New CPU Facilities in the IBM zEnterprise™ 196

Dan Greiner
dgreiner@us.ibm.com
z/Server Architecture
SHARE 116 in Anaheim
Session 8546, 2 March 2011, 3:00 pm

IBM Systems and Technology Group (STG)

© Copyright International Business Machines Corporation 2010-2011.

Note: This PowerPoint presentation 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

- **Trademarks:**
  - **The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:**
    - **ESA/390**
    - **IBM**
    - **z/Architecture**
    - **z/OS**
    - **z/VM**
  - **IEEE is a trademark of the Institute of Electrical and Electronics Engineers, Inc. in the United States, other countries, or both.**
  - **Linux is a registered trademark of Linus Torvalds in the United States, other countries or both.**
  - **Unicode is a registered trademark of Unicode, Incorporated in the United States, other countries, or both.**
  - **Other trademarks and registered trademarks are the properties of their respective companies.**
- All information contained in this document is subject to change without notice. The products described in this document are not intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.
- While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.
- The information in contained in this document is provided on an “AS IS” basis. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

© Copyright International Business Machines Corporation 2010-2011. Permission is granted to SHARE, Inc. to publish this presentation in the proceedings of SHARE 116.

Topics du Jour

- High-Word Facility
- Interlocked-Access Facility
- Load/Store-on-Condition Facility
- Distinct-Operands Facility
- Population-Count Facility
- Floating-Point-Extension Facility
- Message-Security-Assist Extension 3
- Message-Security-Assist Extension 4
- Miscellaneous Enhancements

This presentation reviews the new CPU facilities introduced (mostly) by the IBM zEnterprise 196 series of processors (the one exception is the message-security-assist extension 3 [MSA-X3] which was introduced in the System z10 GA3 machines, but was not previously published).

The major focus is on general instructions used by various high-level languages such as C and Java. The final slides will address a few other facilities available for authorized programs.

If you have a PowerPoint version of the presentation, this slide, and the section headings that they designate, contain hyperlinks to the various topics and subtopics. Each slide containing specific information has an “Index” hiperlink in the bottom-right corner that will return you to the next-higher level of information. (Note, SHARE limits their download page to PDFs; if you want the PowerPoint show, see me after the presentation, or send a note to dgreiner@us.ibm.com.)

High-Word Facility (1)

- Suite of instructions to manipulate bits 0-31 of a GPR
- For purposes of address-generation interlock (AGI), leftmost bits (0-31) are treated separately from rightmost bits (32-63)
- Intended to provide register-constraint relief for compilers
- Installation of the high-word facility (& al.) indicated by facility bit 45

Since its introduction in 1964, System 360 and all of its successors have provided 16 general-purpose registers. To alleviate the constraint felt by many programmers, numerous architectural features have been added: The relative branching (short and long) facilities, immediate- and extended-immediate-operand facilities, and the long displacement facility are a few examples. However, the 16-register limit continues to prove daunting to both assembler programmers and compiler designers alike.

Although z/Architecture provides 64-bit addressing and arithmetic, many applications continue to operate in the 31-bit addressing mode, and rarely require higher-precision arithmetic than 32 bits. For such programs, the leftmost 32 bits of the 64-bit registers have been of little use … until now.

The high-word facility provides a means by which selected new instructions can operate on the leftmost 32 bits (bits 0-31) of a general register – independent of the rightmost 32 bits (bits 32-63). This separation extends into address generation performed while in the 24- or 31-bit addressing modes; the updating of the leftmost 32 bits of a general-purpose register, using the high-word instructions, does not affect any pipeline address-generation interlock used by the rightmost 32 bits.

Several of the facilities discussed in this presentation share a common facility bit. Bit 45 indicates the installation of the high-word, interlocked-access, load/store-on-condition, distinct-operands, population-count, and fast-BCR-serialization facilities.

High-Word Facility (2):

Instruction	Mnemonic	OpCode	1st Operand	2nd Operand	3rd Operand
ADD HIGH	AHHHR	B9C8	R_1.0-31	R_2.0-31	R_3.0-31
ADD HIGH	AHHLR	B9D8	R_1.0-31	R_2.0-31	R_3.32-63
ADD HIGH IMMEDIATE	AIH	CC8	R_1.0-31	I_2 [32 bits]	—
ADD LOGICAL HIGH	ALHHHR	B9CA	R_1.0-31	R_2.0-32	R_3.0-31
ADD LOGICAL HIGH	ALHHLR	B9DA	R_1.0-31	R_2.0-32	R_3.32-63
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH	ALSIH	CCA	R_1.0-31	I_2 [32 bits]	—
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH	ALSIHN	CCB	R_1.0-31	I_2 [32 bits]	—
BRANCH RELATIVE ON COUNT HIGH	BRCTH	CC6	R_1.0-31	RI_2 [16 bits]	—
COMPARE HIGH	CHHR	B9CD	R_1.0-31	R_2.0-31	—
COMPARE HIGH	CHLR	B9DD	R_1.0-31	R_2.32-63	—
COMPARE HIGH	CHF	E3CD	R_1.0-31	S20 [32 bits]	—
COMPARE IMMEDIATE HIGH	CIH	CCD	R_1.0-31	I_2 [32 bits]	—

Explanation:

—	Not applicable
I_2	Second operand is an immediate value
RI_2	Second operand is a relative-immediate branch location
R_n	Register operand 'n'
S20	Storage operand designated by base and index registers with 20-bit signed long displacement

This slide enumerates the first 12 instructions in the high-word facility; the remainder are listed on the following slide. As will be immediately obvious, only a limited subset of the instructions are provided to manipulate the high words: ADD, ADD LOGICAL, BRANCH RELATIVE ON COUNT, COMPARE, COMPARE LOGICAL, LOAD BYTE, LOAD HALFWORD, LOAD, LOAD LOGICAL CHARACTER, LOAD LOGICAL HALFWORD, ROTATE THEN INSERT SELECTED BITS, STORE CHARACTER, STORE HALFWORD, STORE, SUBTRACT and SUBTRACT LOGICAL.

Note that many of the arithmetic-operand instructions have distinct operands; that is, the target register is separate from the two source registers.

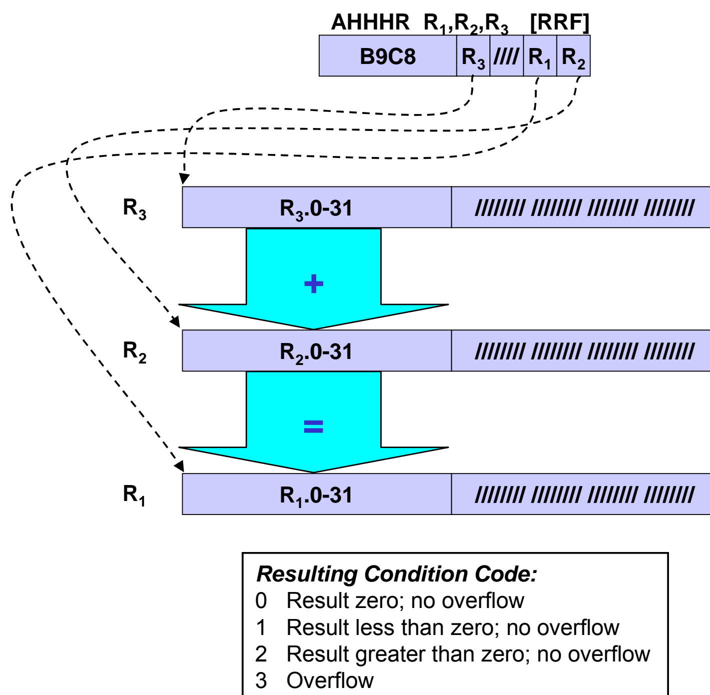
Also note that, of necessity, certain characters in the mnemonics have become a bit overloaded. The rookie programmer will likely find using the high-word facility challenging. We hope the benefits will be worth it.

High-Word Facility (3):

Instruction	Mnemonic	OpCode	1st Operand	2nd Operand	Other
COMPARE LOGICAL HIGH	CLHHR	B9CF	R_1.0-31	R_2.0-31	—
COMPARE LOGICAL HIGH	CLHLR	B9DF	R_1.0-31	R_2.32-63	—
COMPARE LOGICAL HIGH	CLHF	E3CF	R_1.0-31	S20 [32 bits]	—
COMPARE LOGICAL IMMEDIATE HIGH	CLIH	CCF	R_1.0-31	I_2 [32 bits]	—
LOAD BYTE HIGH	LBH	E3C0	R_1.24-31	S20 [8 BITS]	—
LOAD HALFWORD HIGH	LHH	E3C4	R_1.16-31	S20 [16 bits]	—
LOAD HIGH	LFH	E3CA	R_1.0-31	S20 [32 bits]	—
LOAD LOGICAL CHARACTER HIGH	LLCH	E3C2	R_1.24-31	S20 [8 bits]	—
LOAD LOGICAL HALFWORD HIGH	LLHH	E3C6	R_1.16-31	S20 [16 bits]	—
ROTATE THEN INSERT SELECTED BITS HIGH	RISBHG	EC5D	R_1.I_3-I_4	R_2.0-63	I_3, I_4, I_5
ROTATE THEN INSERT SELECTED BITS LOW	RISBLG	EC51	R_1.32+I_3: - 32+I_4	R_2.0-63	I_3, I_4, I_5
STORE CHARACTER HIGH	STCH	E3C3	R_1.24-31	S20 [8 bits]	—
STORE HALFWORD HIGH	STHH	E3C7	R_1.16-31	S20 [16 bits]	—
STORE HIGH	STFH	E3CB	R_1.0-31	S20 [32 bits]	—
SUBTRACT HIGH	SHHHR	B9C9	R_1.0-31	R_2.0-31	R_3.0-31
SUBTRACT HIGH	SHHLR	B9D9	R_1.0-31	R_2.0-31	R_3.32-63
SUBTRACT LOGICAL HIGH	SLHHHR	B9CB	R_1.0-31	R_2.0-31	R_3.0-31
SUBTRACT LOGICAL HIGH	SLHHLR	B9DB	R_1.0-31	R_2.0-31	R_3.32-63

This slide lists the remaining 18 instructions in the high-word facility, for a total of 30 instructions.

ADD HIGH (AHHHR)

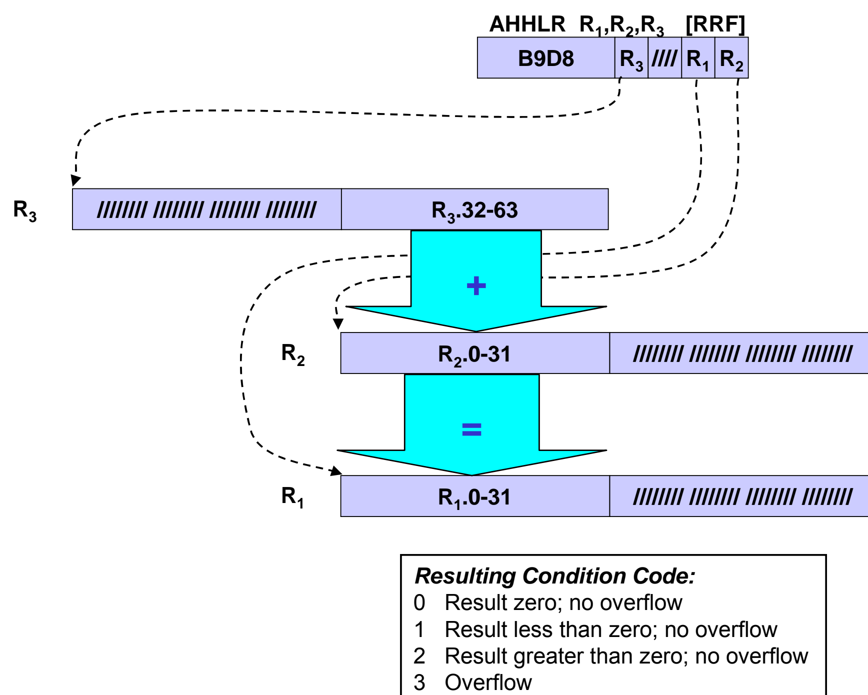


For ADD HIGH (AHHHR), the contents of the leftmost bits (0-31) of the general register designated by the R_3 field of the instruction are added to the contents of the leftmost bits of the general register designated by the R_2 field of the instruction. The results of the addition replace the leftmost bits of the general register designated by the R_1 field of the instruction; bits 32-63 of the result register remain unchanged.

The addition proceeds exactly as for ADD (AR), except that there are two source operands and a separate target operand – and, obviously, the result ends up in the left of the register.

The condition code is set as with any other signed addition operation.

