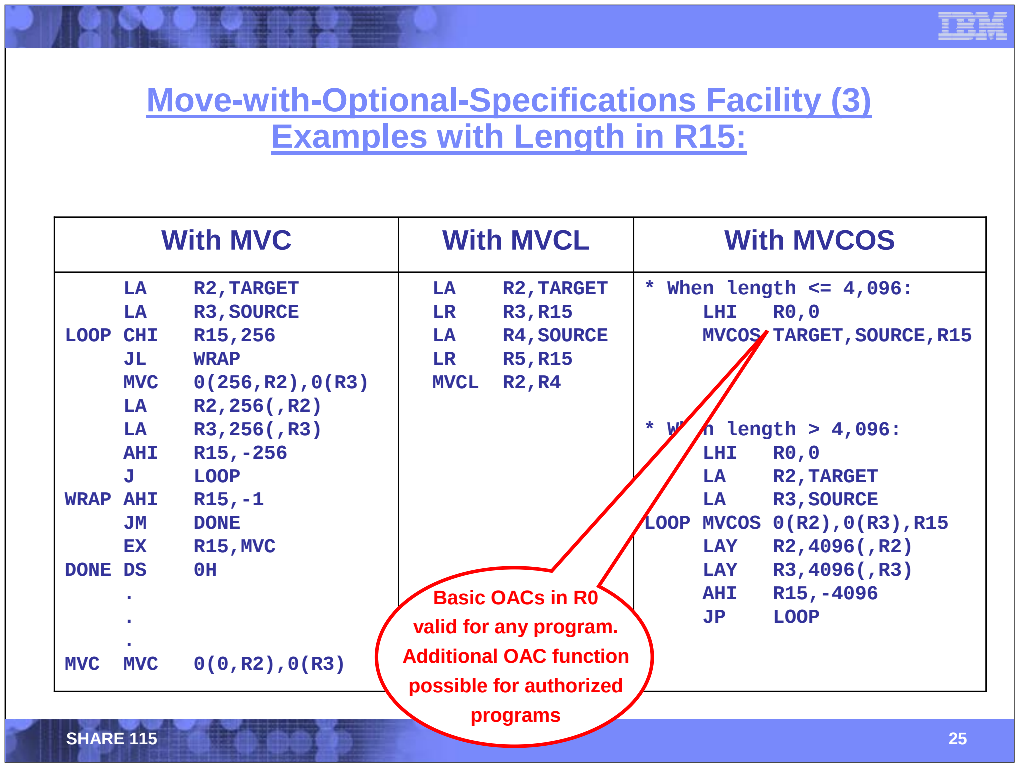
MOVE WITH OPTIONAL SPECIFICATIONS provides functions similar to that of MOVE TO PRIMARY, MOVE TO SECONDARY, MOVE WITH DESTINATION KEY, MOVE WITH KEY, and MOVE WITH SOURCE KEY – and much more.

The source and target operands are both base/displacement storage operands (sorry, short displacement only). General register R_3 contains a true length value; up to 4,096 bytes may be moved for each execution. The condition code indicates whether the true length was greater than 4,096.

The instruction does not perform operand-overlap checking (unlike MVCL). However, because MVCOS performs several other authorization checks, its performance may be less than MVCL.

One of the advantages of MVCOS (as opposed to MVCL or MVCLE) is that MVCOS provides a displacement field for both the source and target operands. Another advantage is the specification of a true length (thus avoiding all that messy executed-MVC hassle for a variable-length operand).

The instruction can specify any address-space control, however specification of AR-mode ASC is not particularly useful: If you already have access-register capability, then any other move instruction (e.g., MVC) can be used.



Move-with-Optional-Specifications Facility (3) Examples with Length in R15:

With MVC

```
      LA    R2,TARGET
      LA    R3,SOURCE
LOOP  CHI   R15,256
      JL    WRAP
      MVC   0(256,R2),0(R3)
      LA    R2,256(,R2)
      LA    R3,256(,R3)
      AHI   R15,-256
      J     LOOP
WRAP  AHI   R15,-1
      JM    DONE
      EX    R15,MVC
DONE  DS    0H
      .
      .
      .
MVC   MVC   0(0,R2),0(R3)
```

With MVCL

```
LA    R2,TARGET
LR    R3,R15
LA    R4,SOURCE
LR    R5,R15
MVCL  R2,R4
```

With MVCOS

```
* When length <= 4,096:
     LHI   R0,0
     MVCOS TARGET,SOURCE,R15


* When length > 4,096:
     LHI   R0,0
     LA    R2,TARGET
     LA    R3,SOURCE
LOOP MVCOS 0(R2),0(R3),R15
     LAY   R2,4096(,R2)
     LAY   R3,4096(,R3)
     AHI   R15,-4096
     JP    LOOP
```

Basic OACs in R0 valid for any program. Additional OAC function possible for authorized programs

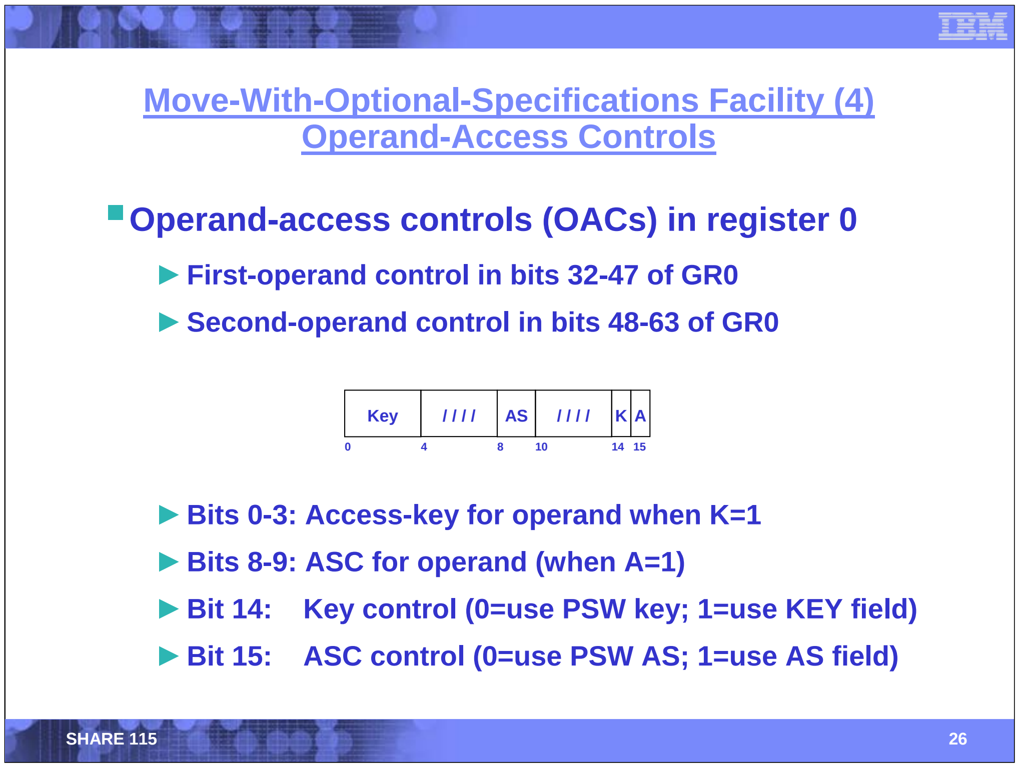
SHARE 115 25

This slide shows a comparison of techniques for moving a variable number of bytes.

The technique on the left uses an executed MOVE (MVC) instruction. If the length to be moved is known to be less than 256 bytes, a single EXECUTE will suffice. However, if the length to be moved exceeds 256 bytes, then the looping technique shown is necessary.

The technique in the center uses a MOVE LONG instruction. Although MVCL allows the specification of a true length, there are several disadvantages: four registers are required for the source and target addresses and lengths. Furthermore, unlike MVC (and MVCOS), the MVCL instruction has no provision for specifying an operand displacement.

The techniques shown on the right shows a simple use of MVCOS in the problem state – using the current key and address-space control in the PSW. If the length is known to be less than 4K, then a single MVCOS is required. If the length is larger than 4K, then the operation needs to be incorporated into a loop, as shown.



Move-With-Optional-Specifications Facility (4)
Operand-Access Controls

- Operand-access controls (OACs) in register 0
  - First-operand control in bits 32-47 of GR0
  - Second-operand control in bits 48-63 of GR0

Key	////	AS	////	K	A
0	4	8	10	14	15

  - Bits 0-3: Access-key for operand when K=1
  - Bits 8-9: ASC for operand (when A=1)
  - Bit 14: Key control (0=use PSW key; 1=use KEY field)
  - Bit 15: ASC control (0=use PSW AS; 1=use AS field)

SHARE 115 26

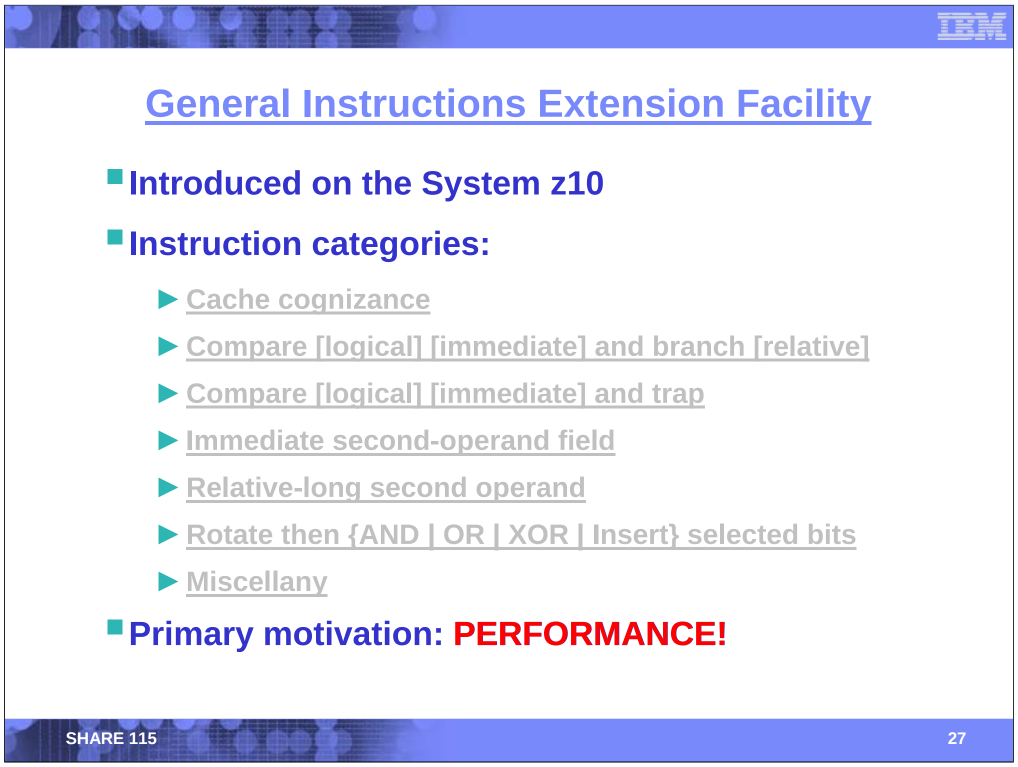
This slide shows the MVCOS implied operand in general register 0.

General register 0 contains operand-access controls (OACs) for both the first (target) and second (source) operands. For each operand, the OAC contains specifications as to the storage key and address-space control (ASC) to be used for accessing the operand. Bits 32-47 of general register 0 are the OAC for the first operand, and bits 48-63 of the register are the OAC for the second operand.

Bit 14 of each OAC is the key control (K) for the operand. When the K bit is zero, the access key in bits 8-11 of the PSW is used. When the K bit is one, bits 0-3 of the OAC are the access key. Similarly, bit 15 of each OAC is the address-space control (A) for the operand. When the K bit is zero, the address-space control in bits 16-17 of the PSW is used. When the K bit is one, bits 8-9 of the OAC are the address-space control.

Obviously, this instruction is extremely powerful, and accordingly, is subject to certain restrictions when executed in the problem state. When executed in the problem state, the key specifications for both operands must be valid in the PSW key mask (CR3.32-47). Other restrictions also apply. However, if a problem-state program simply wishes to use this instruction to simply move using the PSW key and current address space, setting GR0 to zeros works fine.

General register 0 contains operand-access controls (OAC) for both the source and target operands. The storage key may either be that of the current PSW or that specified in the OAC. Similarly, the address space control may be either that of the current PSW or that specified in the OAC. When GR0 contains zero, the source and target key/ASC values come from the PSW.



General Instructions Extension Facility

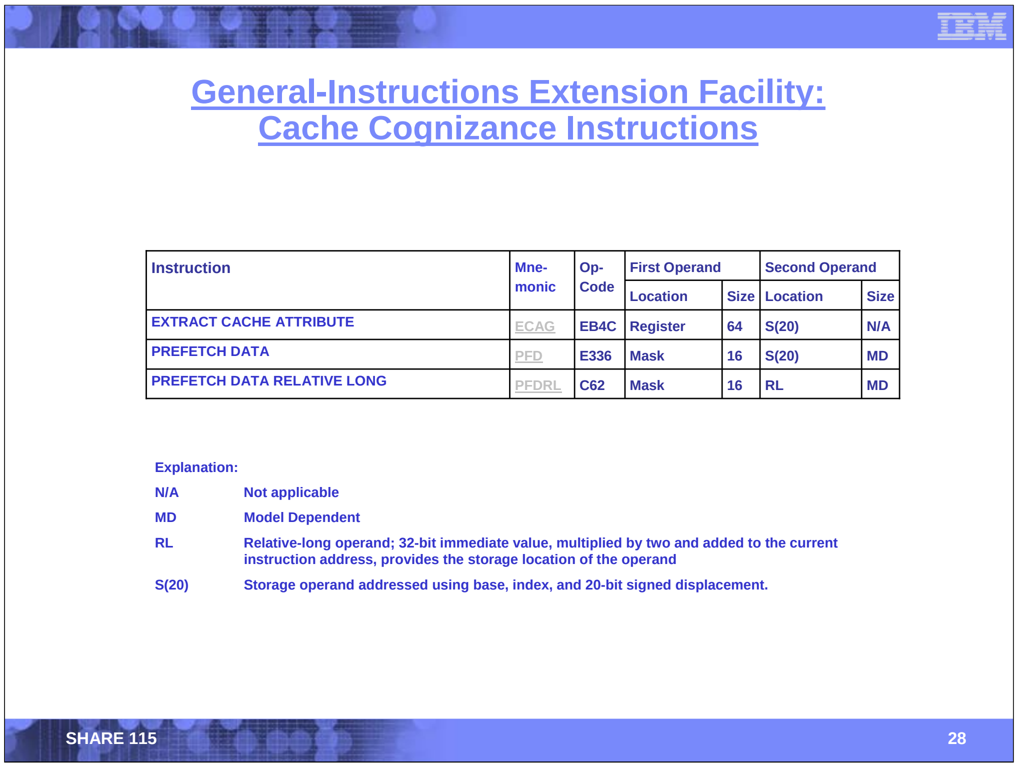
- Introduced on the System z10
- Instruction categories:
  - Cache cognizance
  - Compare [logical] [immediate] and branch [relative]
  - Compare [logical] [immediate] and trap
  - Immediate second-operand field
  - Relative-long second operand
  - Rotate then {AND | OR | XOR | Insert} selected bits
  - Miscellany
- Primary motivation: PERFORMANCE!

SHARE 115 27

The general-instructions-extension facility (GIEF) was primarily developed in response to requirements from IBM’s compiler-development organization in Toronto. As noted on this slide, performance was the driving factor in implementing (most of) these instructions.

The 72 instructions of the GIEF are divided into several categories, based on the instructions’ characteristics. In some cases, an instruction’s category characteristics overlap – for example, PREFETCH DATA RELATIVE LONG, which is described in the cache-cognizance category – also has the relative-long-second-operand characteristic.

The following slides will illustrate a small sampling of the GIEF instructions.



General-Instructions Extension Facility: Cache Cognizance Instructions

Instruction	Mne-monic	Op-Code	First Operand		Second Operand	
			Location	Size	Location	Size
EXTRACT CACHE ATTRIBUTE	ECAG	EB4C	Register	64	S(20)	N/A
PREFETCH DATA	PFD	E336	Mask	16	S(20)	MD
PREFETCH DATA RELATIVE LONG	PFDRL	C62	Mask	16	RL	MD

Explanation:

N/A	Not applicable
MD	Model Dependent
RL	Relative-long operand; 32-bit immediate value, multiplied by two and added to the current instruction address, provides the storage location of the operand
S(20)	Storage operand addressed using base, index, and 20-bit signed displacement.

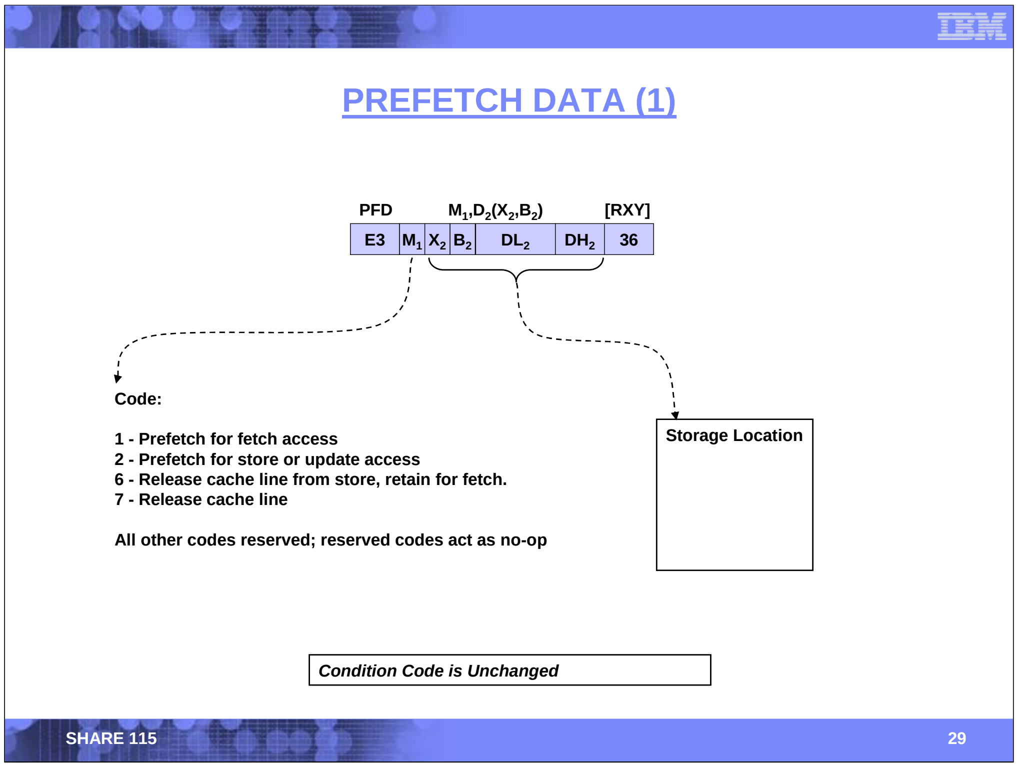
SHARE 115 28

Three instructions fall into the cache-cognizance category:

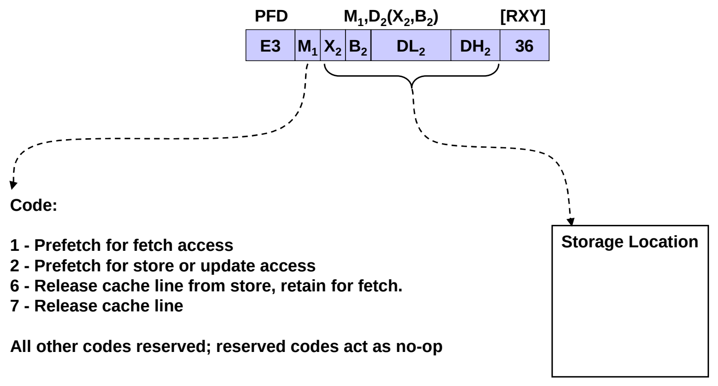
EXTRACT CACHE ATTRIBUTE provides a means by which various characteristics of a CPU’s cache(s) may be determined.

PREFETCH DATA and PREFETCH DATA RELATIVE LONG provide the means by which a storage operand may be fetched into – or released from – a cache line.

We’ll take a look at PREFETCH DATA in the following slides.



PREFETCH DATA (1)



Code:

1 - Prefetch for fetch access
2 - Prefetch for store or update access
6 - Release cache line from store, retain for fetch.
7 - Release cache line

All other codes reserved; reserved codes act as no-op

Condition Code is Unchanged

SHARE 115 29

For PREFETCH DATA (PFD), the second-operand address designates a storage location. Depending on the code specified in the M_3 field (bits 8-11 of the instruction), one of the following actions may be requested for the storage location:

1 -The location is to be brought into a cache line for fetch-access only.

2 -The location is to be brought into a cache line for fetch or store access.

6 -The location, potentially already in a cache line, is to be degraded from fetch-and-store access to fetch-access only.

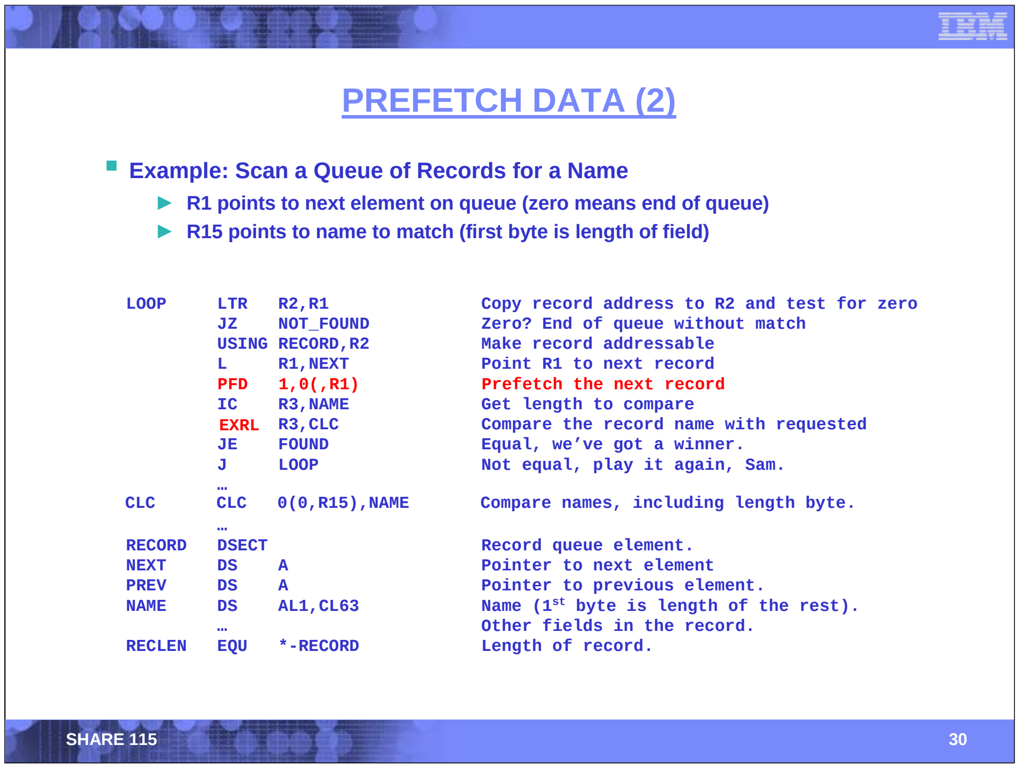
7 - The location, potentially already in a cache line, is to be removed from the cache.

The second operand may designate any storage location; no alignment, access, or PER-storage-alteration exceptions are recognized. The size of a cache line may be determined by the EXTRACT CACHE ATTRIBUTE instruction.

Codes 1 and 2 may be used in anticipation of accessing a storage location by subsequent instructions. Code 6 may be used when a storage location that was previously used for storing will subsequently be used only for fetching. Code 7 may be used when the storage location is not anticipated to be used in the near future.

All of these codes are merely hints to the CPU as to the anticipated use of the second-operand location; the CPU may not implement all codes, in which case the instruction acts as a no-op.

Note: significant performance degradation may occur if an inappropriate code is used, for example, code 7 is specified, but the storage location is immediately referenced.



PREFETCH DATA (2)

- Example: Scan a Queue of Records for a Name
  - R1 points to next element on queue (zero means end of queue)
  - R15 points to name to match (first byte is length of field)

```
LOOP     LTR   R2,R1               Copy record address to R2 and test for zero
         JZ    NOT_FOUND           Zero? End of queue without match
         USING RECORD,R2           Make record addressable
         L     R1,NEXT             Point R1 to next record
         PFD   1,0(,R1)            Prefetch the next record
         IC    R3,NAME             Get length to compare
         EXRL  R3,CLC              Compare the record name with requested
         JE    FOUND               Equal, we've got a winner.
         J     LOOP                Not equal, play it again, Sam.
         …
CLC      CLC   0(0,R15),NAME       Compare names, including length byte.
         …
RECORD   DSECT                     Record queue element.
NEXT     DS    A                   Pointer to next element
PREV     DS    A                   Pointer to previous element.
NAME     DS    AL1,CL63            Name (1st byte is length of the rest).
         …                         Other fields in the record.
RECLEN   EQU   *-RECORD            Length of record.
```

SHARE 115 30

This slide shows a sample use of the PREFETCH DATA (PFD) instruction. This example shows the traversal of a linked list of elements which, among other things, includes a name field to be examined.

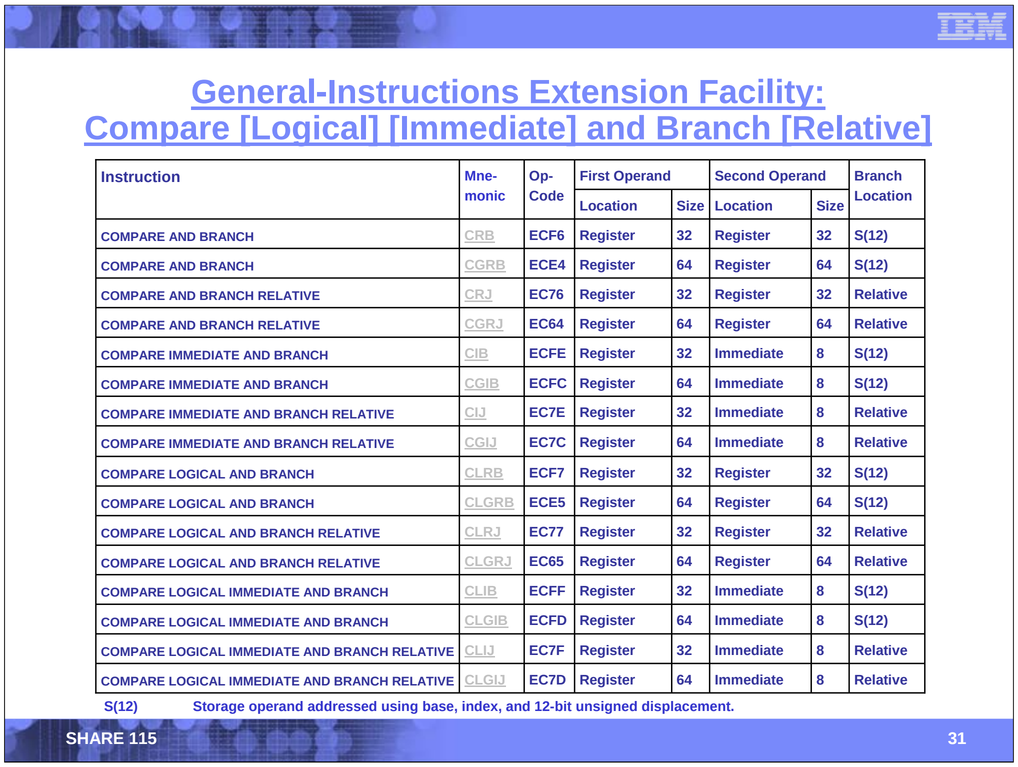
Let's assume that general register 1 contains the starting address of the queue element, the format of which is described in the RECORD DSECT. Let's also assume that general register 15 contains the address of a variable-length name field that we're searching for. For convenience, we'll also assume that the first byte of the name field contains the length of the rest of the valid data in the field.

Note: Embedding the length at the beginning of the field is a handy technique for comparison purposes. If, for example, the name to be compared is "John Smith", the length of the name is 10. Thus, the name field contains 0AD196989540E29489A388 hex. The executed CLC compares both the length byte and the name "John Smith", so that there is no accidental match against "John Smithson". Having the length byte exclude itself means that there is no need to decrement the length prior to performing the EX instruction.

First, a plug for a separate z10 facility: the execute-extension facility. If we replace the EX instruction with an EXECUTE RELATIVE LONG (EXRL, as shown in red), then the CLC instruction can be anywhere within 2 G-bytes of the EXRL … either before or after the EXRL.

In the sample program, note that the pointer to the next element of the queue is loaded before doing the name comparison. This allows us to slip in a PREFETCH DATA instruction immediately thereafter, allowing the next record of the queue to be fetched into the cache while the comparison operation is being executed. Depending on its use, PFD can provide significant performance benefit for applications.

Note: When comparing the name field in the record with that of the search value (addressed by GR15), this code fragment uses the length in the current queue element. This simple example does not address any access-exception conditions that might occur if the name length in the record exceeds that of the searched-for name (addressed by GR15). A more robust technique may be warranted in a real app.