ADD HIGH (AHHLR)

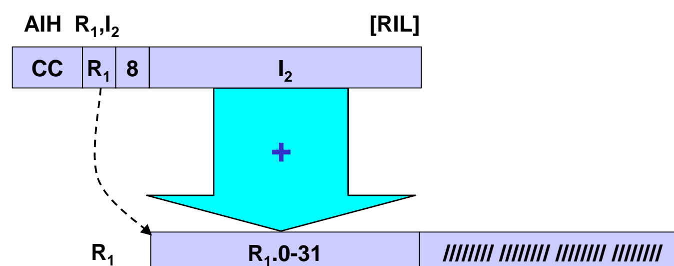


ADD HIGH (AHHLR) should perhaps be called ADD HIGH AND LOW.

The contents of the rightmost bits (32-63) of the general register designated by the R_3 field of the instruction are added to the contents of the leftmost bits (0-31) of the general register designated by the R_2 field of the instruction. The results of the addition replace the leftmost bits of the general register designated by the R_1 operand; bits 32-63 of the result register remain unchanged.

The condition code is set as with any other signed addition operation.

ADD IMMEDIATE HIGH (AIH)



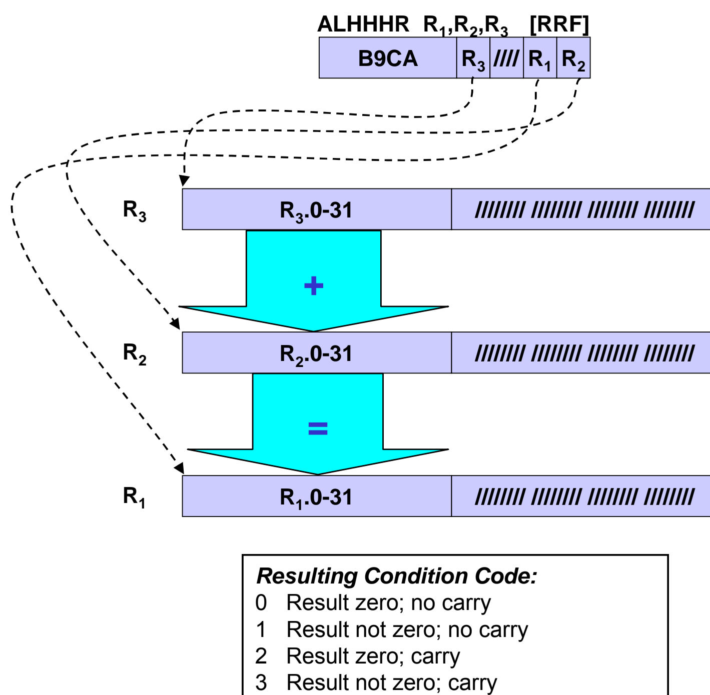
Resulting Condition Code:
0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

ADD IMMEDIATE HIGH (AIH) adds the contents of the 32-bit signed I_2 field (bits 16-47 of the instruction) with the contents of the leftmost bits (0-31) of the general register designated by the R_1 field of the instruction. The results of the addition replace the leftmost bits of the general register designated by the R_1 operand; bits 32-63 of the result register remain unchanged.

Unlike ADD HIGH (AHHHR and AHHLR), the result replaces the leftmost bits of the first-operand register.

The condition code is set as with any other signed addition operation.

ADD LOGICAL HIGH (ALHHHR)

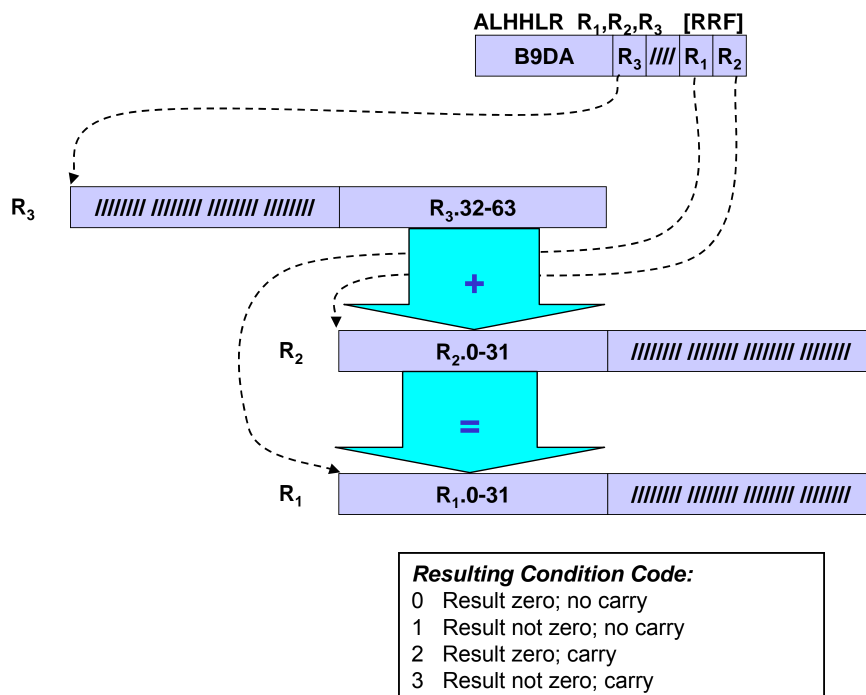


For ADD LOGICAL HIGH (ALHHHR), the contents of the leftmost bits (0-31) of the general register designated by the R_3 field of the instruction are added to the contents of the leftmost bits of the general register designated by the R_2 field of the instruction. The results of the addition replace the leftmost bits of the general register designated by the R_1 field of the instruction; bits 32-63 of the result register remain unchanged.

The addition proceeds exactly as for ADD LOGICAL (ALR), except that there are two source operands and a separate target operand – and, obviously, the result ends up in the left of the register.

The condition code is set as with any other unsigned addition operation.

ADD LOGICAL HIGH (ALHHLR)

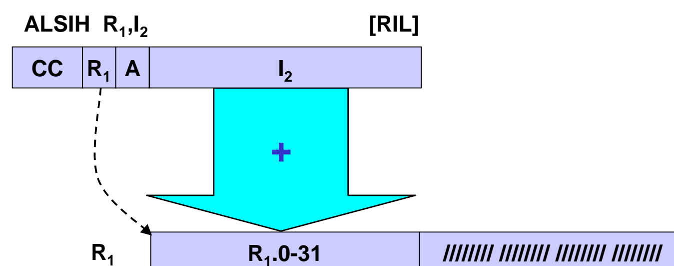


As with ADD HIGH (AHHLR), ADD LOGICAL HIGH (ALHHLR) should perhaps be called ADD LOGICAL HIGH AND LOW.

The contents of the rightmost bits (32-63) of the general register designated by the R_3 field of the instruction are added to the contents of the leftmost bits (0-31) of the general register designated by the R_2 field of the instruction. The results of the addition replace the leftmost bits of the general register designated by the R_1 operand; bits 32-63 of the result register remain unchanged.

The condition code is set as with any other unsigned addition operation.

ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (ALSIH)



Resulting Condition Code:

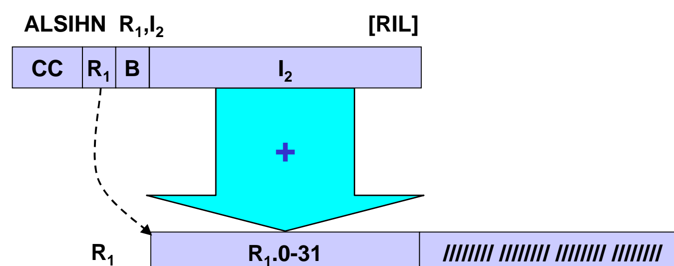
0 Result zero; no carry
1 Result not zero; no carry
2 Result zero; carry
3 Result not zero; carry

ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (ALSIH) adds the contents of the 32-bit signed I_2 field (bits 16-47 of the instruction) with the contents of the leftmost unsigned bits (0-31) of the general register designated by the R_1 field of the instruction. The results of the addition replace the leftmost bits of the general register designated by the R_1 operand; bits 32-63 of the result register remain unchanged.

As with ADD IMMEDIATE HIGH, the result replaces the leftmost bits of the first-operand register.

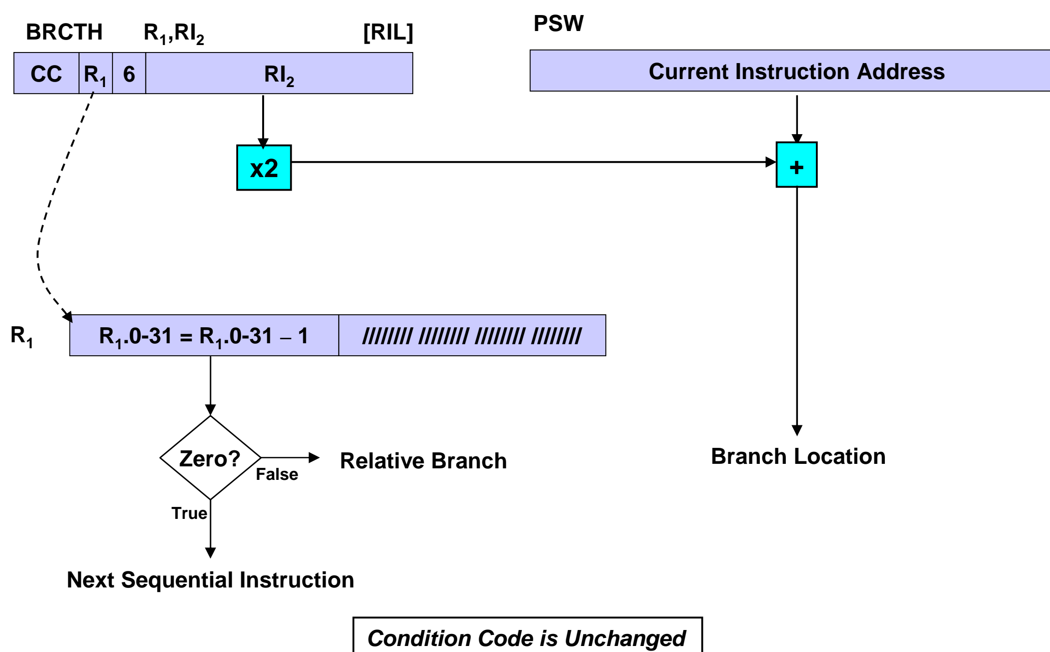
The condition code is set as with any other unsigned addition operation!! Although having the second operand be signed reduces the magnitude of the addend by a power of two, it also eliminates the need to define a separate SUBTRACT LOGICAL IMMEDIATE instruction. To subtract, one simply uses a negative second operand.

ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (ALSIHN)



Condition Code is Unchanged !

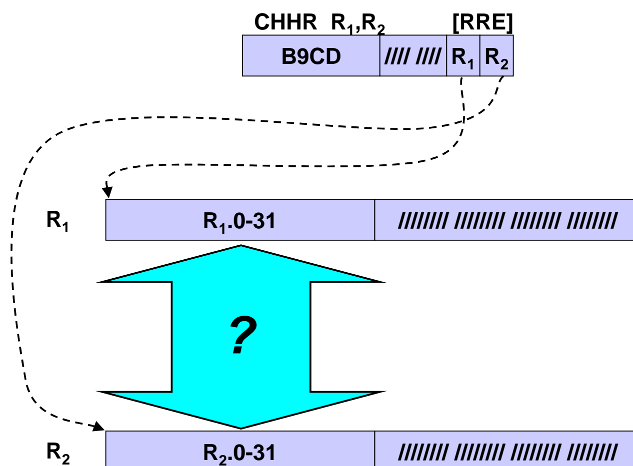
ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (ALSIHN) is identical to ADD LOGICAL WITH SIGNED IMMEDIATE HIGH (ALSIH), except that the condition code remains unchanged.



BRANCH RELATIVE ON COUNT HIGH (BRCTH) is an analog to BRANCH RELATIVE AND COUNT (BRCT). BRCTH works identically to BRCT, except that the decremented value (that is, the counter) is in the leftmost bits of the general register designated by the R_1 field of the instruction.

The rightmost 32 bits (32-63) of the counting register and the condition code remain unchanged.

COMPARE HIGH (CHHR)



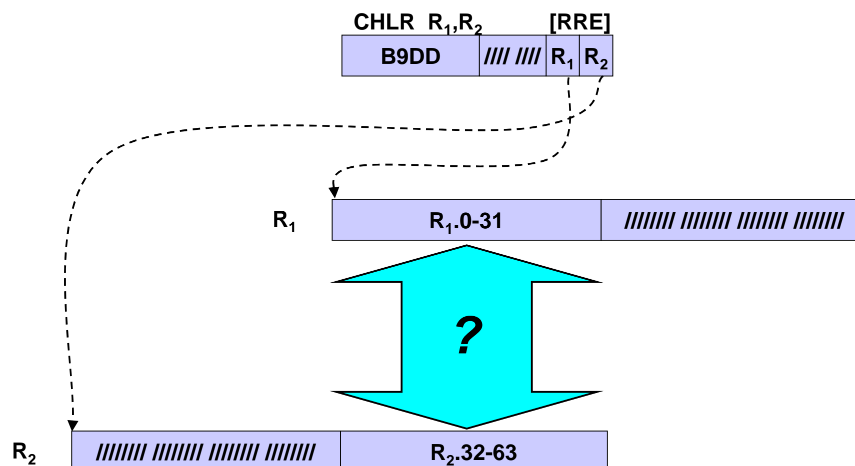
Resulting Condition Code:

0 Operands equal
1 First operand low
2 First operand high
3 —

For COMPARE HIGH (CHHR), the contents of the leftmost bits (0-31) of the general register designated by the R_2 field of the instruction are arithmetically compared with the contents of the leftmost bits of the general register designated by the R_1 field of the instruction. The rightmost 32 bits of each register are ignored.