General-Instructions Extension Facility: Compare [Logical] [Immediate] and Branch [Relative]

Instruction	Mne-monic	Op-Code	First Operand		Second Operand		Branch Location
			Location	Size	Location	Size	
COMPARE AND BRANCH	CRB	ECF6	Register	32	Register	32	S(12)
COMPARE AND BRANCH	CGRB	ECE4	Register	64	Register	64	S(12)
COMPARE AND BRANCH RELATIVE	CRJ	EC76	Register	32	Register	32	Relative
COMPARE AND BRANCH RELATIVE	CGRJ	EC64	Register	64	Register	64	Relative
COMPARE IMMEDIATE AND BRANCH	CIB	ECFE	Register	32	Immediate	8	S(12)
COMPARE IMMEDIATE AND BRANCH	CGIB	ECFC	Register	64	Immediate	8	S(12)
COMPARE IMMEDIATE AND BRANCH RELATIVE	CIJ	EC7E	Register	32	Immediate	8	Relative
COMPARE IMMEDIATE AND BRANCH RELATIVE	CGIJ	EC7C	Register	64	Immediate	8	Relative
COMPARE LOGICAL AND BRANCH	CLRB	ECF7	Register	32	Register	32	S(12)
COMPARE LOGICAL AND BRANCH	CLGRB	ECE5	Register	64	Register	64	S(12)
COMPARE LOGICAL AND BRANCH RELATIVE	CLRJ	EC77	Register	32	Register	32	Relative
COMPARE LOGICAL AND BRANCH RELATIVE	CLGRJ	EC65	Register	64	Register	64	Relative
COMPARE LOGICAL IMMEDIATE AND BRANCH	CLIB	ECFF	Register	32	Immediate	8	S(12)
COMPARE LOGICAL IMMEDIATE AND BRANCH	CLGIB	ECFD	Register	64	Immediate	8	S(12)
COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE	CLIJ	EC7F	Register	32	Immediate	8	Relative
COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE	CLGIJ	EC7D	Register	64	Immediate	8	Relative

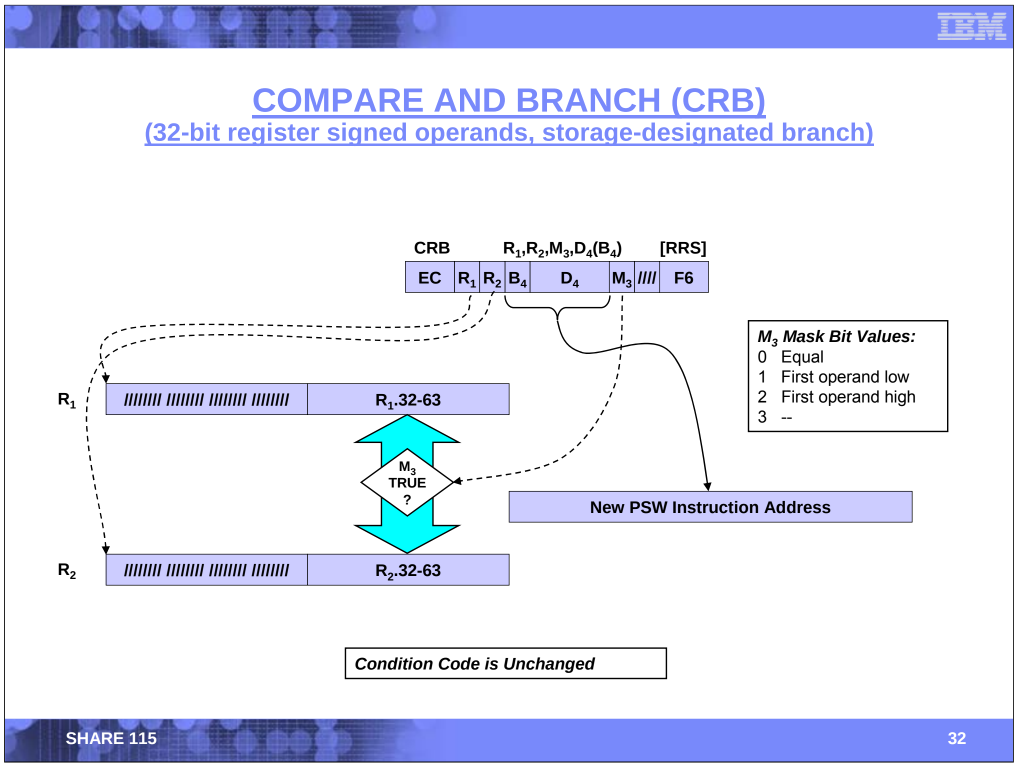
S(12) Storage operand addressed using base, index, and 12-bit unsigned displacement.

SHARE 115 31

The COMPARE AND BRANCH instructions combine a compare operation and, if the specified condition is met, a branch operation, in a single instruction. When the specified condition is not met, execution continues with the next sequential instruction. The bracketed terms in this slide's title illustrate the many forms of the instruction, providing the following characteristics:

- Numeric attribute: signed versus unsigned (top half of the table versus bottom half)
- Second-operand location: register versus immediate field (every four rows)
- Branch designation: base and 12-bit displacement versus 16-bit signed relative (every two rows)
- First operand size: 32-bit versus 64-bit (every other row)

The instructions have a rich set of operands. For the instruction formats with the comparand (second operand) in an immediate field, there is only room for an 8-bit value. We will explore a few of these instructions in the following slides.



COMPARE AND BRANCH (CRB)
(32-bit register signed operands, storage-designated branch)

M_3 Mask Bit Values:

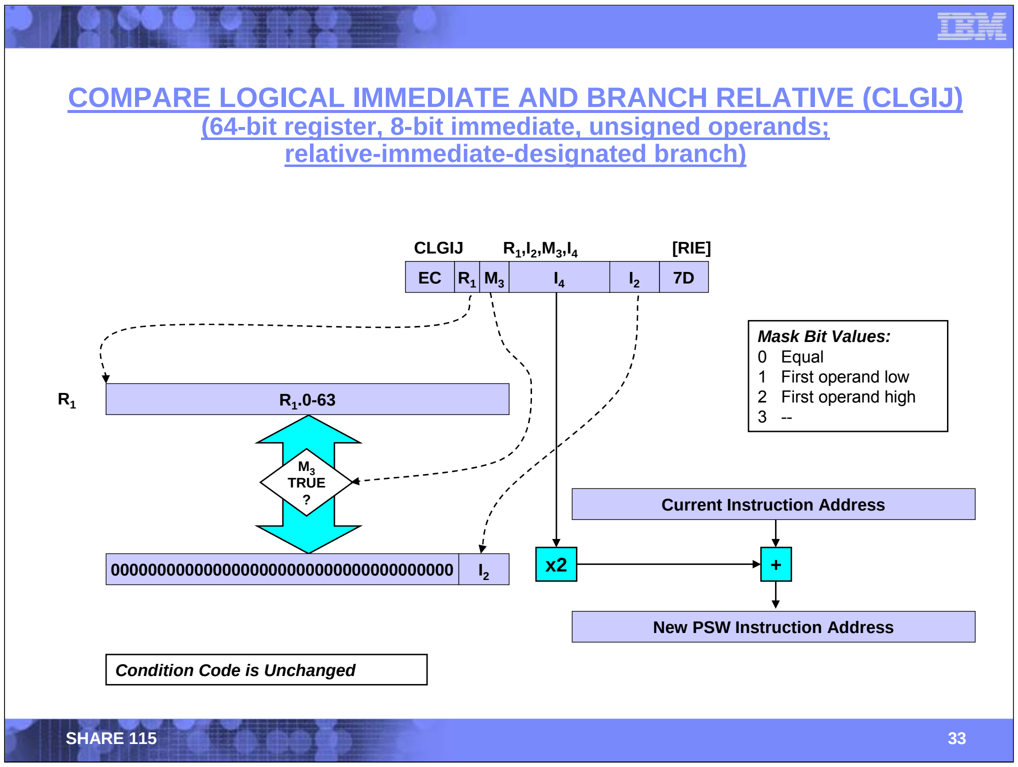
0	Equal
1	First operand low
2	First operand high
3	--

Condition Code is Unchanged

COMPARE AND BRANCH (CRB) compares the 32-bit signed binary integer in bits 32-63 of the first-operand register with a 32-bit signed binary integer in the corresponding bits in the second-operand register.

If the conditions specified by the mask field (the third operand) are true, then control branches to the location specified by the fourth operand. The fourth operand is a storage location designated by a base register and 12-bit unsigned displacement.

If the conditions specified by the mask field are not true, then execution continues with the next sequential instruction.



COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (CLGIJ)
(64-bit register, 8-bit immediate, unsigned operands; relative-immediate-designated branch)

CLGIJ R_1,I_2,M_3,I_4 [RIE]
EC
R_1
M_3
I_4
I_2
7D
R_1
R_1.0-63
M_3 TRUE ?
00000000000000000000000000000000000000 I_2
x2
Current Instruction Address
+
New PSW Instruction Address

Mask Bit Values:

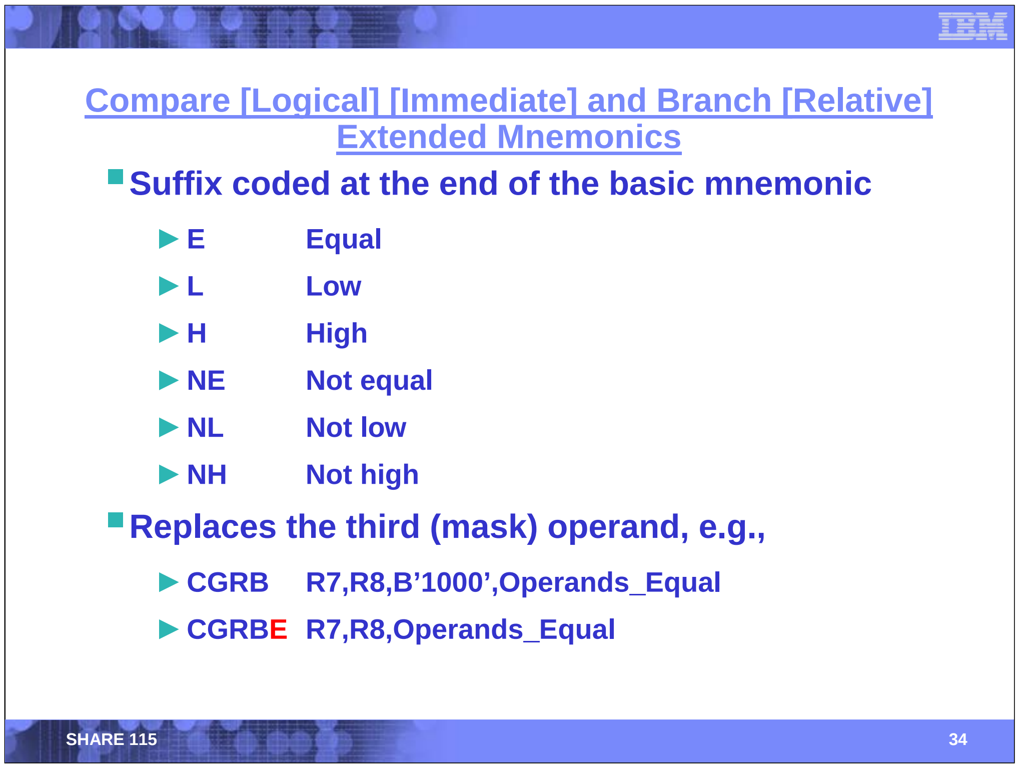
0	Equal
1	First operand low
2	First operand high
3	--

Condition Code is Unchanged

COMPARE LOGICAL IMMEDIATE AND BRANCH RELATIVE (CLGIJ) compares the 64-bit unsigned binary integer in the first-operand register with an 8-bit unsigned binary integer in the I_2 field of the instruction, extended on the left with 56 binary zeros.

If the conditions specified by the mask field (the third operand) are true, then control branches to the location specified by the fourth operand. The fourth operand is relative to the current instruction address. The I_4 field contains a 16-bit signed binary integer which is multiplied by two and then added to the current instruction address (subject to addressing-mode constraints).

If the conditions specified by the mask field are not true, then execution continues with the next sequential instruction.



Compare [Logical] [Immediate] and Branch [Relative] Extended Mnemonics

- Suffix coded at the end of the basic mnemonic

▶ E	Equal
▶ L	Low
▶ H	High
▶ NE	Not equal
▶ NL	Not low
▶ NH	Not high

- Replaces the third (mask) operand, e.g.,

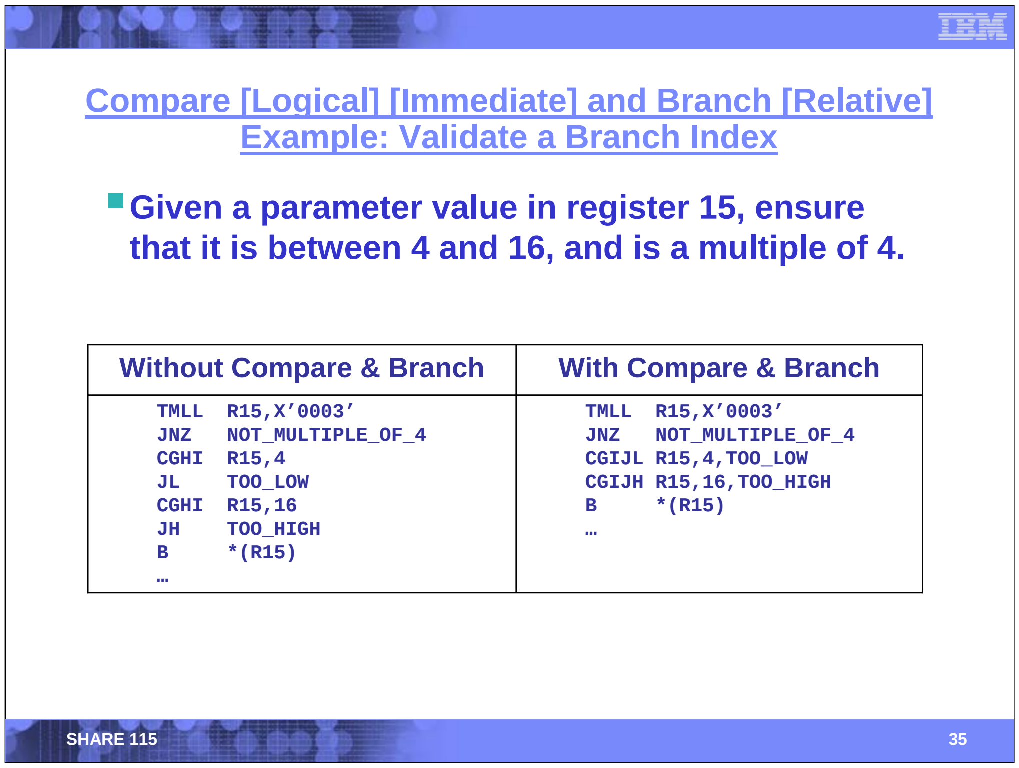
```
CGRB    R7,R8,B'1000',Operands_Equal
CGRBE   R7,R8,Operands_Equal
```

SHARE 115 34

For each of the COMPARE AND BRANCH (and COMPARE AND TRAP) instructions, the High Level Assembler implements extensions to the mnemonics in lieu of the M_3 field. The extensions include:

E	equal	M_3 = 1000 binary
H	high	M_3 = 0010 binary
L	low	M_3 = 0100 binary
NE	not equal	M_3 = 0110 binary
NH	not high	M_3 = 1100 binary
NL	not low	M_3 = 1010 binary

When the mnemonic extension is coded, the M_3 field must be omitted.



Compare [Logical] [Immediate] and Branch [Relative] Example: Validate a Branch Index

- Given a parameter value in register 15, ensure that it is between 4 and 16, and is a multiple of 4.

Without Compare & Branch	With Compare & Branch
`TMLL  R15,X’0003’`	`TMLL  R15,X’0003’`
`JNZ   NOT_MULTIPLE_OF_4`	`JNZ   NOT_MULTIPLE_OF_4`
`CGHI  R15,4`	`CGIJL R15,4,TOO_LOW`
`JL    TOO_LOW`	`CGIJH R15,16,TOO_HIGH`
`CGHI  R15,16`	`B     *(R15)`
`JH    TOO_HIGH`	`…`
`B     *(R15)`	
`…`	

SHARE 115 35

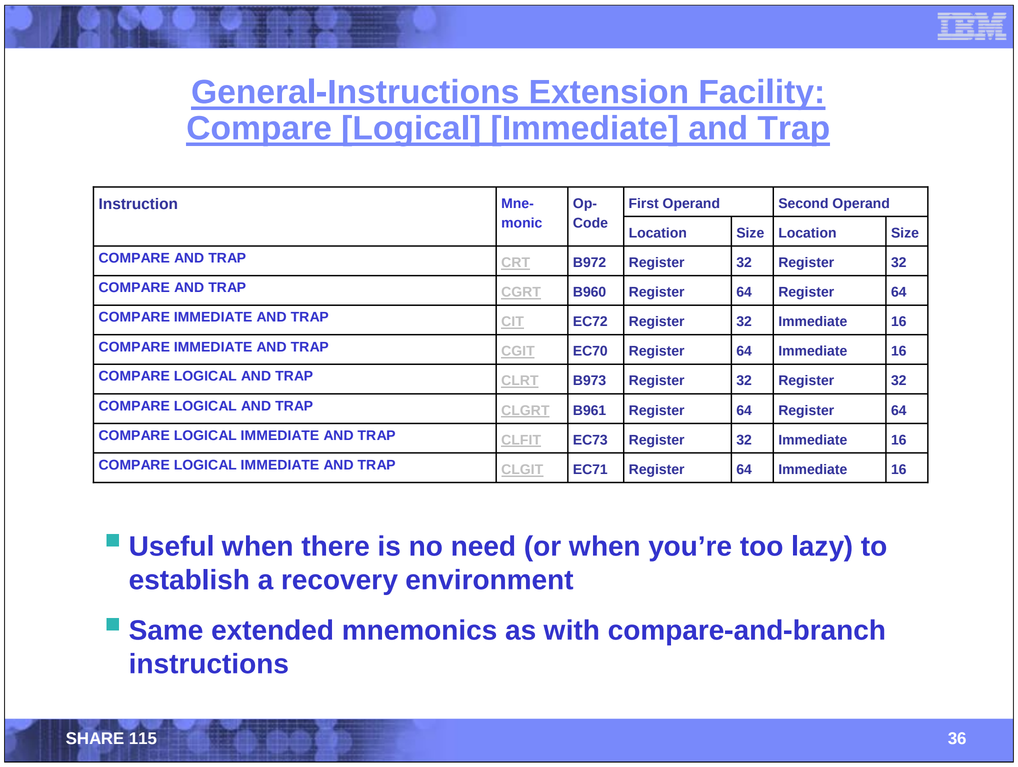
Here we see an example of ensuring a value in general register 15 can safely be used as an index into a table of four branch instructions. We need to do a few checks first:

1. Make sure that the value is a multiple of four
2. Make sure that the value is equal to or greater than 4 (a value of zero would cause this code sequence to loop forever).
3. Make sure that the value is less than or equal to 16.

The first test is performed by the TEST UNDER MASK (TMLL) instruction that examines bits 62 and 63 of general register 15. If either of these bits is one, the value cannot be a multiple of four, and the following branch invokes an error routine.

On the left side (without the compare-and-branch) instructions, a separate comparison instruction, followed by a branch instruction is needed to determine if the value is within the low and high bounds.

On the right side, the compare and branch instructions are combined into a single instruction, saving code space and machine cycles.



General-Instructions Extension Facility: Compare [Logical] [Immediate] and Trap

Instruction	Mne-monic	Op-Code	First Operand		Second Operand	
			Location	Size	Location	Size
COMPARE AND TRAP	CRT	B972	Register	32	Register	32
COMPARE AND TRAP	CGRT	B960	Register	64	Register	64
COMPARE IMMEDIATE AND TRAP	CIT	EC72	Register	32	Immediate	16
COMPARE IMMEDIATE AND TRAP	CGIT	EC70	Register	64	Immediate	16
COMPARE LOGICAL AND TRAP	CLRT	B973	Register	32	Register	32
COMPARE LOGICAL AND TRAP	CLGRT	B961	Register	64	Register	64
COMPARE LOGICAL IMMEDIATE AND TRAP	CLFIT	EC73	Register	32	Immediate	16
COMPARE LOGICAL IMMEDIATE AND TRAP	CLGIT	EC71	Register	64	Immediate	16

- **Useful when there is no need (or when you're too lazy) to establish a recovery environment**
- **Same extended mnemonics as with compare-and-branch instructions**

SHARE 115 36

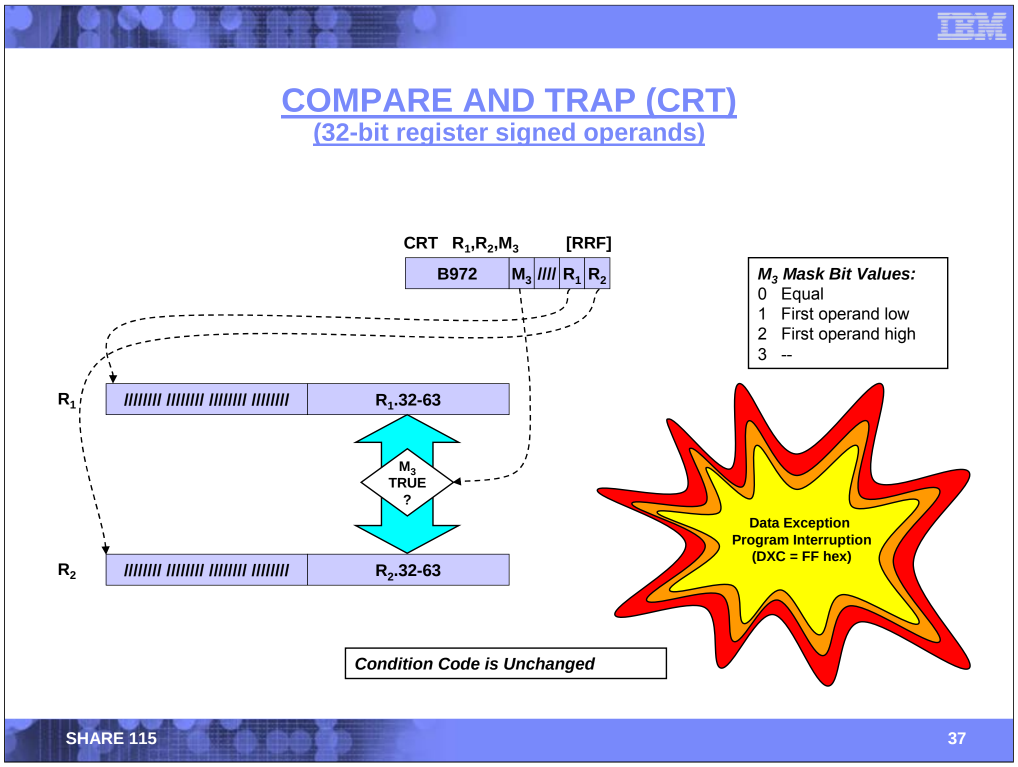
The COMPARE AND TRAP instructions combine a compare operation and, if the specified condition is met, a program interruption, in a single instruction. When the specified condition is met, a data-exception program interruption is generated, and a data-exception code (DXC) of FF hex is stored at real location 147. When the specified condition is not met, execution continues with the next sequential instruction. Many forms of the instruction are provided based on these characteristics:

- Numeric attribute: signed versus unsigned
- Operand size: 32-bit versus 64-bit
- Second-operand location: register versus immediate field

Because there is no branch location required (as in COMPARE AND BRANCH), the instruction formats with an immediate-field comparand provide a 16-bit value.

For each of the COMPARE AND TRAP instructions, the High Level Assembler implements extensions to the mnemonics in lieu of the M_3 field, as described in the notes for COMPARE AND BRANCH.

COMPARE AND TRAP is useful in a coding environment where a comparison is required (for example, checking for a null pointer), but the application is not immediately concerned with the recovery from such a comparison. Rather, if the comparison results in a true condition, the recovery is escalated to whatever recovery routine (if any) is provided.



COMPARE AND TRAP (CRT)
(32-bit register signed operands)

M_3 Mask Bit Values:

0	Equal
1	First operand low
2	First operand high
3	--

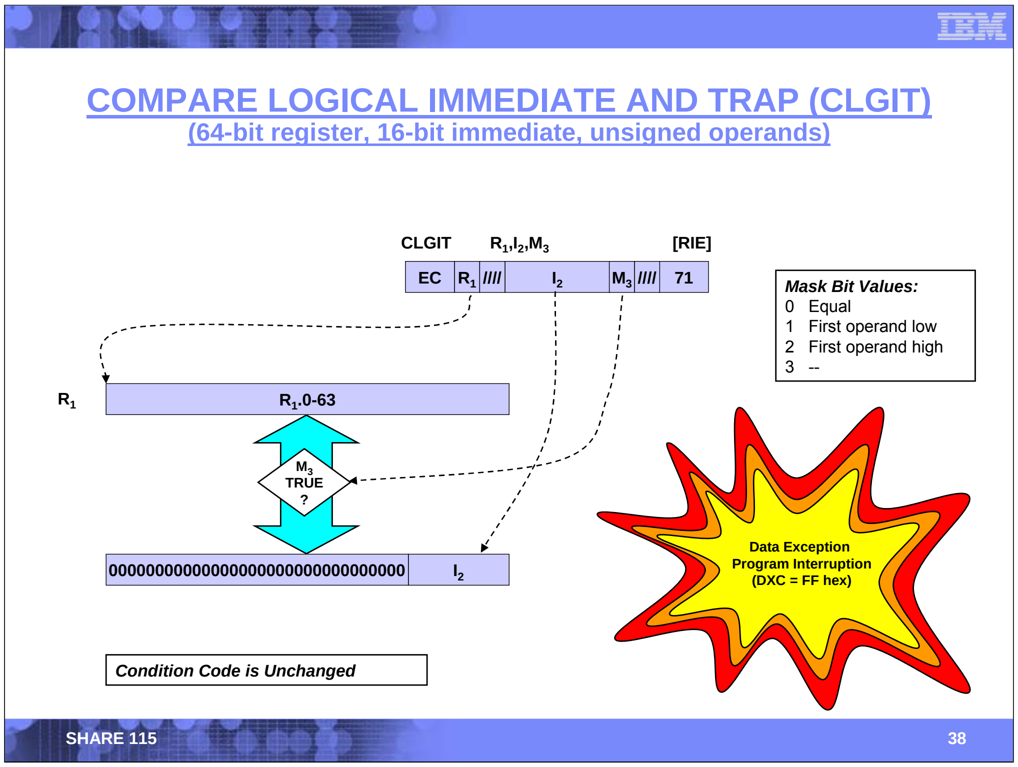
Condition Code is Unchanged

SHARE 115 37

COMPARE AND TRP (CRT) compares the 32-bit signed binary integer in bits 32-63 of the first-operand register with a 32-bit signed binary integer in the corresponding bits in the second-operand register.

If the conditions specified by the mask field (the third operand) are true, then a data exception program-interruption condition is recognized. The data-exception code (DXC) contains FF hex.

If the conditions specified by the mask field are not true, then execution continues with the next sequential instruction.



COMPARE LOGICAL IMMEDIATE AND TRAP (CLGIT)

(64-bit register, 16-bit immediate, unsigned operands)

Mask Bit Values:

0	Equal
1	First operand low
2	First operand high
3	--

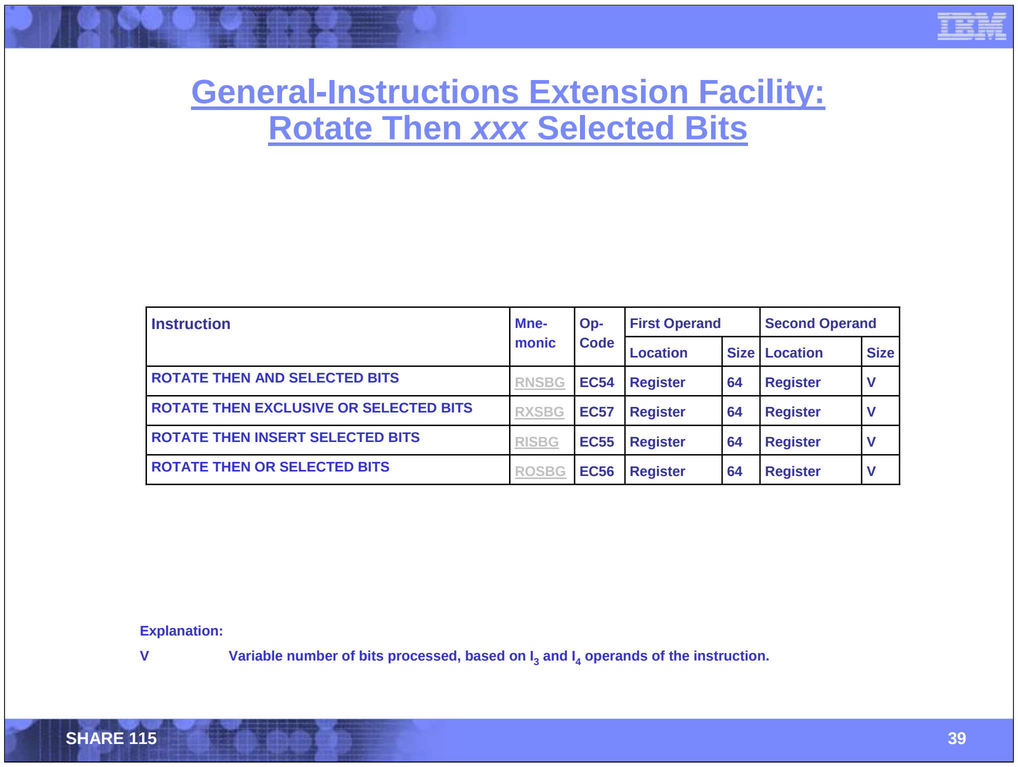
Condition Code is Unchanged

SHARE 115 38

COMPARE LOGICAL IMMEDIATE AND TRAP (CLGIT) compares the 64-bit unsigned binary integer in the first-operand register with a 16-bit unsigned binary integer in the I_2 field of the instruction, extended on the left with 48 binary zeros.