The condition code is set as with any other signed binary arithmetic comparison.

COMPARE HIGH (CHLR)

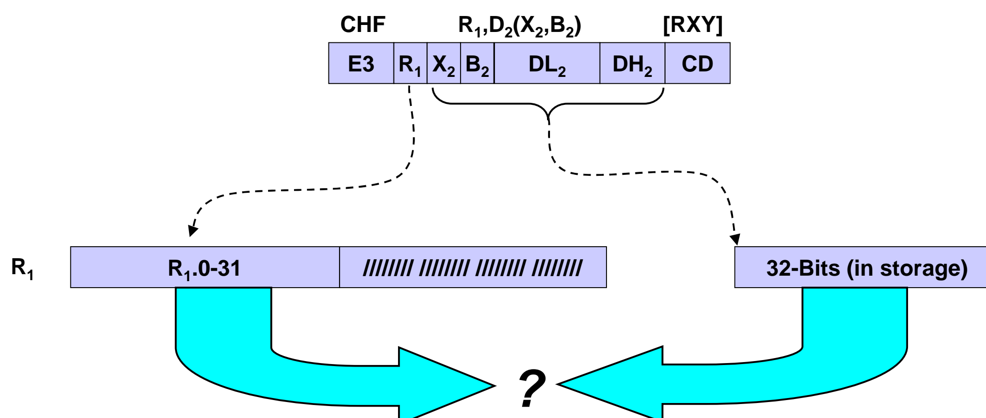


Resulting Condition Code:
0 Operands equal
1 First operand low
2 First operand high
3 —

For COMPARE HIGH (CHLR), the contents of the rightmost bits (32-63) of the general register designated by the R_2 field of the instruction are arithmetically compared with the contents of the leftmost bits (0-31) of the general register designated by the R_1 field of the instruction. The rightmost 32 bits of general register R_1 and the leftmost 32 bits of general register R_2 are ignored.

The condition code is set as with any other signed binary arithmetic comparison.

COMPARE HIGH (CHF)



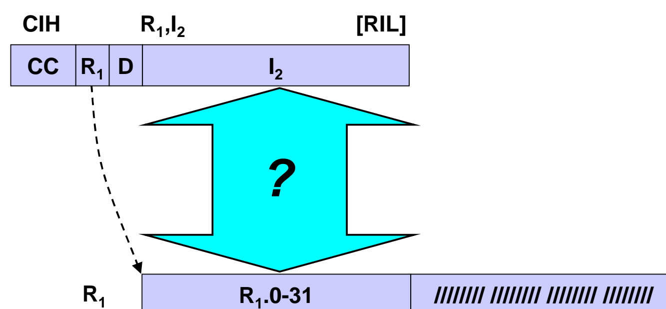
Resulting Condition Code:
0 Operands equal
1 First operand low
2 First operand high
3 --

COMPARE HIGH (CHF) is an analog to the COMPARE (C) instruction; the difference being that for CHF, the leftmost 32 bits of the register are compared.

The 32-bit second operand in storage is arithmetically compared with the contents of the leftmost bits of the general register designated by the R_1 field of the instruction. The rightmost 32 bits of general register R_1 are ignored.

The condition code is set as with any other signed binary arithmetic comparison.

COMPARE IMMEDIATE HIGH (CIH)



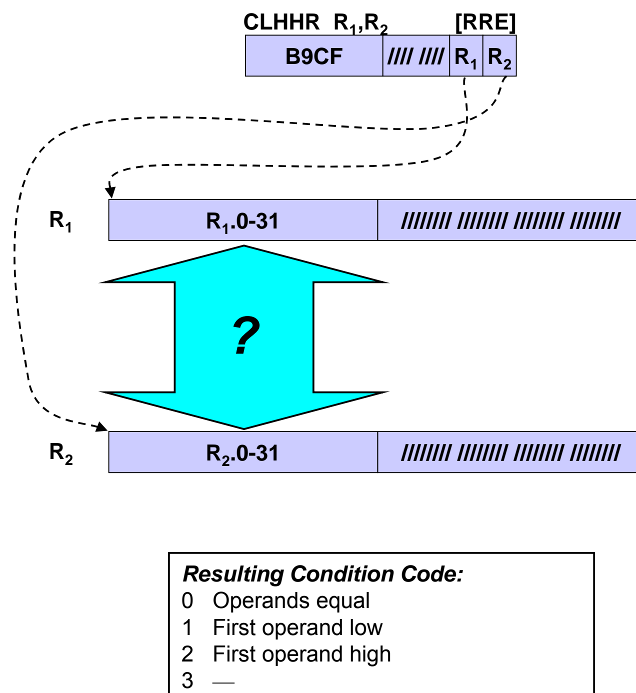
Resulting Condition Code:
0 Operands equal
1 First operand low
2 First operand high
3 —

COMPARE IMMEDIATE HIGH (CIH) is an analog to the COMPARE IMMEDIATE (CFI) instruction; the difference being that for CIH, the leftmost 32 bits of the register are compared. (CFI was introduced with the general-instruction extension facility in the System z10.)

The 32-bit second immediate field (bits 16-47) of the instruction is arithmetically compared with the contents of the leftmost bits of the general register designated by the R_1 field of the instruction. The rightmost 32 bits of general register R_1 are ignored.

The condition code is set as with any other signed binary arithmetic comparison.

COMPARE LOGICAL HIGH (CLHHR)

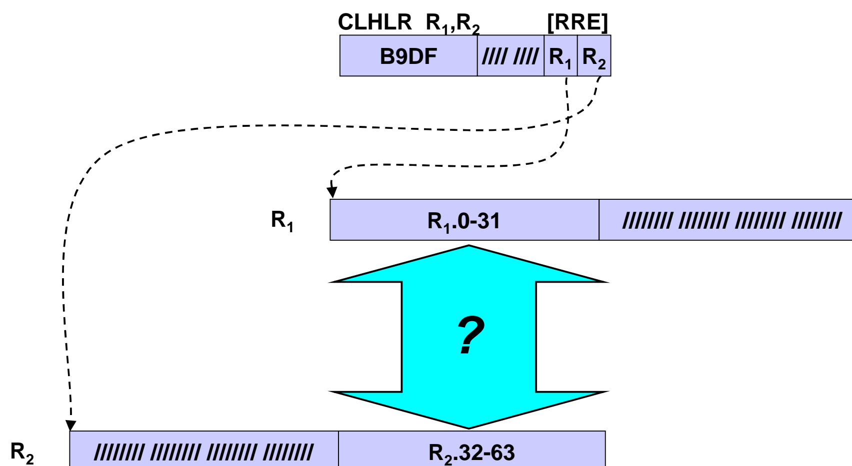


COMPARE LOGICAL HIGH (CLHHR) is an analog to the COMPARE LOGICAL (CLR) instruction; the difference being that for CLHHR, the leftmost 32 bits of the register are compared.

The contents of the leftmost bits (0-31) of the general register designated by the R_2 field of the instruction are logically compared with the contents of the leftmost bits of the general register designated by the R_1 field of the instruction. The rightmost 32 bits of each register are ignored.

The condition code is set as with any other unsigned binary arithmetic comparison.

COMPARE LOGICAL HIGH (CLHLR)

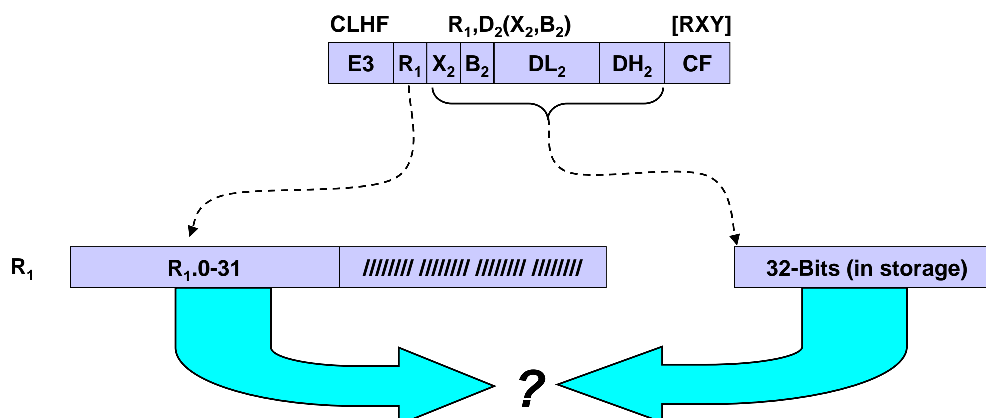


Resulting Condition Code:
0 Operands equal
1 First operand low
2 First operand high
3 —

For COMPARE LOGICAL HIGH (CLHLR), the contents of the rightmost bits (32-63) of the general register designated by the R_2 field of the instruction are logically compared with the contents of the leftmost bits (0-31) of the general register designated by the R_1 field of the instruction. The rightmost 32 bits of general register R_1 and the leftmost 32 bits of general register R_2 are ignored.

The condition code is set as with any other unsigned binary arithmetic comparison.

COMPARE LOGICAL HIGH (CLHF)



Resulting Condition Code:

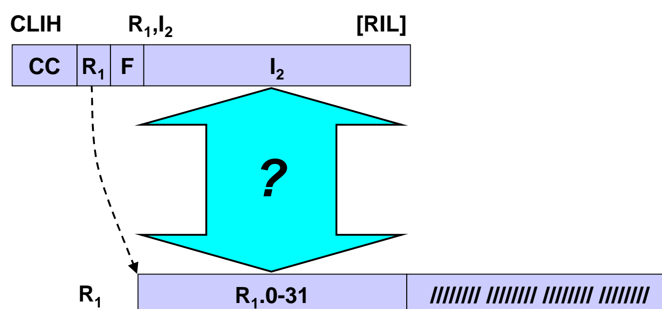
0 Operands equal
1 First operand low
2 First operand high
3 --

COMPARE LOGICAL HIGH (CLHF) is an analog to the COMPARE LOGICAL (CL) instruction; the difference being that for CLHF, the leftmost 32 bits of the register are compared.

The 32-bit second operand in storage is logically compared with the contents of the leftmost bits of the general register designated by the R_1 field of the instruction. The rightmost 32 bits of general register R_1 are ignored.

The condition code is set as with any other unsigned binary arithmetic comparison.

COMPARE LOGIGAL IMMEDIATE HIGH (CLIH)



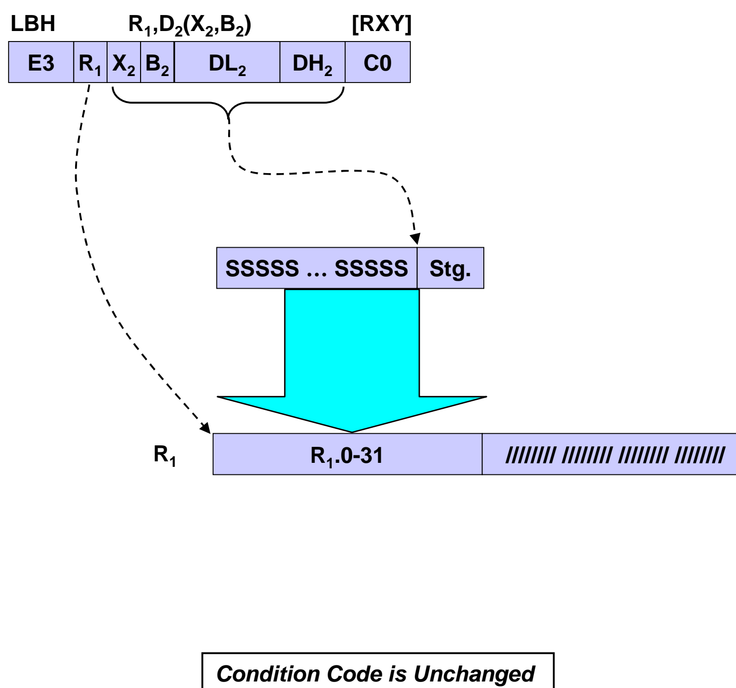
Resulting Condition Code:
0 Operands equal
1 First operand low
2 First operand high
3 —

COMPARE LOGICAL IMMEDIATE HIGH (CLIH) is an analog to the COMPARE LOGICAL IMMEDIATE (CLFI) instruction; the difference being that for CLIH, the leftmost 32 bits of the register are compared. (CLFI was introduced with the general-instructions extension facility in the System z10.)