If the conditions specified by the mask field (the third operand) are true, then a data exception program-interruption condition is recognized. The data-exception code (DXC) contains FF hex.

If the conditions specified by the mask field are not true, then execution continues with the next sequential instruction.



General-Instructions Extension Facility: Rotate Then *xxx* Selected Bits

Instruction	Mne-monic	Op-Code	First Operand		Second Operand	
			Location	Size	Location	Size
ROTATE THEN AND SELECTED BITS	RNSBG	EC54	Register	64	Register	V
ROTATE THEN EXCLUSIVE OR SELECTED BITS	RXSBG	EC57	Register	64	Register	V
ROTATE THEN INSERT SELECTED BITS	RISBG	EC55	Register	64	Register	V
ROTATE THEN OR SELECTED BITS	ROSBG	EC56	Register	64	Register	V

Explanation:

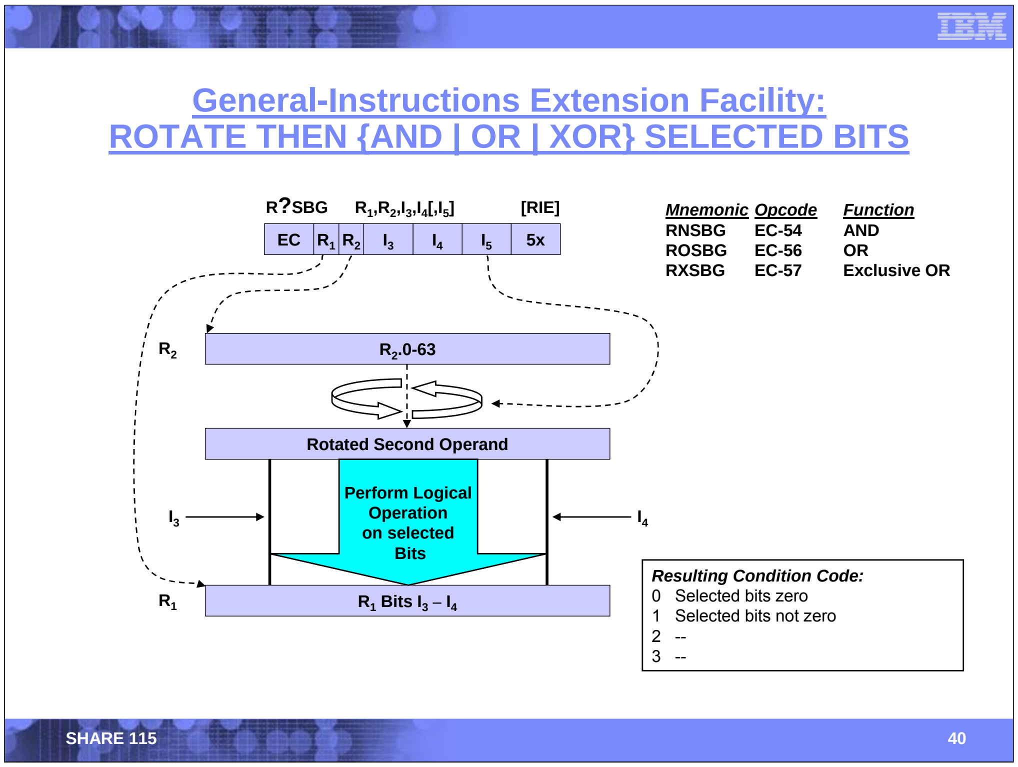
V	Variable number of bits processed, based on I_3 and I_4 operands of the instruction.

SHARE 115 39

Four instructions perform a rotate-left operation on the second-operand register; bits that rotate out of bit position zero reenter the register at bit position 63. Subsequently, depending on the instruction, one of four operations is performed using selected bits of the rotated value and the first-operand register.

It is the opinion of this author that these instructions, particularly the ROTATE THEN INSERT SELECTED BITS instruction, are some of the most powerful and useful operations in the architecture.

The following slides provide additional details.



General-Instructions Extension Facility: ROTATE THEN {AND | OR | XOR} SELECTED BITS

Mnemonic	*Opcode*	*Function*
RNSBG	EC-54	AND
ROSBG	EC-56	OR
RXSBG	EC-57	Exclusive OR

Resulting Condition Code:
0 Selected bits zero
1 Selected bits not zero
2 --
3 --

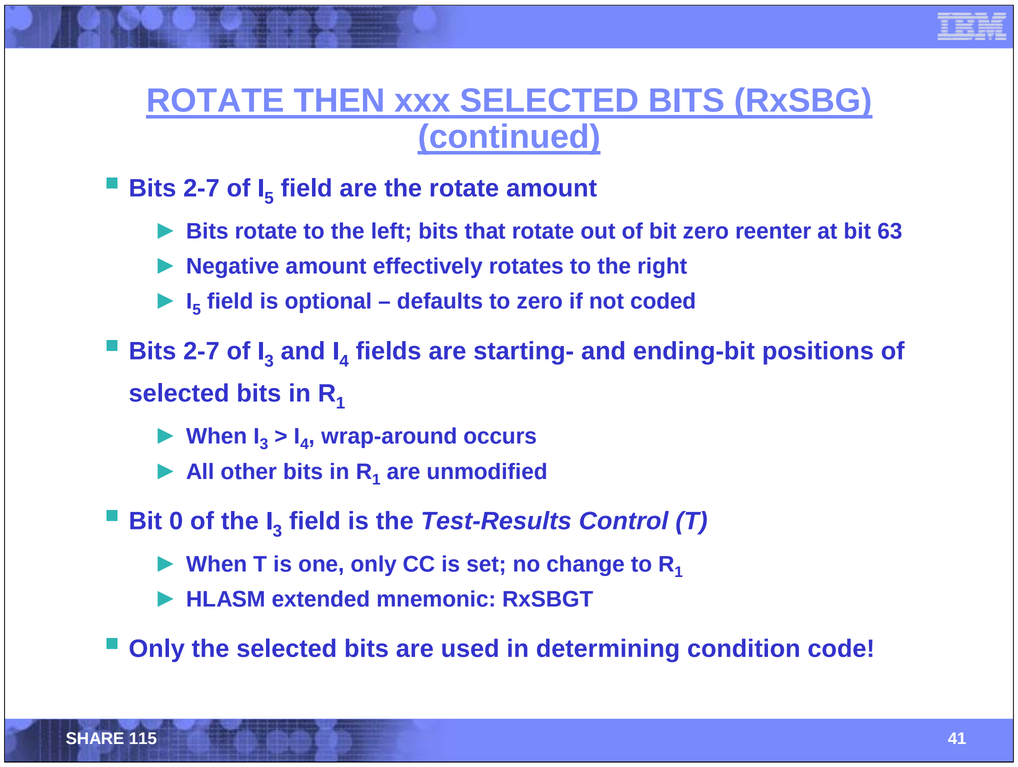
Three instructions, ROTATE THEN AND SELECTED BITS (RNSBG), ROTATE THEN OR SELECTED BITS (ROSBG), and ROTATE THEN EXCLUSIVE OR SELECTED BITS (RXSBG), rotate the value contained in the second-operand register by the number of bits specified in the I_5 field. However, the contents of the second-operand register remain unchanged.

Subsequently, a logical operation (AND, OR, or XOR, depending on the instruction) is performed using a selected range of the rotated value and the corresponding bits of the first-operand register. The range of bits is specified by the I_3 and I_4 fields.

Bit 0 of the I_3 field contains the test-results control (T). When the T bit is zero, the results of the logical operation replace the selected bits of the first-operand register, and the remaining (nonselected) bits remain unchanged. When the T bit is one, the entire first-operand register is unchanged.

Regardless of the setting of the T bit, the condition code indicates the results of the logical operation as performed on the selected bits only.

Note: HLASM treats the I_5 field as optional. If not specified, a rotate amount of zero is used.



ROTATE THEN xxx SELECTED BITS (RxSBG) (continued)

- Bits 2-7 of I_5 field are the rotate amount
  - Bits rotate to the left; bits that rotate out of bit zero reenter at bit 63
  - Negative amount effectively rotates to the right
  - I_5 field is optional – defaults to zero if not coded
- Bits 2-7 of I_3 and I_4 fields are starting- and ending-bit positions of selected bits in R_1
  - When $I_3 > I_4$, wrap-around occurs
  - All other bits in R_1 are unmodified
- Bit 0 of the I_3 field is the *Test-Results Control (T)*
  - When T is one, only CC is set; no change to R_1
  - HLASM extended mnemonic: RxSBGT
- Only the selected bits are used in determining condition code!

SHARE 115 41

Bits 2-7 of the I_3, I_4, and I_5 fields contain the starting bit position of the selected bits, then ending-bit position of the selected bits, and the rotate amount, respectively.

When the ending-bit position is less than the starting-bit position, the selected bits wrap around. For example, if the instruction is coded:

```
RNSBG  R1,R2,60,3,24
```

then the selected bits are bits 60, 61, 62, 63, 0, 1, 2, and 3.

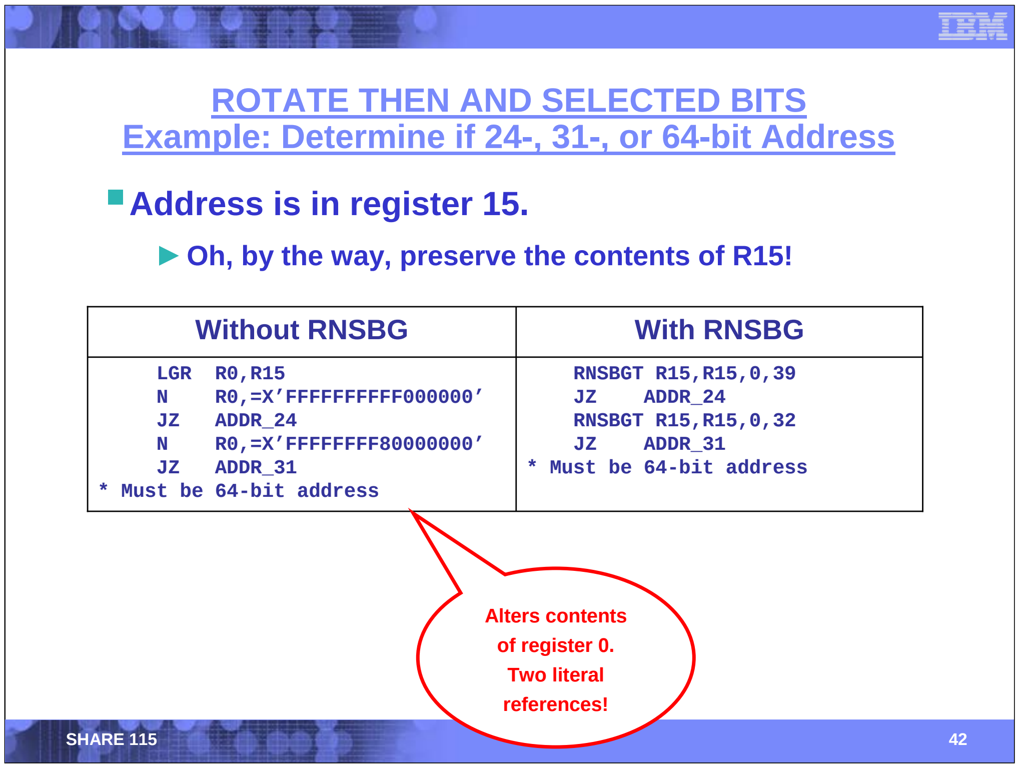
The test-results control (T) is contained in bit 0 of the I_3 field. The following statement illustrates the example shown above, but with the T control set:

```
RNSBG  R1,R2,128+60,3,24
```

The High Level Assembler provides a mnemonic extension, “T” which causes a value of X’80’ to be ORed into the I_3 field. Thus the following statement generates code equivalent to the preceding example:

```
RNSBGT  R1,R2,60,3,24
```

The High Level Assembler also treats the fifth operand (the rotation amount) as being optional. If the operand is not specified, no rotation is performed.



ROTATE THEN AND SELECTED BITS
Example: Determine if 24-, 31-, or 64-bit Address

- Address is in register 15.
  - Oh, by the way, preserve the contents of R15!

Without RNSBG	With RNSBG

`LGR R0,R15`
`N R0,=X'FFFFFFFFFF000000'`
`JZ ADDR_24`
`N R0,=X'FFFFFFFF80000000'`
`JZ ADDR_31`
`* Must be 64-bit address`
	`RNSBGT R15,R15,0,39`
	`JZ ADDR_24`
	`RNSBGT R15,R15,0,32`
	`JZ ADDR_31`
	`* Must be 64-bit address`

Alters contents of register 0. Two literal references!

SHARE 115

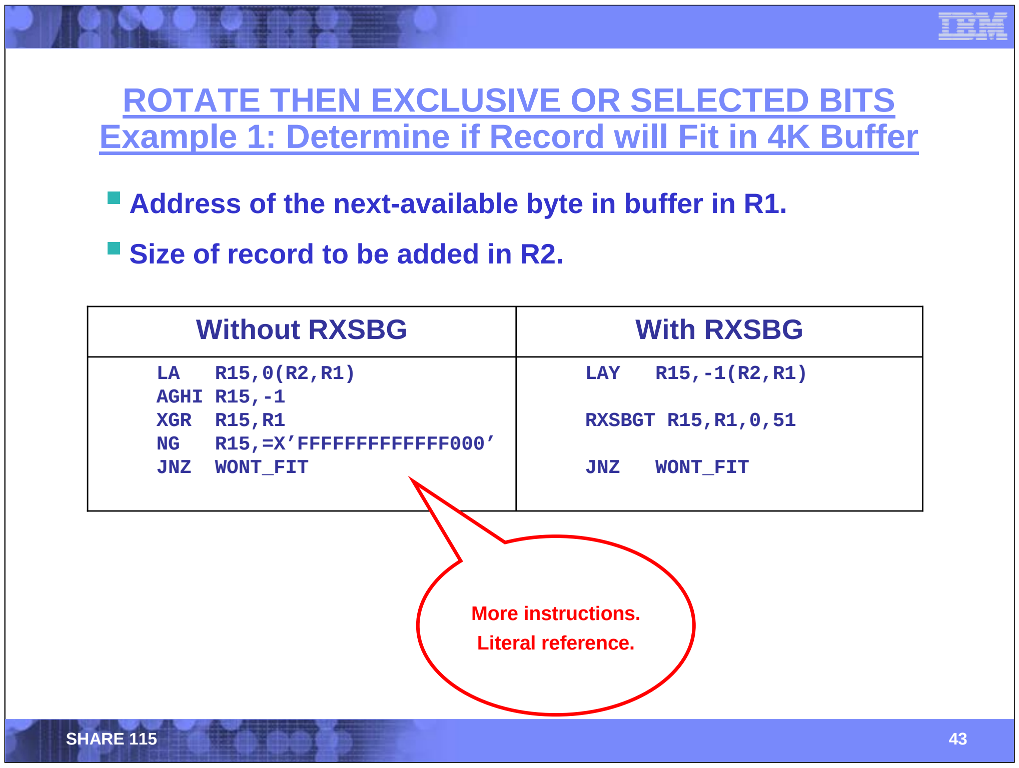
42

This slide shows how to determine if an address in general register 15 can be used in various addressing modes. If the address exceeds 16 megabytes, then it must be handled in either the 31- or 64-bit addressing modes. If the address exceeds 2 gigabytes, then it must be handled in the 64-bit addressing mode.