The 32-bit second immediate field (bits 16-47) of the instruction is logically compared with the contents of the leftmost bits of the general register designated by the R_1 field of the instruction. The rightmost 32 bits of general register R_1 are ignored.

The condition code is set as with any other unsigned binary arithmetic comparison.

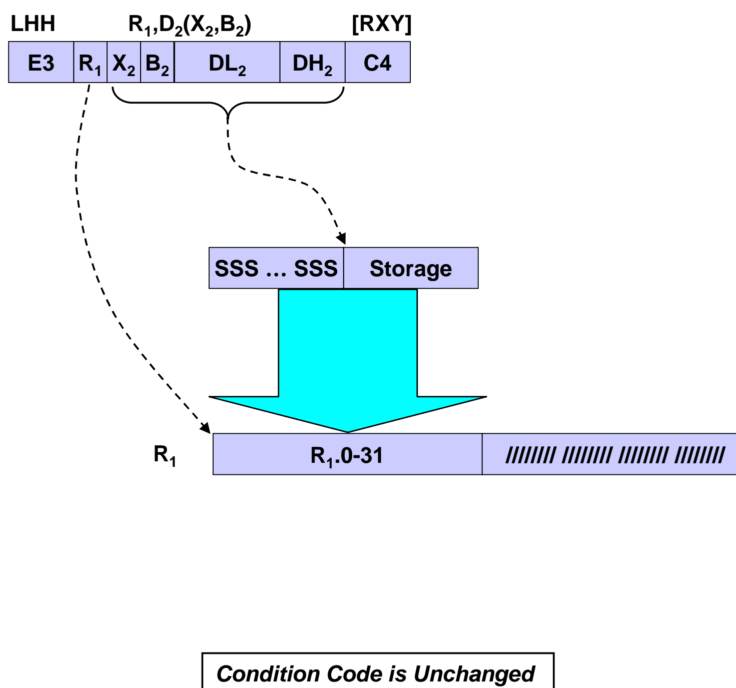
LOAD BYTE HIGH (LBH)



LOAD BYTE HIGH (LBH) is the analog to the LOAD BYTE (LB), except that the results are placed in the leftmost bits of the first-operand register. (LOAD BYTE (LB) was introduced with the long-displacement facility in the z990.)

The byte in storage designated by the second-operand location is sign extended on the left and the result is placed in bits 0-31 of the general register designated by the R_1 field of the instruction.

LOAD HALFWORD HIGH (LHH)

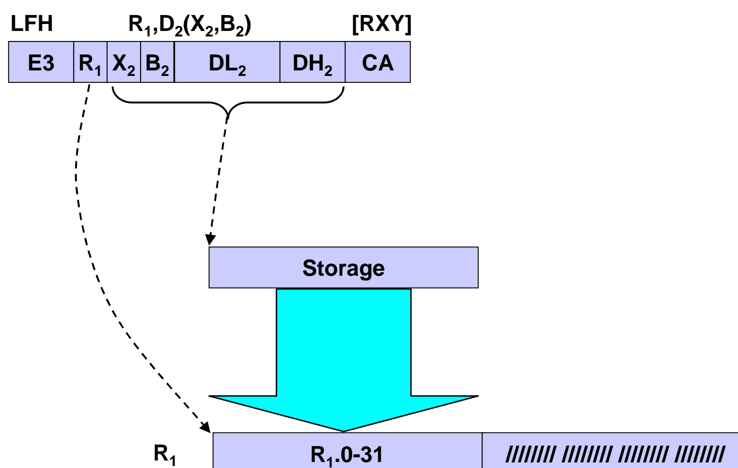


Condition Code is Unchanged

LOAD HALFWORD HIGH (LHH) is the analog to the LOAD HALFWORD (LH), except that the results are placed in the leftmost bits of the first-operand register.

The two-byte field in storage designated by the second-operand location is sign extended on the left and the result is placed in bits 0-31 of the general register designated by the R_1 field of the instruction.

LOAD HIGH (LFH)

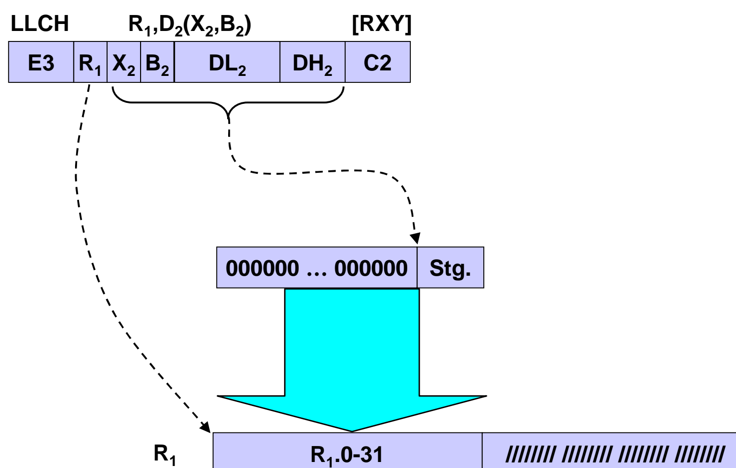


Condition Code is Unchanged

LOAD HIGH (LFH) is the analog to the LOAD (L), except that the results are placed in the leftmost bits of the first-operand register.

The four-byte field in storage designated by the second-operand location is placed in bits 0-31 of the general register designated by the R_1 field of the instruction.

LOAD LOGICAL CHARACTER HIGH (LLCH)

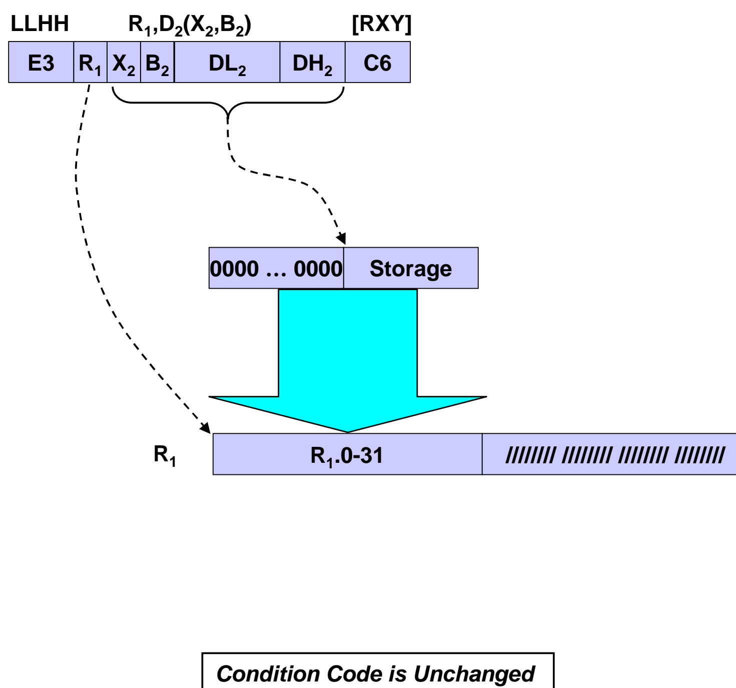


Condition Code is Unchanged

LOAD LOCICAL CHARACTER HIGH (LLCH) is the analog to the LOAD LOGICAL CHARACTER (LLC), except that the results are placed in the leftmost bits of the first-operand register. (LOAD LOGICAL CHARACTER (LLC) was introduced with the extended-immediate facility in the z9-109.)

The byte in storage designated by the second-operand location is zero extended on the left and the result is placed in bits 0-31 of the general register designated by the R_1 field of the instruction.

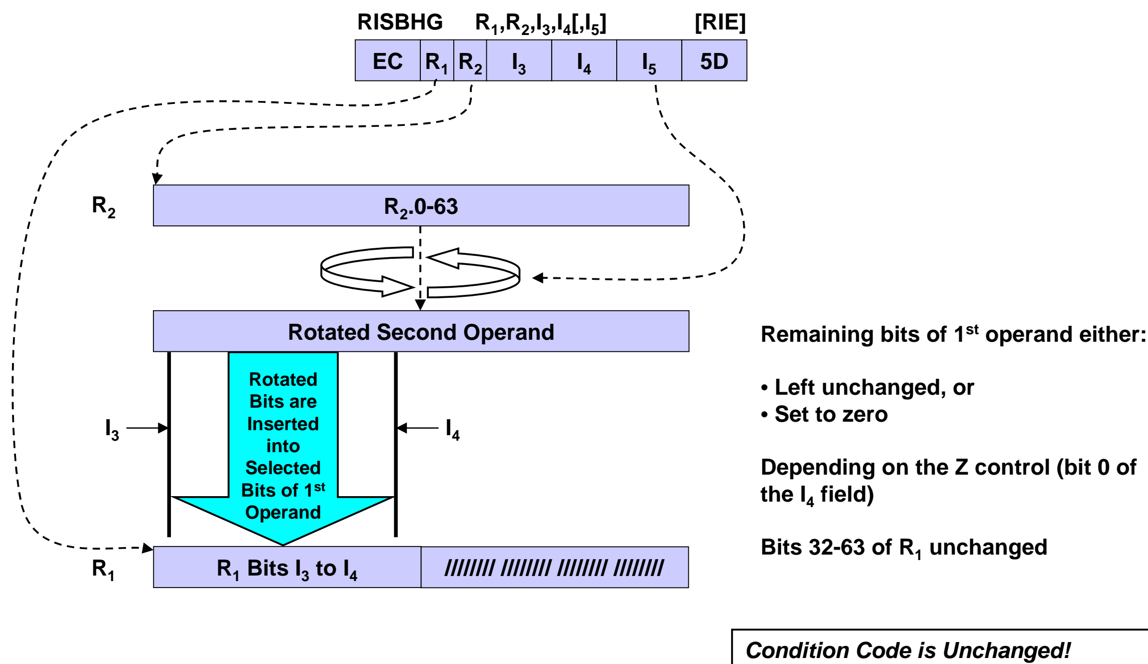
LOAD LOGICAL HALFWORD HIGH (LLHH)



LOAD LOCICAL HALFWORD HIGH (LLHH) is the analog to the LOAD LOGICAL HALFWORD (LLH), except that the results are placed in the leftmost bits of the first-operand register. (LOAD LOGICAL HALFWORD (LLH) was introduced with the extended-immediate facility in the z9-109.)

The two bytes in storage designated by the second-operand location are zero extended on the left and the result is placed in bits 0-31 of the general register designated by the R_1 field of the instruction.

ROTATE THEN INSERT SELECTED BITS HIGH (RISBHG)



ROTATE THEN INSERT SELECTED BITS HIGH (RISBHG) is the analog to ROTATE THEN INSERT SELECTED BITS (RISBG), except that the results of RISBHG are limited to the leftmost bits of general register R_1. Note ROTATE THEN INSERT SELECTED BITS (RISBG) was introduced with the general-instructions enhancement facility on the System z10.

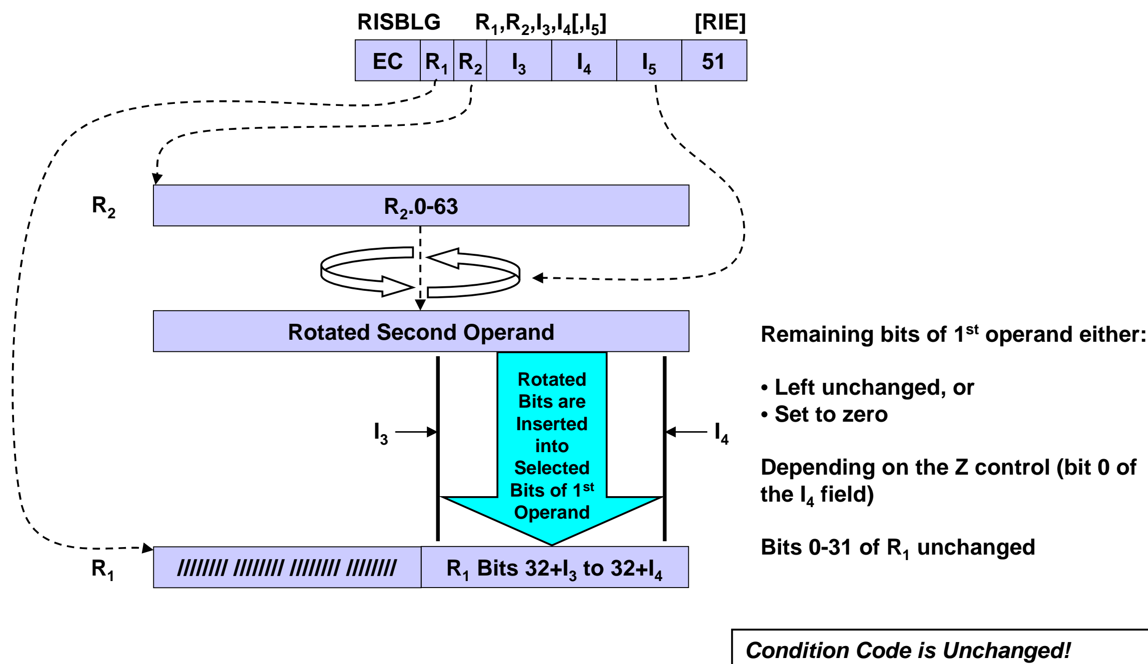
All 64 bits of the second operand are rotated to the left by the number of bits specified in the fifth operand (note, if the fifth operand is coded as a negative value, the rotation appears to occur to the right).