In the examples on the left, a copy of the value is first ANDed with a mask representing any bits that can must be zeros in the 24-bit addressing mode. If all of these bits are zero, then the address can safely be treated as a 24-bit address. Next, a copy of the value is ANDed with a mask representing bits that must be zeros in a 31-bit address. If any allo these bits are zero, then the address can be treated as a 31-bit address. If the code falls through, we can treat the value as being a 64-bit address.

Why a copy of the value? Because the AND instruction is destructive! The code is carefully arranged so that we only need to make one copy of GR15, but it still makes two storage references to two different literals.

The ROTATE THEN AND SELECTED BITS is using the test-results flag (the Z mnemonic suffix), thus the results are not actually written to the first operand. So, there are no additional storage references, and no need to copy the value into a separate register.



ROTATE THEN EXCLUSIVE OR SELECTED BITS
Example 1: Determine if Record will Fit in 4K Buffer

- Address of the next-available byte in buffer in R1.
- Size of record to be added in R2.

Without RXSBG	With RXSBG

`LA R15,0(R2,R1)`
`AGHI R15,-1`
`XGR R15,R1`
`NG R15,=X'FFFFFFFFFFFFF000'`
`JNZ WONT_FIT`
	`LAY R15,-1(R2,R1)`
	
	`RXSBGT R15,R1,0,51`
	
	`JNZ WONT_FIT`

More instructions.
Literal reference.

SHARE 115 43

Regardless of whether you use the code on the left or on the right, this technique is something that every programmer should have in their tool box. It is an efficient means of determining if an address crosses an integral (power of two) boundary. For example, when adding a record to a 4K-byte buffer, the program needs to determine whether it will fit.

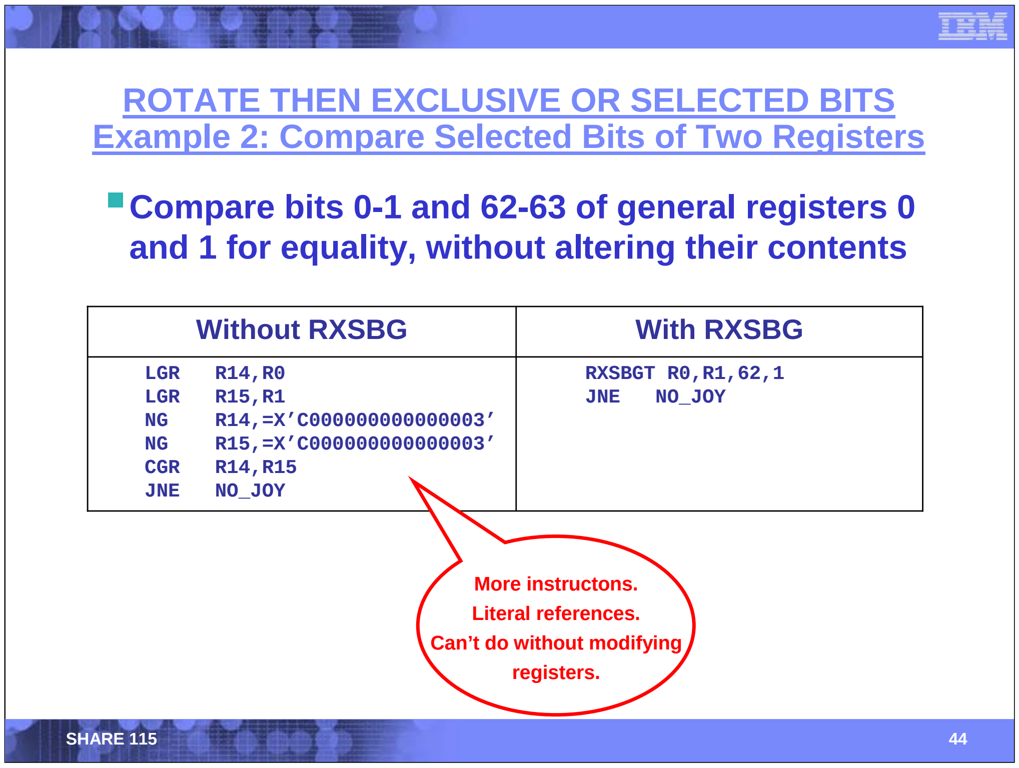
In our example, general register 1 contains the address of the next available byte in the buffer, and general register two contains the length of the record to be added. The boundary in this example is 4K, however with slight modifications (to the mask constant on the left, or to the bit range on the right), this example works for any power-of-two boundary.

Both code fragments begin by determining the address of the last byte that the new record would occupy if added; this address is placed into general register 15. The code on the left does not take advantage of the long-displacement facility, and needs two instructions to accomplish this (the LA and AGHI), whereas the code on the right uses a long-displacement LOAD ADDRESS that allows for negative displacement.

The magic of this determination comes from the exclusive-OR operation. By XORing the original address with the last byte of the new address, only the bits that differ will be set to one. The code on the left must subsequently AND off the rightmost 12 bits to isolate only the portion of the address identifying the 4K-byte block. The code on the right can accomplish this by using ROTATE THEN EXCLUSIVE OR SELECTED bits, but the XORing – and the setting of the condition code – is limited to the selected bits (bits 0-51).

In either case, if any of bits 0-51 of the XORed value are one, then the addition has overflowed into the next block, and the record won't fit.

This technique is easily adaptable to any integral boundary that is known at compile time. The technique can be adapted to a variable power-of-two boundary with only a few more instructions.



ROTATE THEN EXCLUSIVE OR SELECTED BITS
Example 2: Compare Selected Bits of Two Registers

- Compare bits 0-1 and 62-63 of general registers 0 and 1 for equality, without altering their contents

Without RXSBG	With RXSBG

`LGR R14,R0`
`LGR R15,R1`
`NG R14,=X’C000000000000003’`
`NG R15,=X’C000000000000003’`
`CGR R14,R15`
`JNE NO_JOY`
	`RXSBGT R0,R1,62,1`
	`JNE NO_JOY`

More instructons.
Literal references.
Can’t do without modifying registers.

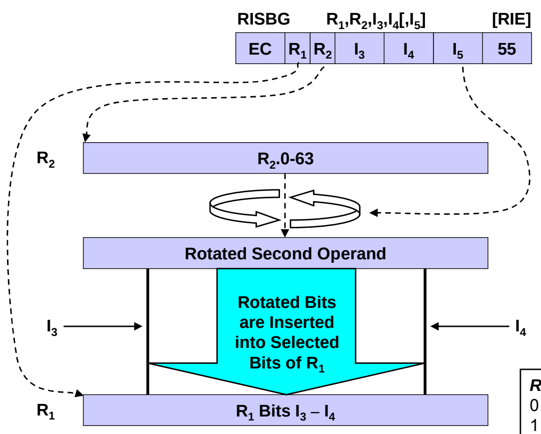
SHARE 115 44

This example uses the ROTATE THEN EXCLUSIVE OR instruction as a simple comparator.

Consider an example where we want to determine if bits 0-1 and 62-63 of two registers are identical (an obtuse example, but one that has applicability when examining control register 12 – the trace control). We’ll assume that the two registers to be compared are general registers 0 and 1, and to make matters more complicated, we’ll assume that we want to retain the register’s contents.

In the example on the left, we must make copies of the registers, then isolate the bits to be compared using an AND mask, and finally perform the comparison. In the example on the right, the ROTATE THEN EXCLUSIVE OR SELECTED BITS instruction, with the *test-results* control set accomplished this comparison with a single nondestructive operation.

ROTATE THEN INSERT SELECTED BITS (RISBG)



Remaining bits of R_1 either:

- **Left unchanged, or**
- **Set to zero**

Depending on the Z control (bit 0 of the I_4 field)

Resulting Condition Code:

0	Result zero
1	Result less than zero
2	Result greater than zero
3	--

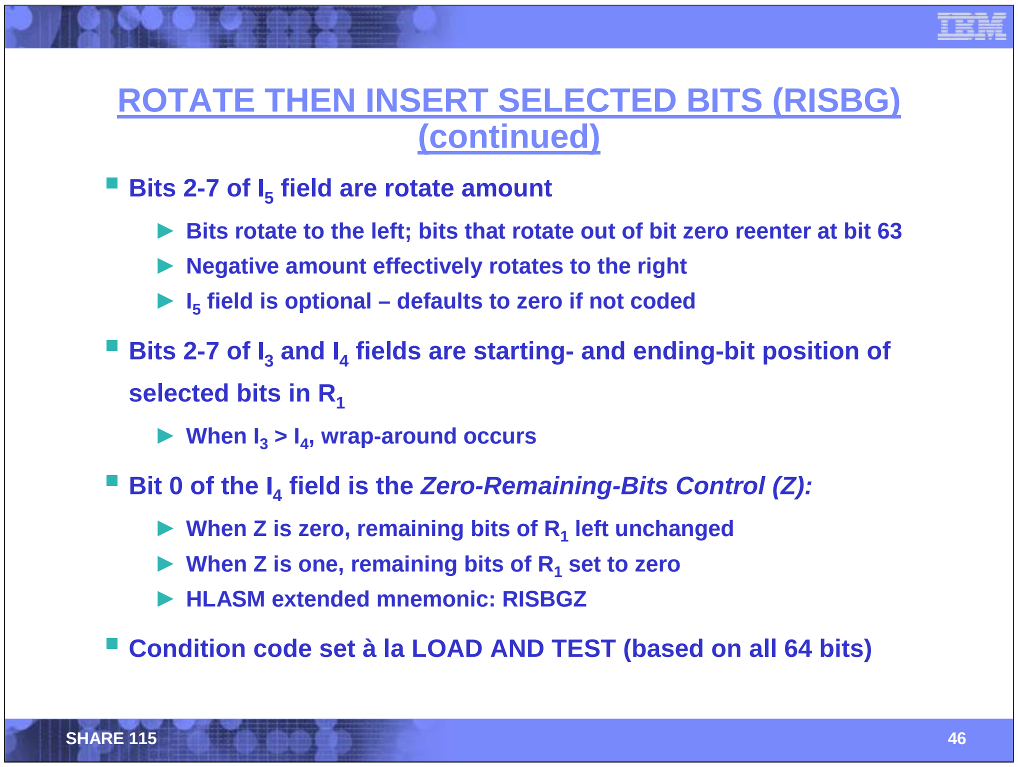
The ROTATE THEN INSERT SELECTED BITS (RISBG) instruction rotates the value contained in the second-operand register by the number of bits specified in the I_5 field. However, the contents of the second-operand register remain unchanged.

Subsequently, the selected range of the rotated bits are inserted into the corresponding bits of the first-operand register. The selected range of bits is specified by the I_3 and I_4 fields.

Bit 0 of the I_4 field contains the zero-remaining-bits control (Z). When the Z bit is zero, the remaining (non-selected) bits of the first operand remain unchanged; when the Z bit is one, the remaining bits of the first operand are set to zero.

The condition code is set based on the entire contents of the first-operand result register, similar to that of LOAD AND TEST.

Note: HLASM treats the I_5 field as optional. If not specified, a rotate amount of zero is used.



ROTATE THEN INSERT SELECTED BITS (RISBG) (continued)

- Bits 2-7 of I_5 field are rotate amount
  - Bits rotate to the left; bits that rotate out of bit zero reenter at bit 63
  - Negative amount effectively rotates to the right
  - I_5 field is optional – defaults to zero if not coded
- Bits 2-7 of I_3 and I_4 fields are starting- and ending-bit position of selected bits in R_1
  - When $I_3 > I_4$, wrap-around occurs
- Bit 0 of the I_4 field is the ***Zero-Remaining-Bits Control (Z):***
  - When Z is zero, remaining bits of R_1 left unchanged
  - When Z is one, remaining bits of R_1 set to zero
  - HLASM extended mnemonic: RISBGZ
- Condition code set à la LOAD AND TEST (based on all 64 bits)

SHARE 115 46

Bits 2-7 of the I_3, I_4, and I_5 fields contain the starting bit position of the selected bits, then ending-bit position of the selected bits, and the rotate amount, respectively.

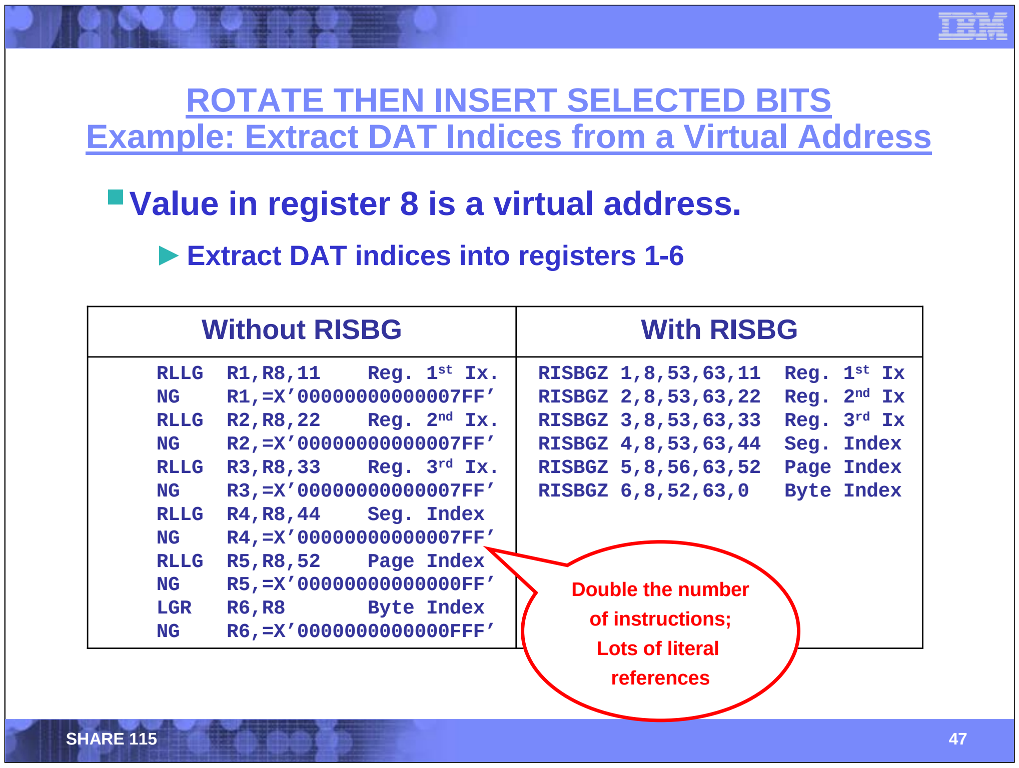
When the ending-bit position is less than the starting-bit position, the selected bits wrap around in a manner similar to that of RNSBG, ROSBG, and RXSBG.