The I_3 and I_4 fields of the instruction are used to specify a starting and ending bit position in the result register (that is, the general register designated by the R_1 field of the instruction). The selected bits of the rotated second operand are inserted into the corresponding bits of the result register.

The remaining bits of the leftmost 32 bits of the result register are either left unchanged or set to zeros, depending on whether the zero-remaining-bits control (bit 0 of the I_3 field of the instruction) is zero or one, respectively.

Unless the R_1 and R_2 fields designate the same register, the general register designated by the R_2 field of the instruction remains unchanged. The rightmost 32 bits of the general register designated by the R_1 field always remain unchanged.

ROTATE THEN INSERT SELECTED BITS LOW (RISBLG)



ROTATE THEN INSERT SELECTED BITS LOW (RISBLG) is the analog to ROTATE THEN INSERT SELECTED BITS (RISBG), except that the results of RISBLG are limited to the rightmost bits of general register R_1. Note ROTATE THEN INSERT SELECTED BITS (RISBG) was introduced with the general-instructions enhancement facility on the System z10.

All 64 bits of the second operand are rotated to the left by the number of bits specified in the fifth operand (note, if the fifth operand is coded as a negative value, the rotation appears to occur to the right).

The I_3 and I_4 fields of the instruction are used to specify a starting and ending bit position in the rightmost 32 bits of the result register (that is, the general register designated by the R_1 field of the instruction). Although the values of the I_3 and I_4 fields are each encoded in a range of 0-31, the effective bit positions in the 64-bit register are 32 bits higher. The selected (rightmost 32) bits of the rotated second operand are inserted into the corresponding (rightmost 32) bits of the result register.

The remaining bits of the rightmost 32 bits of the result register are either left unchanged or set to zeros, depending on whether the zero-remaining-bits control (bit 0 of the I_3 field of the instruction) is zero or one, respectively.

Unless the R_1 and R_2 fields designate the same register, the general register designated by the R_2 field of the instruction remains unchanged. The leftmost 32 bits of the general register designated by the R_1 field always remain unchanged.

ROTATE THEN INSERT SELECTED BITS HIGH/LOW

- **Bits 2-7 of I_5 field are rotate amount**
  - R_2 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
- **Starting and ending bit positions of selected bits specified in bits 3-7 of the I_3 and I_4 fields, respectively**
  - For RISBHG, I_3 and I_4 fields are appended on the left with a binary zero (0-31)
  - For RISBLG, I_3 and I_4 fields are appended on the left with a binary one (32-63)
  - 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 mnemonics: RISBHGZ, RISBLGZ
- **Condition code remains unchanged (different from RISBG)**

Note that for RISBHG, the I_3 and I_4 fields directly designate bits 0-31 of the result register. For RISBLG, a binary one is implicitly appended to the left of the values coded in the I_3 and I_4 fields, thus the effective bit positions in the result register are 32-63.

Unlike ROTATE THEN INSERT SELECTED BITS (RISBG), the ROTATE THEN INSERT SELECTED BITS HIGH / LOW instructions do not set the condition code. This allows the instructions to be used to implement pseudo-instructions (see the next slide).

ROTATE THEN INSERT SELECTED BITS HIGH/LOW: Extended Mnemonics

Instruction Name	Extended Mnemonic	RISBHG / RISBLG Equiv.
LOAD (HIGH←HIGH)	LHHR R_1,R_2	RISBHGZ R_1,R_2,0,31
LOAD (HIGH←LOW)	LHLR R_1,R_2	RISBHGZ R_1,R_2,0,31,32
LOAD (LOW←HIGH)	LLHFR R_1,R_2	RISBLGZ R_1,R_2,0,31,32
LOAD LOGICAL HALFWORD (HIGH←HIGH)	LLHHHR R_1,R_2	RISBHGZ R_1,R_2,16,31
LOAD LOGICAL HALFWORD (HIGH←LOW)	LLHHLR R_1,R_2	RISBHGZ R_1,R_2,16,31,32
LOAD LOGICAL HALFWORD (LOW←HIGH)	LLHLHR R_1,R_2	RISBLGZ R_1,R_2,16,31,32
LOAD LOGICAL CHARACTER (HIGH←HIGH)	LLCHHR R_1,R_2	RISBHGZ R_1,R_2,24,31
LOAD LOGICAL CHARACTER (HIGH←LOW)	LLCHLR R_1,R_2	RISBHGZ R_1,R_2,24,31,32
LOAD LOGICAL CHARACTER (LOW←HIGH)	LLCLHR R_1,R_2	RISBLGZ R_1,R_2,24,31,32

With ROTATE THEN INSERT SELECTED BITS HIGH and ROTATE THEN INSERT SELECTED BITS LOW, a large group of other pseudo-instructions can be implemented, as illustrated on this slide.

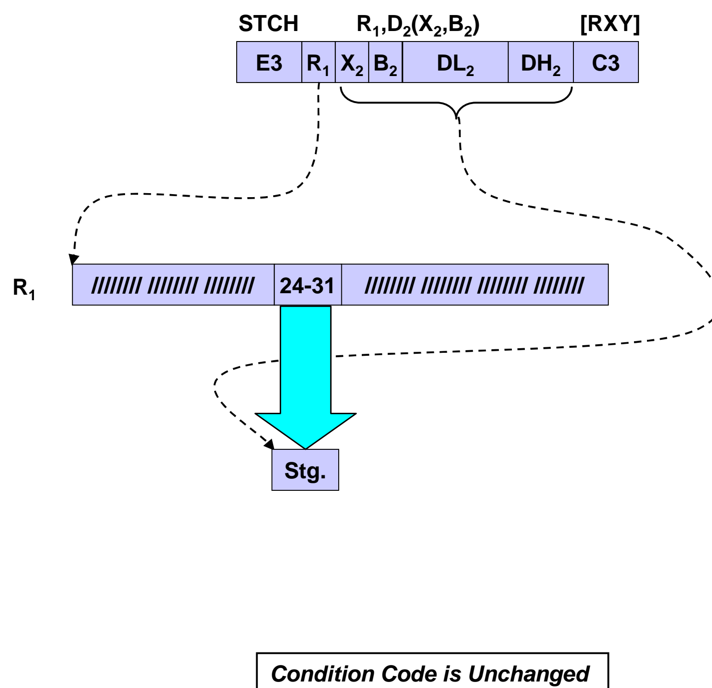
The High-Level Assembler provides extended mnemonics that implement these pseudo-instructions, even though they are actually implemented with RISBHG and RISBLG.

ROTATE THEN * SELECTED BITS: Extended Mnemonics

Instruction Name	Extended Mnemonic		R*SBG Equivalent	
AND HIGH (HIGH←HIGH)	NHHR	R_1,R_2	RNSBG	R_1,R_2,0,31
AND HIGH (HIGH←LOW)	NHLR	R_1,R_2	RNSBG	R_1,R_2,0,31,32
AND HIGH (LOW←HIGH)	NLHR	R_1,R_2	RNSBG	R_1,R_2,32,63,32
EXCLUSIVE OR (HIGH←HIGH)	XHHR	R_1,R_2	RXSBG	R_1,R_2,0,31
EXCLUSIVE OR (HIGH←LOW)	XHLR	R_1,R_2	RXSBG	R_1,R_2,0,31,32
EXCLUSIVE OR (LOW←HIGH)	XLHR	R_1,R_2	RXSBG	R_1,R_2,32,63,32
OR (HIGH←HIGH)	OHHR	R_1,R_2	ROSBG	R_1,R_2,0,31
OR (HIGH←LOW)	OHLR	R_1,R_2	ROSBG	R_1,R_2,0,31,32
OR (LOW←HIGH)	OLHR	R_1,R_2	ROSBG	R_1,R_2,32,63,32

The High-Level Assembler also provides pseudo-instructions to perform high-word logical operations by using the ROTATE THEN AND SELECTED BITS, ROTATE THEN OR SELECTED BITS, and ROTATE THEN EXCLUSIVE OR SELECTED BITS instructions (RNSBG, ROSBG, and RXSBG were introduced with the System z10).

STORE CHARACTER HIGH (STCH)

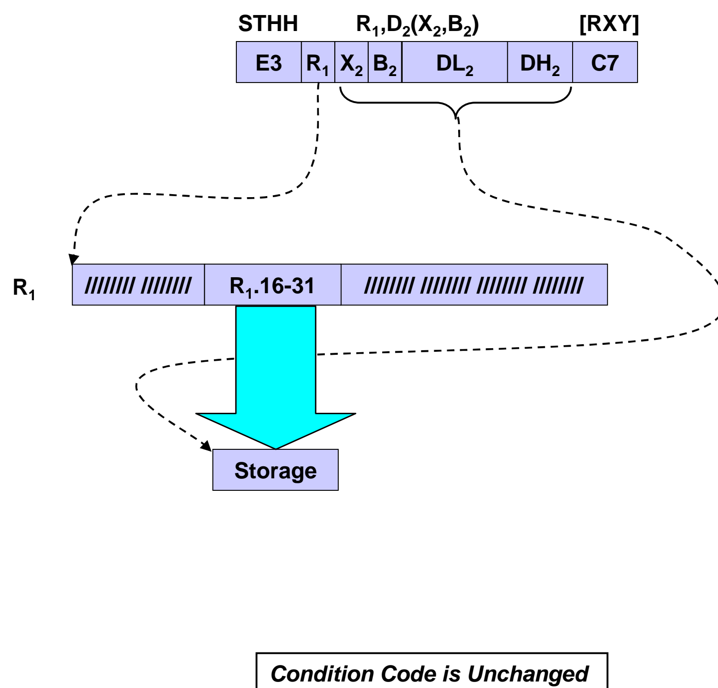


Condition Code is Unchanged

STORE CHARACTER HIGH (STCH) is the analog to STORE CHARACTER (STC), except that the byte stored is in bits 24-31 of the general register designated by the R_1 field of the instruction.

Bits 24-31 of general register R_1 are placed into the byte in storage designated by the second-operand location.

STORE HALFWORD HIGH (STHH)

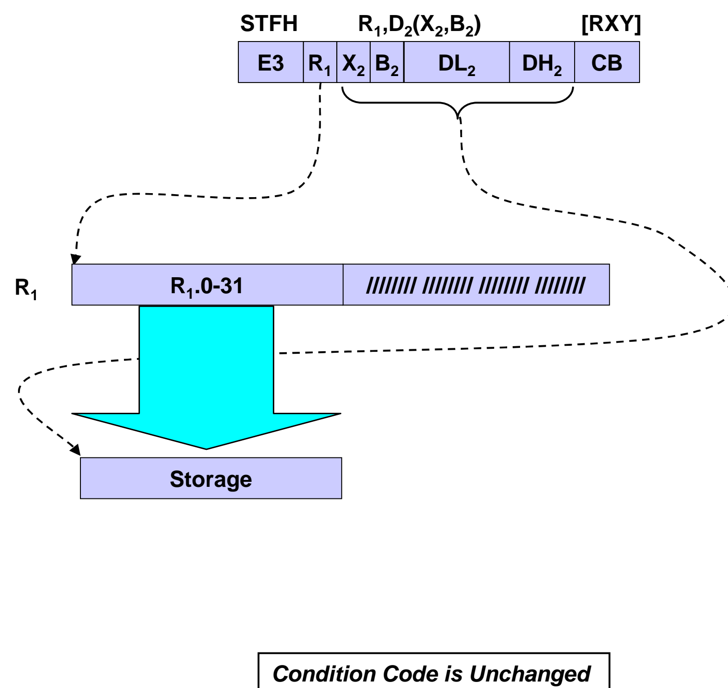


Condition Code is Unchanged

STORE HALFWORD HIGH (STHH) is the analog to STORE HALFWORD (STH), except that the two bytes stored are in bits 16-31 of the general register designated by the R_1 field of the instruction.

Bits 16-31 of general register R_1 are placed into the two bytes in storage designated by the second-operand location.

STORE HIGH (STFH)

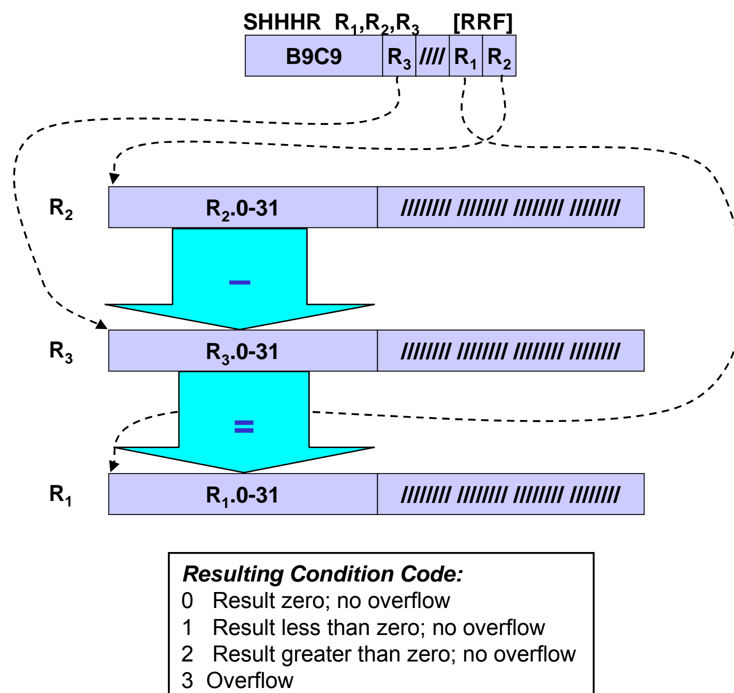


Condition Code is Unchanged

STORE HIGH (STFH) is the analog to STORE (ST), except that the four bytes stored are in bits 0-31 of the general register designated by the R_1 field of the instruction.

Bits 0-31 of general register R_1 are placed into the four bytes in storage designated by the second-operand location.

SUBTRACT HIGH (SHHHR)



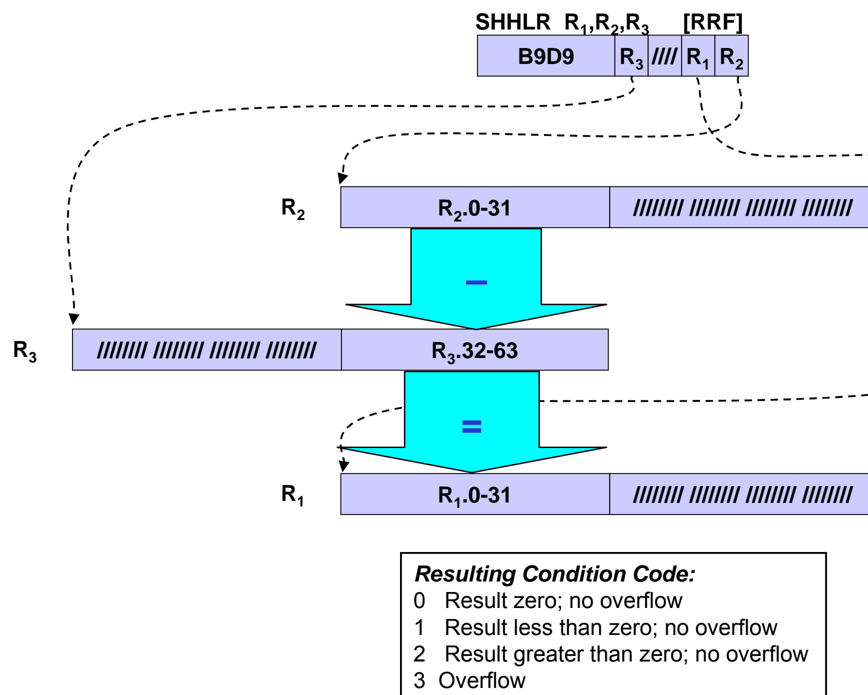
Resulting Condition Code:
0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

For SUBTRACT HIGH (SHHHR), the contents of the leftmost bits (0-31) of the general register designated by the R_3 field of the instruction are arithmetically subtracted from the contents of the leftmost bits of the general register designated by the R_2 field of the instruction. The difference replaces the leftmost bits of the general register designated by the R_1 field of the instruction; bits 32-63 of the result register remain unchanged.

The subtraction proceeds exactly as for SUBTRACT (SR), except that there are two source operands and a separate target operand – and, obviously, the result ends up in the leftmost 32 bits of the register.

The condition code is set as with any other signed subtraction operation.

SUBTRACT HIGH (SHHLR)

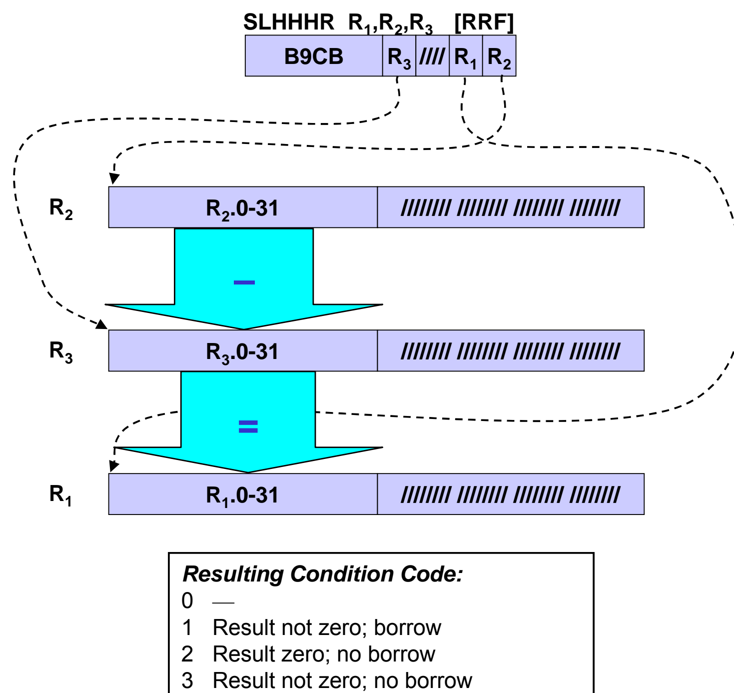


SUBTRACT HIGH (SHHLR) should perhaps be called SUBTRACT LOW FROM HIGH.

The contents of the rightmost bits (32-63) of the general register designated by the R_3 field of the instruction are arithmetically subtracted from the contents of the leftmost bits (0-31) of the general register designated by the R_2 field of the instruction. The difference replaces the leftmost bits of the general register designated by the R_1 operand; bits 32-63 of the result register remain unchanged.

The condition code is set as with any other signed addition operation.

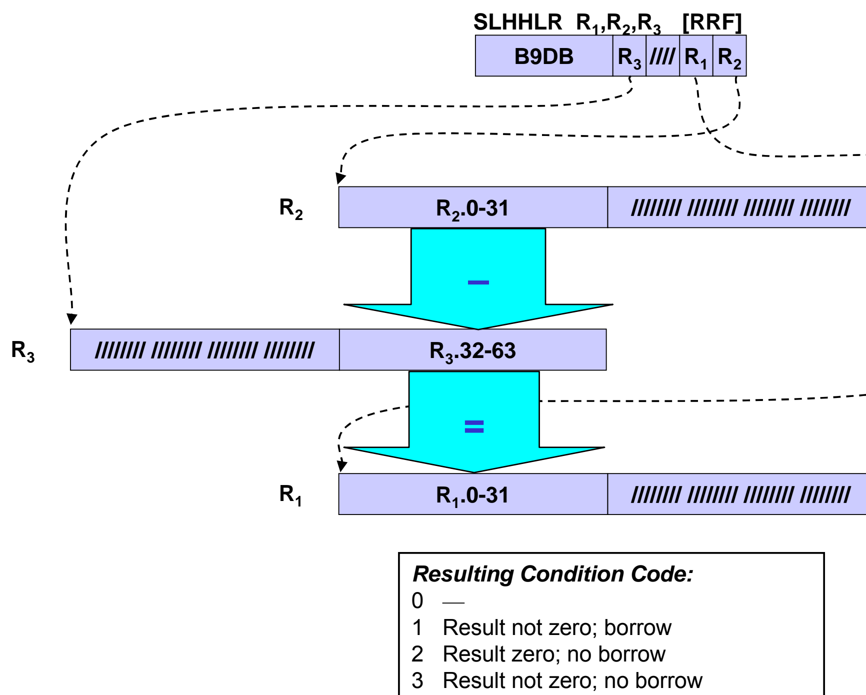
SUBTRACT LOGICAL HIGH (SLHHHR)



For SUBTRACT LOGICAL HIGH (SLHHHR), the contents of the leftmost bits (0-31) of the general register designated by the R_3 field of the instruction are logically subtracted from the contents of the leftmost bits of the general register designated by the R_2 field of the instruction. The difference replaces the leftmost bits of the general register designated by the R_1 field of the instruction; bits 32-63 of the result register remain unchanged.

The subtraction proceeds exactly as for SUBTRACT LOGICAL (SLR), except that there are two source operands and a separate target operand – and, obviously, the result ends up in the leftmost 32 bits of the register.

The condition code is set as with any other unsigned addition operation.



As with SUBTRACT HIGH (SHHLR), SUBTRCT LOGICAL HIGH (SLHHLR) should perhaps be called SUBTRACT LOGICAL LOW FROM HIGH.

The contents of the rightmost bits (32-63) of the general register designated by the R_3 field of the instruction are logically subtracted from the contents of the leftmost bits (0-31) of the general register designated by the R_2 field of the instruction. The difference replaces the leftmost bits of the general register designated by the R_1 operand; bits 32-63 of the result register remain unchanged.

The condition code is set as with any other unsigned addition operation.

Interlocked-Access Facility (1)

- Suite of instructions to perform interlocked-update operations on various storage operands
  - LOAD AND ADD
  - LOAD AND ADD LOGICAL
  - LOAD AND AND
  - LOAD AND EXCLUSIVE OR
  - LOAD AND OR
  - LOAD PAIR DISJOINT
- Changes to existing instructions to provide interlocked update when operands are aligned on an integral boundary
  - ADD IMMEDIATE (ASI, AGSI)
  - ADD LOGICAL WITH SIGNED IMMEDIATE (ALSI, ALGSI)
- Installation of the interlocked-access facility (& al.) indicated by facility bit 45

The interlocked-access facility provides instructions that are designed to facilitate multiprogramming; most of the instructions access memory in a block-concurrent, interlocked-update fashion (more details on the next slides).

Also, when the interlocked-access facility is installed, the ADD IMMEDIATE (ASI and AGSI) and ADD LOGICAL WITH SIGNED IMMEDIATE (ALSI and ALGSI) perform their storage accesses using block-concurrent, interlocked update when the storage operand is aligned on an integral boundary. Thus, as observed by other CPUs and the channel subsystem, the fetch, addition, and store of the result appear to occur atomically … there is no need for a COMPARE AND SWAP loop to perform these operations!

Interlocked-Access Facility (2):

Instruction	Mnemonic	OpCode	1st Operand	2nd Operand	3rd Operand
LOAD AND ADD	LAA	EBF8	R_1.32-63	S12 [32 bits]	R_3.32-63
LOAD AND ADD	LAAG	EBE8	R_1.0-63	S12 [64 bits]	R_3.0-63
LOAD AND ADD LOGICAL	LAAL	EBFA	R_1.32-63	S12 [32 bits]	R_3.32-63
LOAD AND ADD LOGICAL	LAALG	EBEA	R_1.0-63	S12 [64 bits]	R_3.0-63
LOAD AND AND	LAN	EBF4	R_1.32-63	S12 [32 bits]	R_3.32-63
LOAD AND AND	LANG	EBE4	R_1.0-63	S12 [64 bits]	R_3.0-63
LOAD AND EXCLUSIVE OR	LAX	EBF7	R_1.32-63	S12 [32 bits]	R_3.32-63
LOAD AND EXCLUSIVE OR	LAXG	EBE7	R_1.0-63	S12 [64 bits]	R_3.0-63
LOAD AND OR	LAO	EBF6	R_1.32-63	S12 [32 bits]	R_3.32-63
LOAD AND OR	LAOG	EBE6	R_1.0-63	S12 [64 bits]	R_3.0-63
LOAD PAIR DISJOINT	LPD	C84	S12 [32 bits]	S12 [32 bits]	R_3.32-63 R_3+1.32-63
LOAD PAIR DISJOINT	LPDG	C85	S12 [32 bits]	S12 [32 bits]	R_3.0-63 R_3+1.0-63

Explanation:

R_n Register operand ‘n’