The zero-remaining-results control (Z) is contained in bit 0 of the I_4 field. When the Z bit is zero, the remaining (non-selected) bits of the first-operand register are unmodified. When the Z bit is one, the remaining bits of the first-operand register are set to zero.

The High Level Assembler provides a mnemonic extension, “Z” which causes a value of X’80’ to be ORed into the I_4 field. Thus the following statement sets the Z bit to one, without any complicated encoding of the fourth operand:

```
RISBGZ  R1,R2,60,3,24
```

The High Level Assembler also treats the fifth operand (the rotation amount) as being optional. If the operand is not specified, no rotation is performed.



ROTATE THEN INSERT SELECTED BITS
Example: Extract DAT Indices from a Virtual Address

- Value in register 8 is a virtual address.
  - Extract DAT indices into registers 1-6

Without RISBG	With RISBG
`RLLG  R1,R8,11     Reg. 1st Ix.`	`RISBGZ 1,8,53,63,11  Reg. 1st Ix`
`NG    R1,=X'00000000000007FF'`	`RISBGZ 2,8,53,63,22  Reg. 2nd Ix`
`RLLG  R2,R8,22     Reg. 2nd Ix.`	`RISBGZ 3,8,53,63,33  Reg. 3rd Ix`
`NG    R2,=X'00000000000007FF'`	`RISBGZ 4,8,53,63,44  Seg. Index`
`RLLG  R3,R8,33     Reg. 3rd Ix.`	`RISBGZ 5,8,56,63,52  Page Index`
`NG    R3,=X'00000000000007FF'`	`RISBGZ 6,8,52,63,0   Byte Index`
`RLLG  R4,R8,44     Seg. Index`	
`NG    R4,=X'00000000000007FF'`	
`RLLG  R5,R8,52     Page Index`	
`NG    R5,=X'00000000000000FF'`	
`LGR   R6,R8        Byte Index`	
`NG    R6,=X'0000000000000FFF'`	

Double the number of instructions; Lots of literal references

This slide contains an example of using ROTATE THEN INSERT SELECTED BITS with the *zero-remaining-bits* control set to one.

In this example, we are given a virtual address in general register 8. The code extracts the various indices used by the dynamic-address-translation (DAT) process:

* Region first index (bits 0-10)

* Region second index (bits 11-21)

* Region third index (bits 22-32)

* Segment index (bits 33-43)

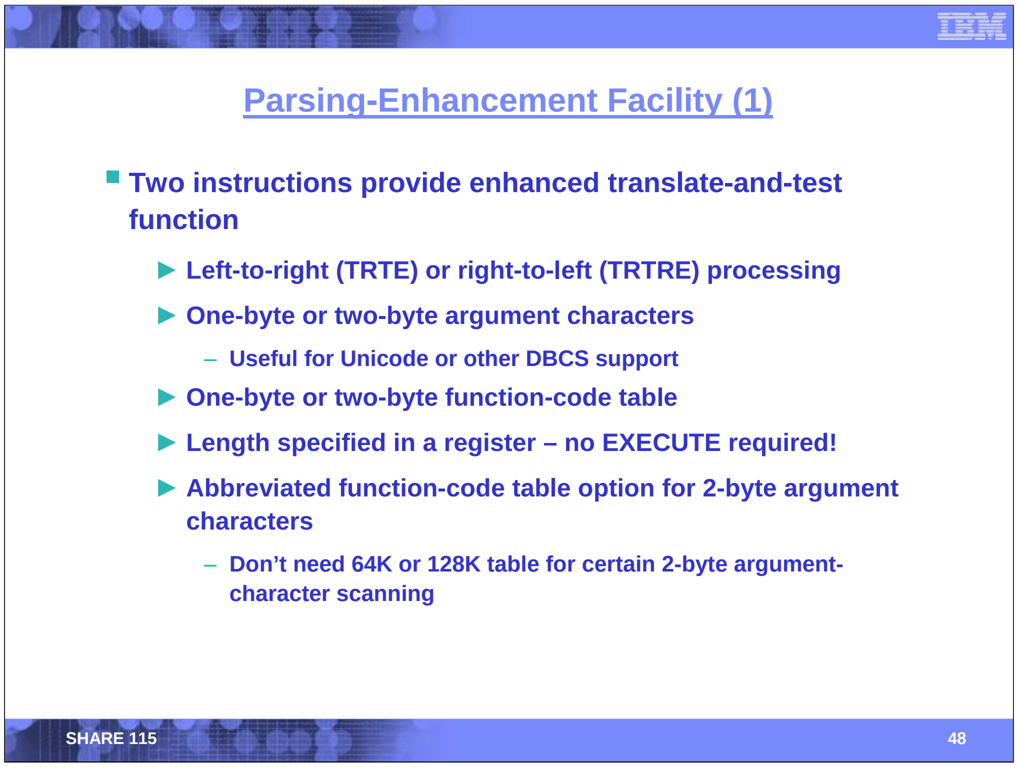
* Page index (bits 44-51), and

* Byte index (bits 52-63)

into the rightmost bits of general registers 1-6, respectively.

The example on the left accomplishes this task using ROTATE (RLLG) instructions, followed by AND instructions to isolate the applicable bits. SHIFT RIGHT LOGICAL (SRLG) could also be used in lieu of the ROTATE instruction.

The example on the right uses ROTATE THEN INSERT SELECTED BITS. The selected-bits operation, combined with the zero-remaining-bits operation makes this extremely efficient.



Parsing-Enhancement Facility (1)

- Two instructions provide enhanced translate-and-test function
  - Left-to-right (TRTE) or right-to-left (TRTRE) processing
  - One-byte or two-byte argument characters
    - Useful for Unicode or other DBCS support
  - One-byte or two-byte function-code table
  - Length specified in a register – no EXECUTE required!
  - Abbreviated function-code table option for 2-byte argument characters
    - Don't need 64K or 128K table for certain 2-byte argument-character scanning

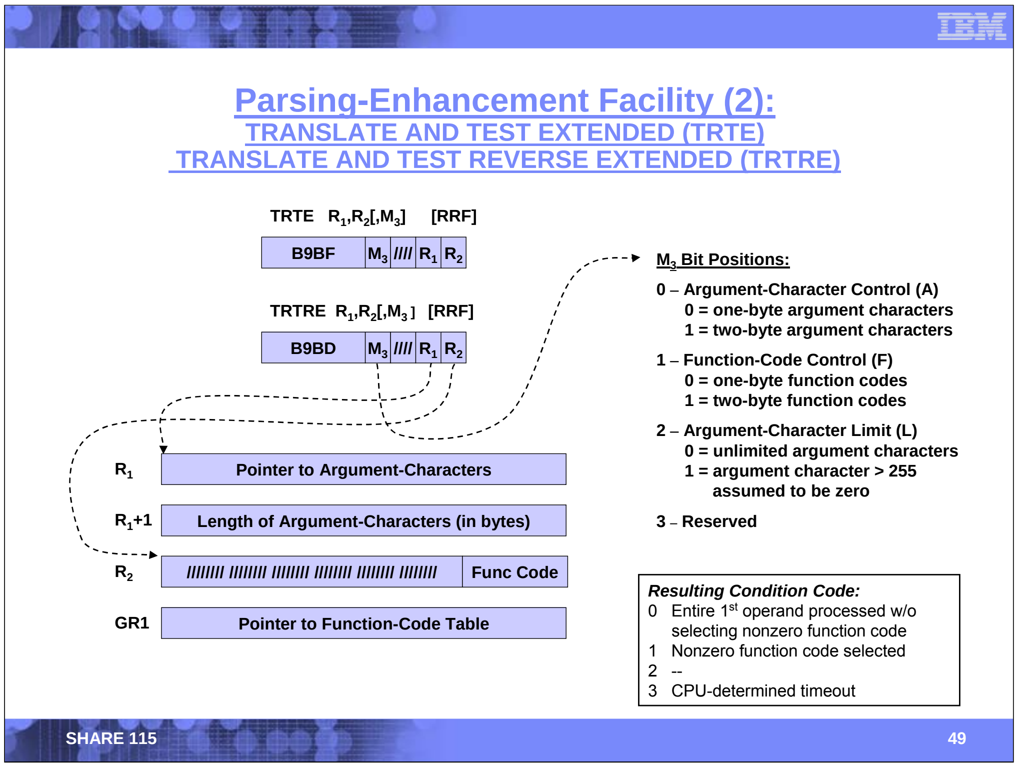
SHARE 115 48

Ever since the IBM S/360 was introduced in 1964, TRANSLATE AND TEST (TRT) has provided a remarkably efficient means of parsing text. The TRANSLATE AND TEST REVERSE (TRTR) instruction, added with the extended-translation facility 3 (2004), provides a right-to-left analog to TRT's left-to-right processing. With well crafted function-code tables and corresponding function-code tables, a programmer can implement a software implementation of a sophisticated parsing state machine.

However, TRT and TRTR are limited to one-byte argument and function characters. Therefore, TRT and TRTR are less useful in parsing modern multi-byte character representations such as Unicode™. Furthermore, with TRT and TRTR, the length of the data being parsed is part of the instruction's text, thus requiring more complicated use of EXECUTE (EX) to supply a variable length.

The parsing-enhancement facility addresses these limitations by providing two extended forms of the aforementioned instructions.

The next few slides provide an overview of the extended forms of TRT and TRTR (aptly called TRTE and TRTRE, respectively). These slides were extracted from my SHARE 113 session 1245 which gave a much deeper discussion of parsing using these instructions. For more detail, see that session.



Parsing-Enhancement Facility (2):
TRANSLATE AND TEST EXTENDED (TRTE)
TRANSLATE AND TEST REVERSE EXTENDED (TRTRE)

TRTE R_1,R_2[,M_3] [RRF]

B9BF	M_3	////	R_1	R_2

TRTRE R_1,R_2[,M_3] [RRF]

B9BD	M_3	////	R_1	R_2

R_1	Pointer to Argument-Characters	
R_1+1	Length of Argument-Characters (in bytes)	
R_2	//////// //////// //////// //////// //////// ////////	Func Code
GR1	Pointer to Function-Code Table	

M_3 Bit Positions:

0 – Argument-Character Control (A)
  0 = one-byte argument characters
  1 = two-byte argument characters

1 – Function-Code Control (F)
  0 = one-byte function codes
  1 = two-byte function codes

2 – Argument-Character Limit (L)
  0 = unlimited argument characters
  1 = argument character > 255 assumed to be zero

3 – Reserved

Resulting Condition Code:

0	Entire 1st operand processed w/o selecting nonzero function code
1	Nonzero function code selected
2	--
3	CPU-determined timeout

For both TRANSLATE AND TEST EXTENDED (TRTE) and TRANSLATE AND TEST REVERSE EXTENDED (TRTRE), the first operand designates an even/odd pair of registers. The even-numbered register contains the address of the argument characters to be scanned; the odd-numbered register contains the true length of the first operand (in bytes). General register 1 contains the address of the function-code table (sometimes referred to as the translate table).

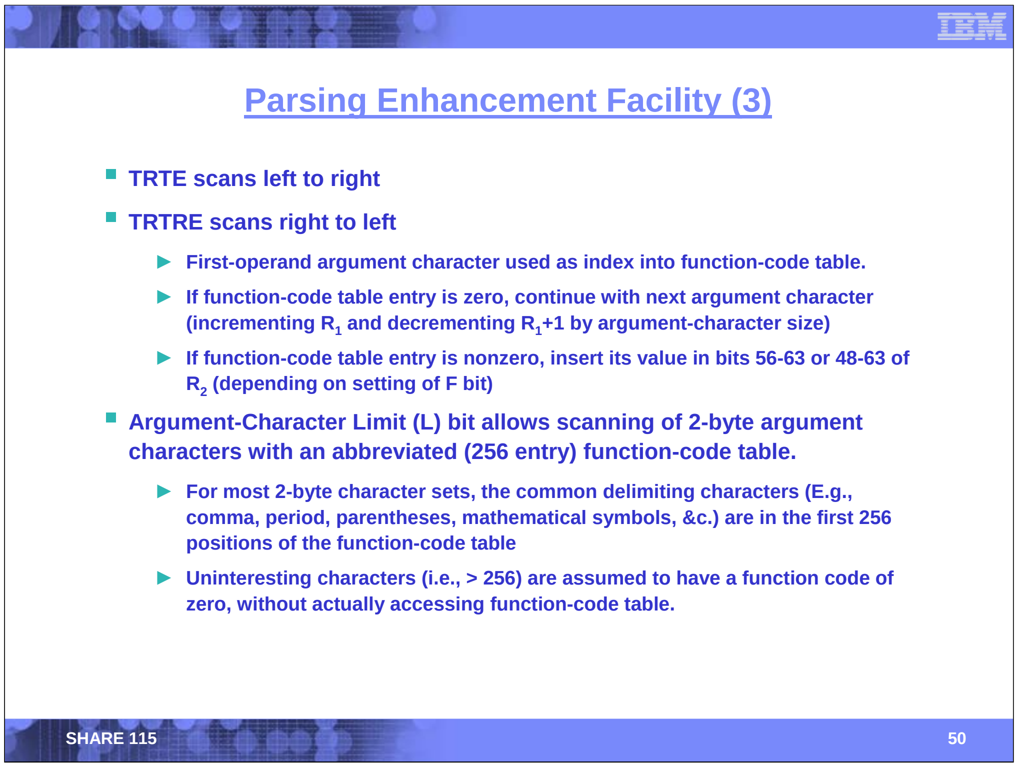
The M_3 field of the instruction contains three separate binary controls that affect the operation of the instruction:

• Bit position 0 of the M_3 field contains the argument-character size control (A). When A is zero, argument characters are one byte; when A is one, argument characters are two bytes.

• Bit position 1 of the M_3 field contains the function-character size control (F). When F is zero, function characters are one byte; when F is one, function characters are two bytes.

• Bit position 3 of the M_3 field contains the argument-character limit control (L). When L is zero, argument characters are always used to index into the function table. When L is one, any argument character greater than 255 is assumed to designate a function code of zero, without actually having to examine the function table. More on this on the next slide.

When an argument character causes a non-zero function code to be fetched, that function code is inserted in the rightmost bits of the second-operand register.



Parsing Enhancement Facility (3)

- TRTE scans left to right
- TRTRE scans right to left
  - First-operand argument character used as index into function-code table.
  - If function-code table entry is zero, continue with next argument character (incrementing R_1 and decrementing R_1+1 by argument-character size)
  - If function-code table entry is nonzero, insert its value in bits 56-63 or 48-63 of R_2 (depending on setting of F bit)
- Argument-Character Limit (L) bit allows scanning of 2-byte argument characters with an abbreviated (256 entry) function-code table.
  - For most 2-byte character sets, the common delimiting characters (E.g., comma, period, parentheses, mathematical symbols, &c.) are in the first 256 positions of the function-code table
  - Uninteresting characters (i.e., > 256) are assumed to have a function code of zero, without actually accessing function-code table.

SHARE 115 50

Operation of TRTE and TRTRE is similar to that of TRT and TRTR: The next argument character is fetched from first-operand location and used as an index into the function table. The size of the argument character is determined by the A control in the M_3 field.

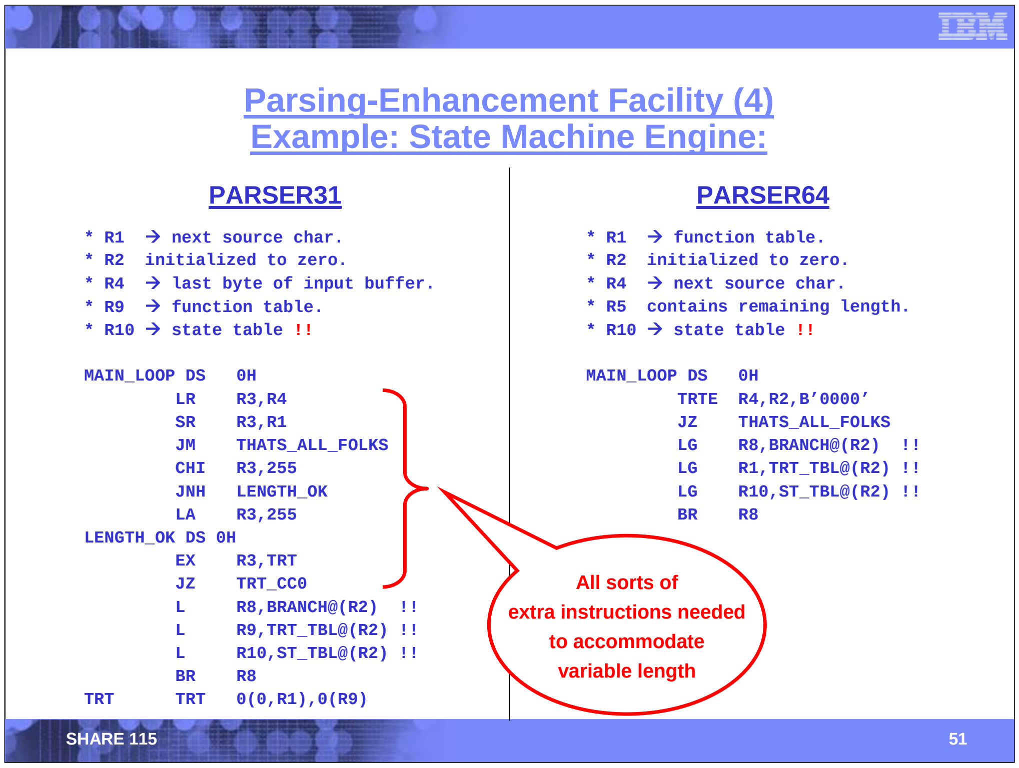
The function code designated by the argument character is examined. The size of the function code is determined by the F control in the M_3 field. If the function code is zero, then processing continues with the next argument character; otherwise, execution completes. If the function code is nonzero, then the code is inserted into general register R_2, and the instruction completes with condition code 1.

If the entire first operand is processed without locating a nonzero function code, then execution completes with condition code 0.

If a model-dependent number of characters has been processed, then execution completes with condition code 3.

When the instruction completes, the R_1+1 register has been decremented by the number of argument bytes processed, and the R_1 register is incremented by the same amount.

In many Unicode character representations, characters that are commonly used as syntactical delimiters (such as commas, parentheses, mathematical symbols, &c.) are contained in the first 256 positions of the character set. The L bit of the M_3 field allows a reduced-size function-code table to be used, even when the arguments are two characters each.



Parsing-Enhancement Facility (4)
Example: State Machine Engine:

PARSER31

```
* R1  → next source char.
* R2  initialized to zero.
* R4  → last byte of input buffer.
* R9  → function table.
* R10 → state table !!

MAIN_LOOP DS   0H
          LR   R3,R4
          SR   R3,R1
          JM   THATS_ALL_FOLKS
          CHI  R3,255
          JNH  LENGTH_OK
          LA   R3,255
LENGTH_OK DS 0H
          EX   R3,TRT
          JZ   TRT_CC0
          L    R8,BRANCH@(R2)  !!
          L    R9,TRT_TBL@(R2) !!
          L    R10,ST_TBL@(R2) !!
          BR   R8
TRT       TRT  0(0,R1),0(R9)
```

PARSER64

```
* R1  → function table.
* R2  initialized to zero.
* R4  → next source char.
* R5  contains remaining length.
* R10 → state table !!

MAIN_LOOP DS   0H
          TRTE R4,R2,B'0000'
          JZ   THATS_ALL_FOLKS
          LG   R8,BRANCH@(R2)  !!
          LG   R1,TRT_TBL@(R2) !!
          LG   R10,ST_TBL@(R2) !!
          BR   R8
```

All sorts of extra instructions needed to accommodate variable length

SHARE 115 51

This slide compares two simple parsing functions, PARSER31 and PARSER64. The former uses instructions only available on ESA/390, whereas the latter uses z/Architecture instructions – specifically the TRANSLATE AND TEST EXTENDED instruction introduced on the z10.

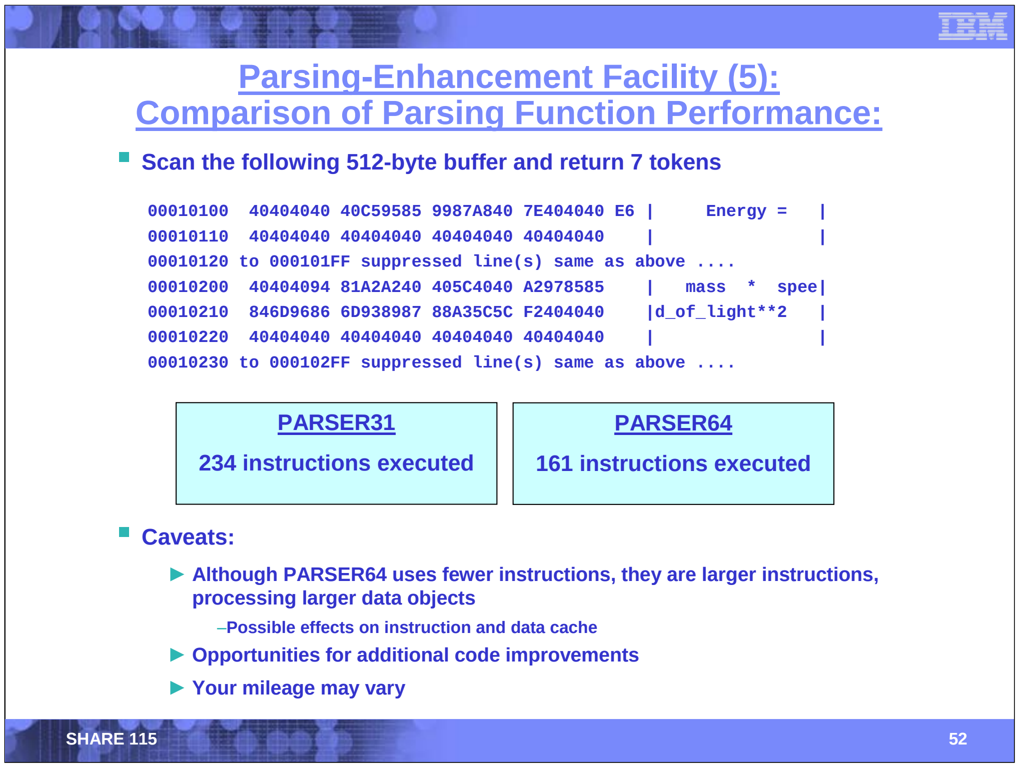
Given that the registers are pre-initialized as shown, this slide shows the core of the parser functions. This technique puts no limits on the size of the buffer being parsed.

PARSER31's code requires more instructions as it must use an executed TRT to account for the variable length. Thus PARSER31 must also account for an input buffer that is larger than 256 characters; the TRT_CC0 code (not shown on this slide) allows PARSER31 to scan a buffer that is larger than 256 characters.

Note that the register usage for PARSER64 is somewhat different. This is because it uses the TRANSLATE AND TEST EXTENDED (TRTE) instruction.

In both cases, the functions use register 10 to point to the current state (triplet) table. Following the executed TRT or the TRTE instruction are three LOAD instructions that load the registers determining the next state: branch address, function-table address, and next state-table address. The second operand for each of these loads uses register 10 as the base and register 2 as the index; register 2 was set by the TRT or TRTE instruction, based on the nonzero function code that was encountered.

See SHARE 113 session 1245 for more details on the operation of these two functions.



Parsing-Enhancement Facility (5): Comparison of Parsing Function Performance:

- Scan the following 512-byte buffer and return 7 tokens

```
00010100  40404040 40C59585 9987A840 7E404040 E6 |     Energy =   |
00010110  40404040 40404040 40404040 40404040    |                |
00010120 to 000101FF suppressed line(s) same as above ....
00010200  40404094 81A2A240 405C4040 A2978585    |   mass  *  spee|
00010210  846D9686 6D938987 88A35C5C F2404040    |d_of_light**2   |
00010220  40404040 40404040 40404040 40404040    |                |
00010230 to 000102FF suppressed line(s) same as above ....
```

PARSER31	PARSER64
234 instructions executed	161 instructions executed

- Caveats:
  - Although PARSER64 uses fewer instructions, they are larger instructions, processing larger data objects
    - –Possible effects on instruction and data cache
  - Opportunities for additional code improvements
  - Your mileage may vary

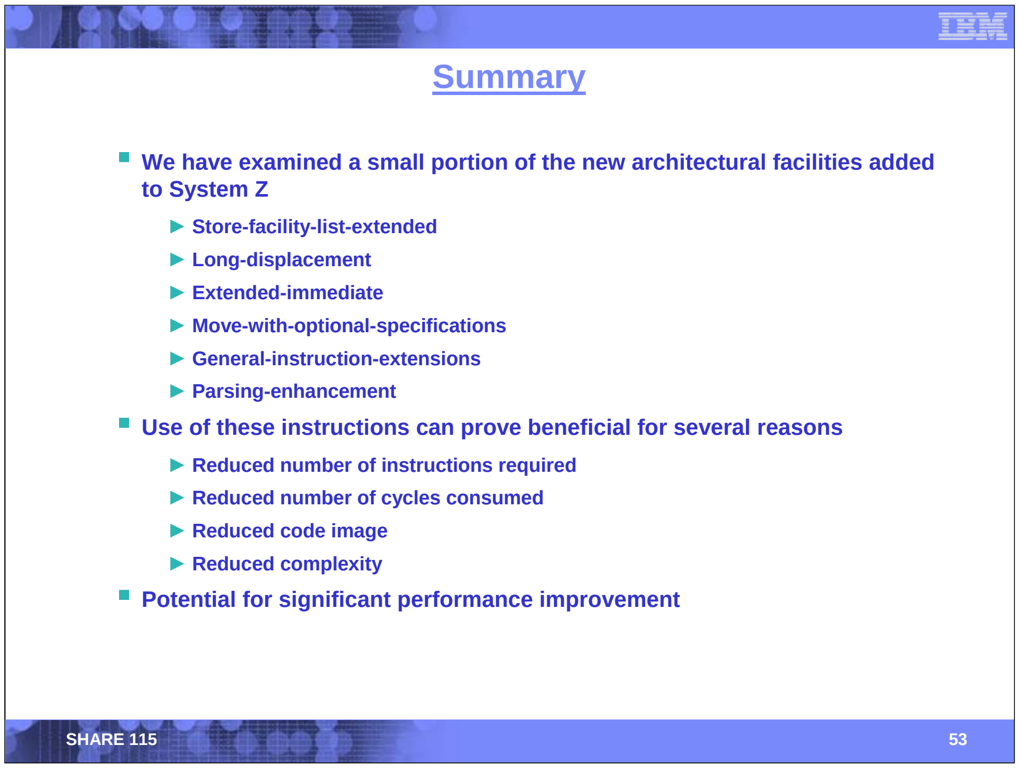
SHARE 115 52

This slide shows the results of scanning the statement shown. This statement is spread over a 512-byte buffer beginning at location 10100 hex.

The advantages of PARSER64's use of TRANSLATE AND TEST EXTENDED can be seen in that it uses substantially fewer instructions.

In fairness, we must disclose that there is a slight penalty for using 64-bit mode instructions. Obviously, the 64-bit addresses used by PARSER64 consume more space in the data cache than the 31-bit addresses used by PARSER31. Some 64-bit z/Architecture instructions – particularly those that access storage – may be larger than their 31-bit ESA/390 analogues (6 bytes versus 4). Thus the 64-bit model may also take slightly more instruction cache. However, the software implementation of the finite-state machine implemented in both functions is extremely tight code, thus a minimum of cache lines are used in either case.

The performance improvement from using the newer instructions – particularly TRANSLATE AND TEST EXTENDED – is far more significant than any minor cache-hit penalty from using 64-bit addressing.



Summary

- We have examined a small portion of the new architectural facilities added to System Z
  - Store-facility-list-extended
  - Long-displacement
  - Extended-immediate
  - Move-with-optional-specifications
  - General-instruction-extensions
  - Parsing-enhancement
- Use of these instructions can prove beneficial for several reasons
  - Reduced number of instructions required
  - Reduced number of cycles consumed
  - Reduced code image
  - Reduced complexity
- Potential for significant performance improvement

SHARE 115 53

From the introduction of z/Architecture in 2000 to the System z10 in 2008, there have been approximately 283 new instructions added. (I say approximately, as I did a rough count when making up this slide, but it's pretty close … suffice it to say that this is not a RISC architecture.) Thus, in a one hour session, I've barely had the opportunity to scratch the surface.

We've examined some of the major facilities that have been introduced, and shown concrete examples of how these instructions can improve the performance of your applications. (Store-facility-list-extended really doesn't get you any improved performance, but it gives the application program an accurate indication of what other facilities are present.) By using these facilities, you can:

1. Reduce the number of instructions required to perform a complex operation. In the FLOGR example, when searching through all 64 bits of a register, the single instruction is 80x to 100x faster than the coding example on the left (and I squeezed every cycle out of that example that I could).

2. By reducing the number of storage references, the newer code can significantly reduce the number of cycles that a program spends accessing memory – particularly when an operand is not in the cache.

3. By reducing the constants and literals in storage, the program's size is reduced, requiring fewer cache lines.

4. In some cases, a single instruction can replace numerous other, more complicated code sequences.

Judicious use of the newer z/Architecture instructions has the potential of significantly improving the performance of applications.

Questions?

SHARE 115

54

For those in the live audience, I will gladly entertain questions here.

For those who view this on the SHARE web site, your questions are also welcome. My email address is listed on the first slide.