S12 Storage operand designated by 12-bit unsigned displacement

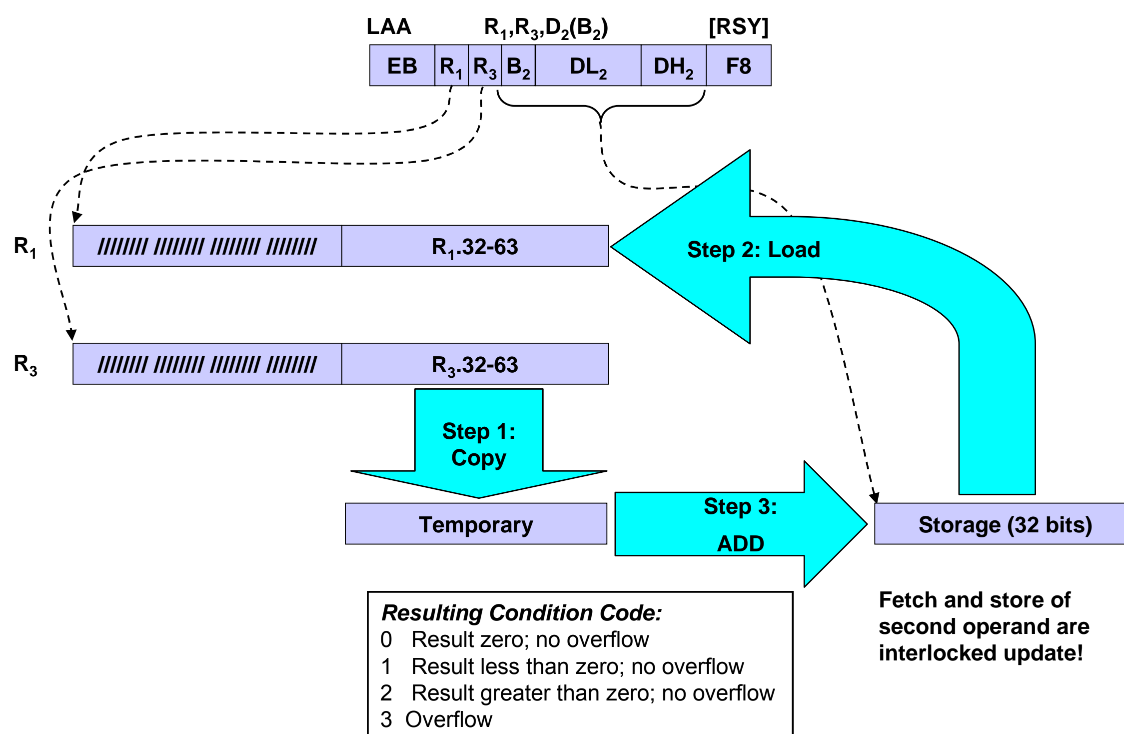
The interlocked-access facility comprises two types of arithmetic operations (signed addition and unsigned addition), and three types of logical operations (AND, OR and XOR). For each of these operations. For each of these five operations, the instruction performs the following:

1. The second-operand storage location is fetched.
2. An operation is performed using the contents of the third-operand register and the storage location, with the result being placed into the storage location. The access of the storage location (beginning with the fetch in step 1, through the store in this step) is performed as a block-concurrent, interlocked update (that is, it’s atomic).
3. The original second-operand value (prior to any modification in step 2) is placed in the first-operand register.

The illustrative sequence of the operation shown on the following slides differs somewhat from that described here, however the result is the same.

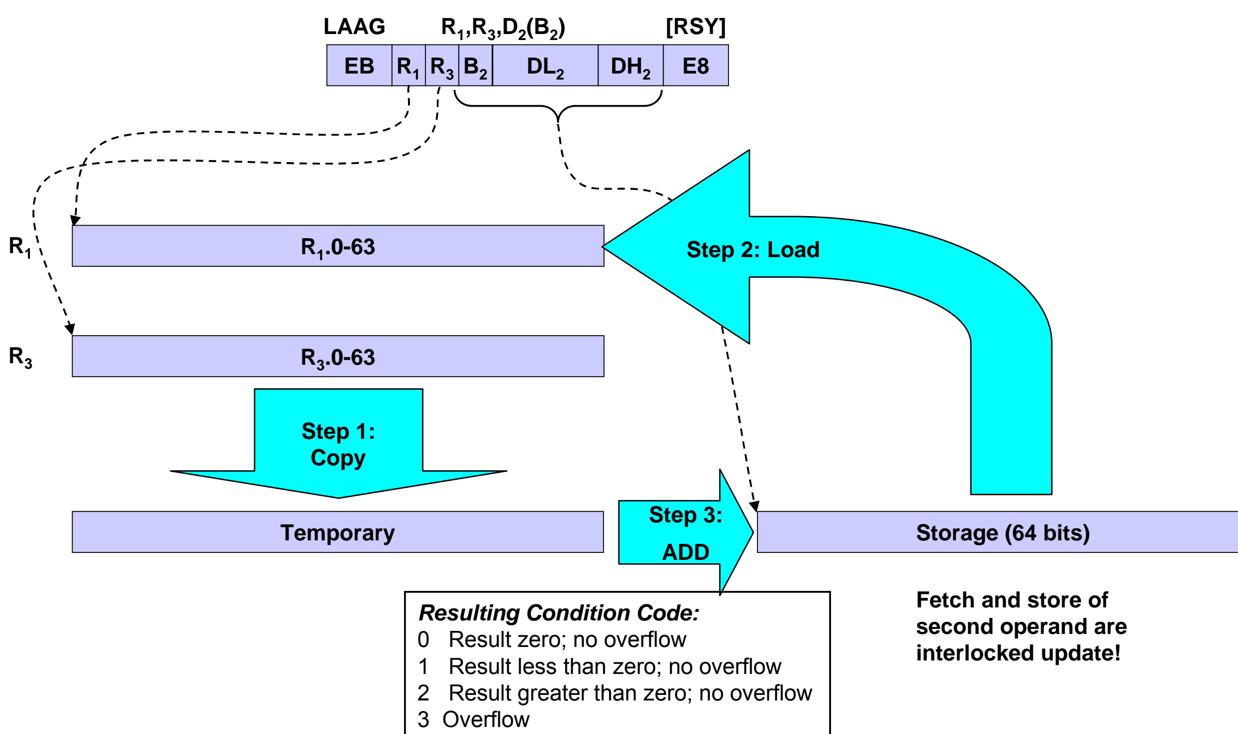
The facility also includes an operation to access two discrete storage locations, providing an indication as to whether any other CPU altered one of the locations during the fetch.

For each of these operations, there is a 32-bit and a 64-bit version of the instruction.



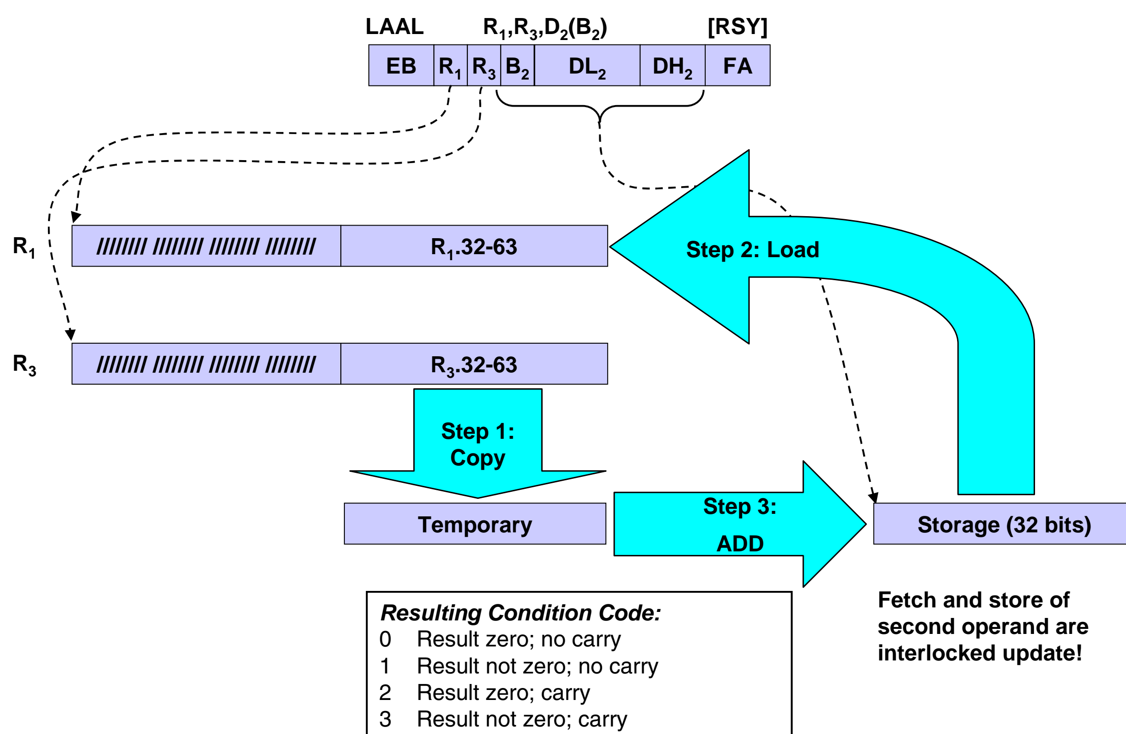
For LOAD AND ADD (LAA), the contents of bits 32-63 of the general register designated by the R_3 field of the instruction are preserved in a temporary location in the CPU. Then the word in storage designated by the second-operand location is fetched into bits 32-63 of the general register designated by the R_1 field of the instruction. Finally, the temporary 32-bit value is added to the contents of the word in storage, and the result replaces the word in storage. As observed by other CPUs and the channel subsystem, the fetching and storing of the word in storage appear to occur as a block-concurrent interlocked update.

Alternatively, the word in storage may be fetched into a temporary location, the addition of that word and general register R_3 occurs, and then the temporary value place in general register R_1. Regardless of method, the fetching into a temporary location ensures that the result in general register R_1 is the original contents of the storage location (prior to the addition).

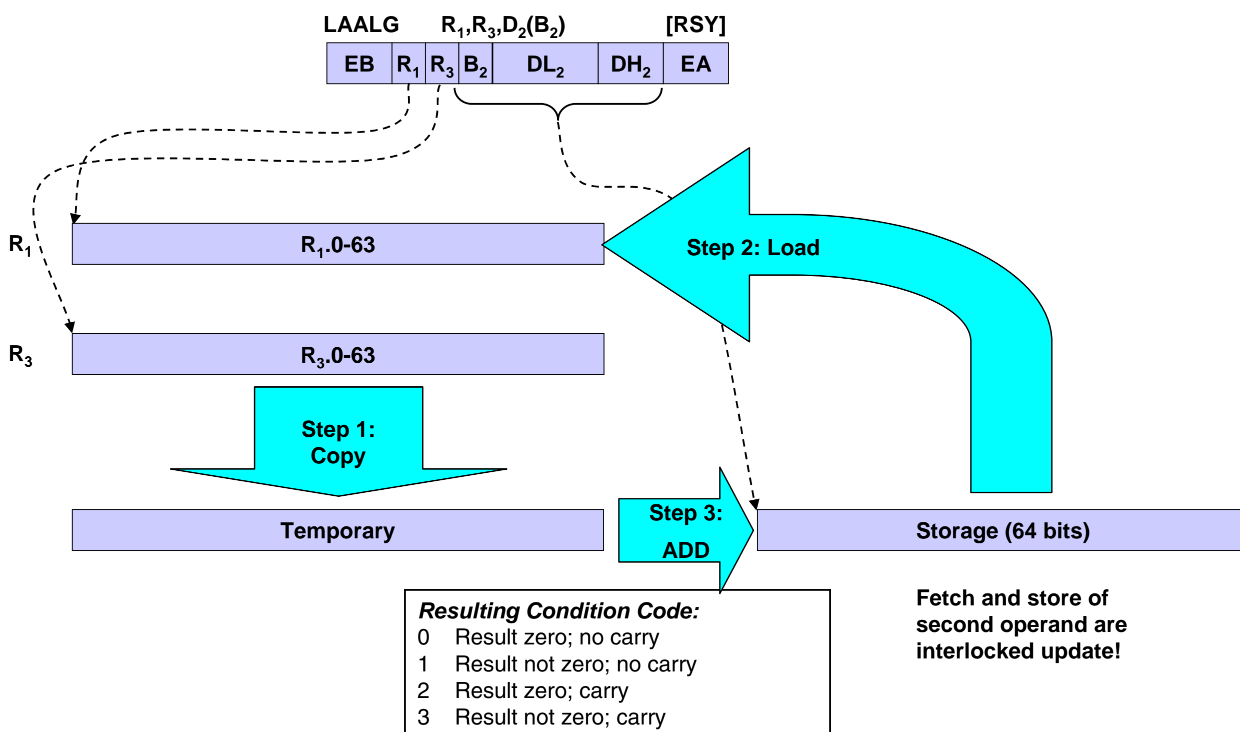


For LOAD AND ADD (LAAG), the contents of bits 0-63 of the general register designated by the R_3 field of the instruction are preserved in a temporary location in the CPU. Then the doubleword in storage designated by the second-operand location is fetched into bits 0-63 of the general register designated by the R_1 field of the instruction. Finally, the temporary 64-bit value is added to the contents of the doubleword in storage, and the result replaces the doubleword in storage. As observed by other CPUs and the channel subsystem, the fetching and storing of the doubleword in storage appear to occur as a block-concurrent interlocked update.

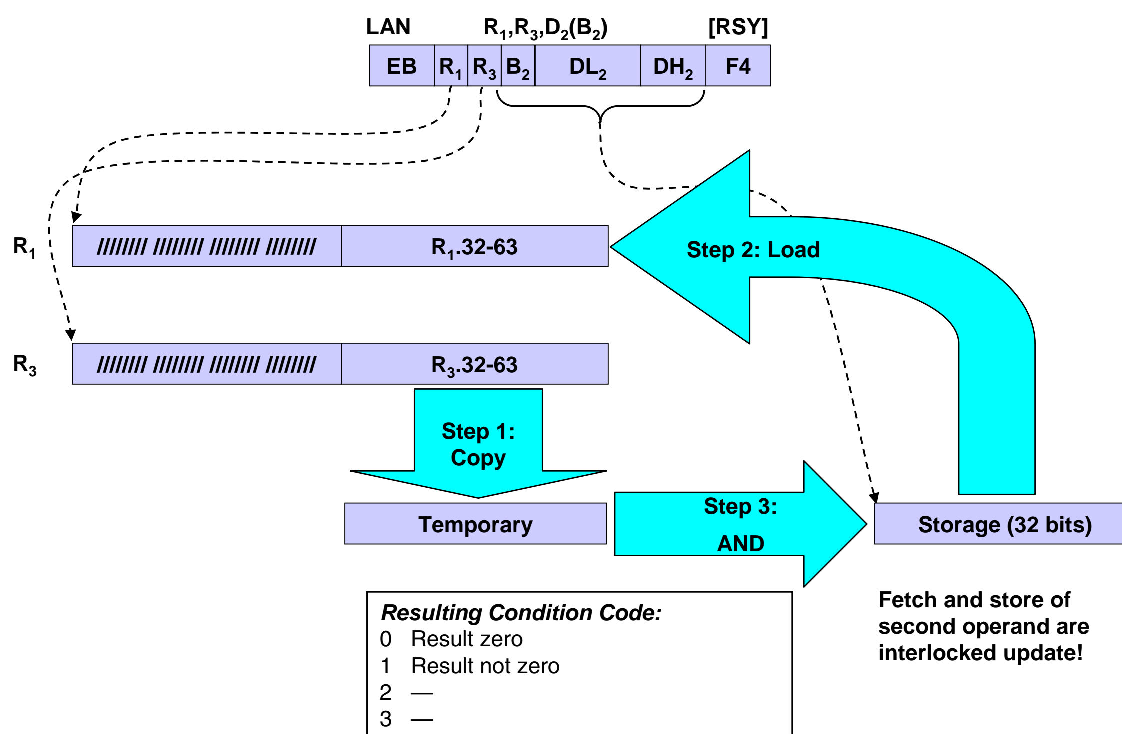
See the description of LOAD AND ADD (LAA) for an explanation of the temporary buffering of the doubleword in the CPU.



The operation of LOAD AND ADD LOGICAL (LAAL) is identical to that of LOAD AND ADD (LAA), except for the setting of the condition code. LAAL sets the condition code consistent with other unsigned additions.

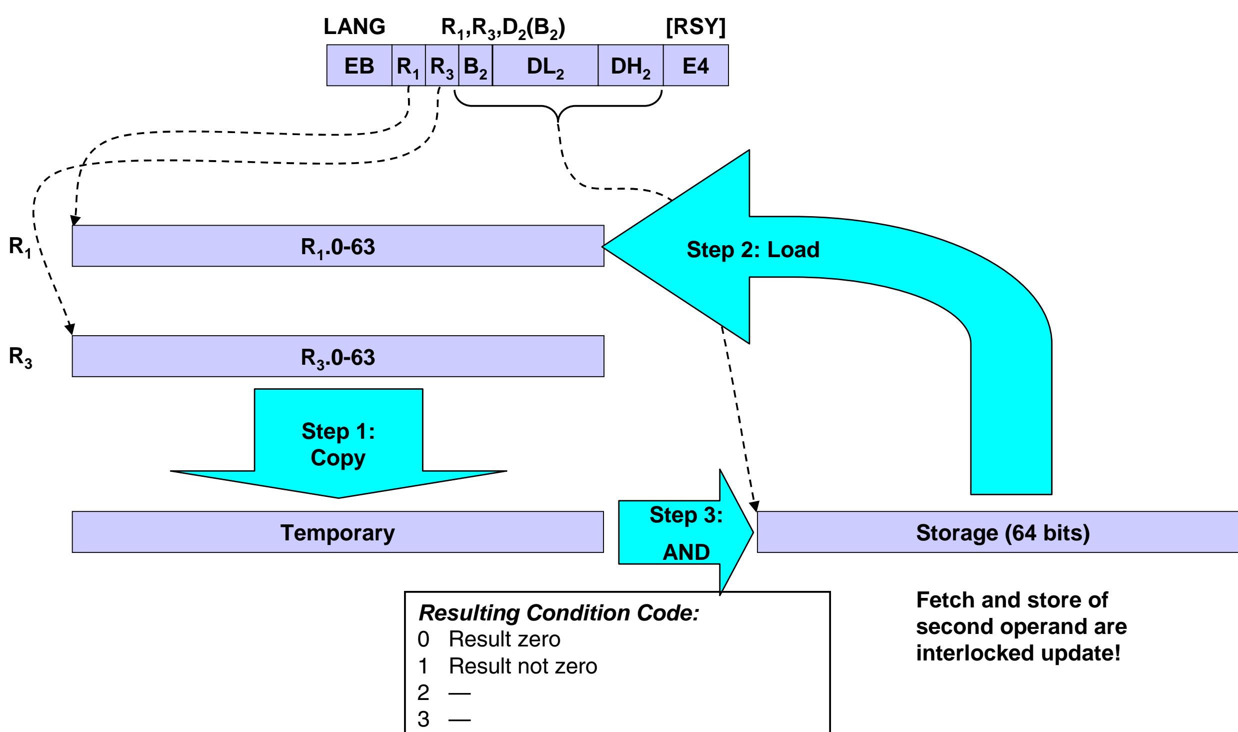


The operation of LOAD AND ADD LOGICAL (LAALG) is identical to that of LOAD AND ADD (LAAG), except for the setting of the condition code. LAALG sets the condition code consistent with other unsigned additions.



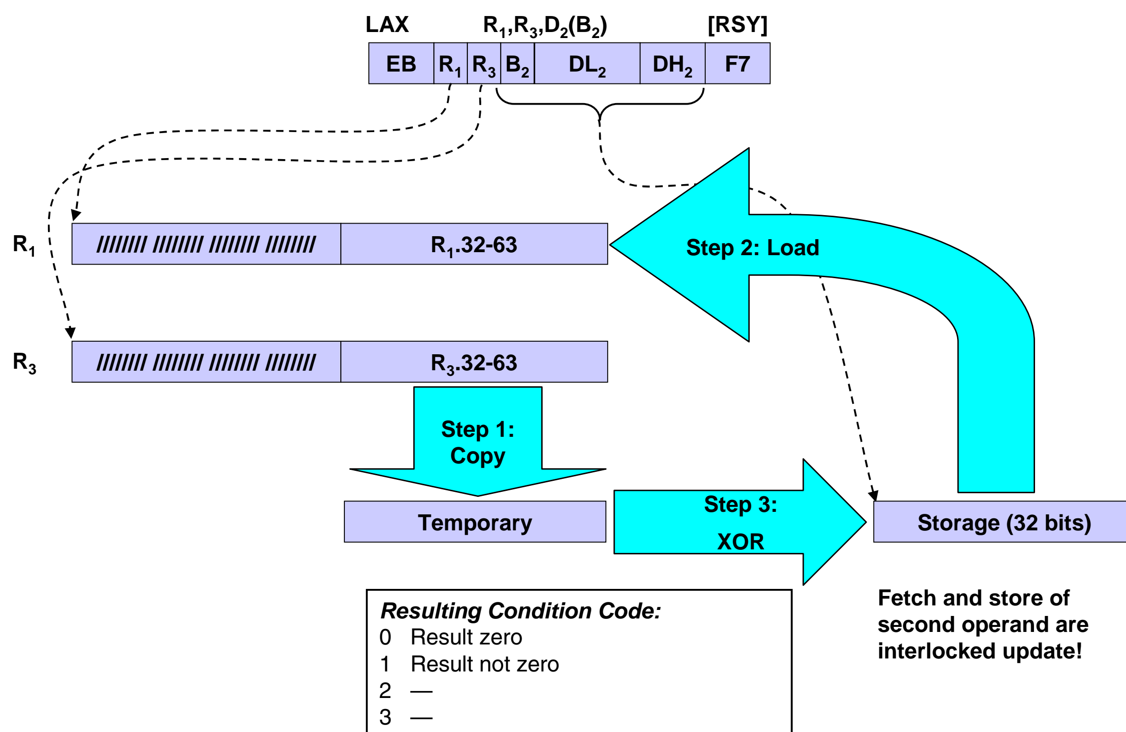
For LOAD AND AND (LAN), the contents of bits 32-63 of the general register designated by the R_3 field of the instruction are preserved in a temporary location in the CPU. Then the word in storage designated by the second-operand location is fetched into bits 32-63 of the general register designated by the R_1 field of the instruction. Finally, the temporary 32-bit value is logically ANDed with the contents of the word in storage, and the result replaces the word in storage. As observed by other CPUs and the channel subsystem, the fetching and storing of the word in storage appear to occur as a block-concurrent interlocked update.

See the description of LOAD AND ADD (LAA) for an explanation of the temporary buffering of the word in the CPU.



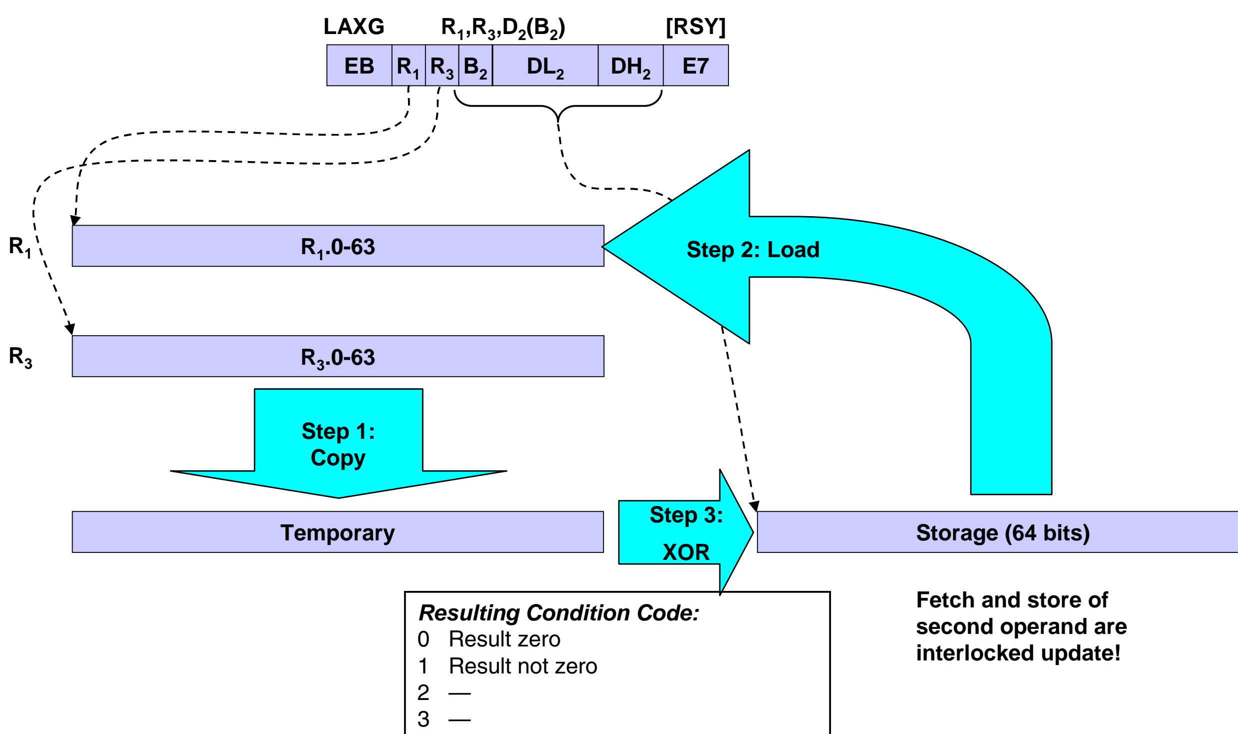
For LOAD AND AND (LANG), the contents of bits 0-63 of the general register designated by the R_3 field of the instruction are preserved in a temporary location in the CPU. Then the doubleword in storage designated by the second-operand location is fetched into bits 0-63 of the general register designated by the R_1 field of the instruction. Finally, the temporary 64-bit value is logically ANDed with the contents of the doubleword in storage, and the result replaces the doubleword in storage. As observed by other CPUs and the channel subsystem, the fetching and storing of the doubleword in storage appear to occur as a block-concurrent interlocked update.

See the description of LOAD AND ADD (LAA) for an explanation of the temporary buffering of the word in the CPU.



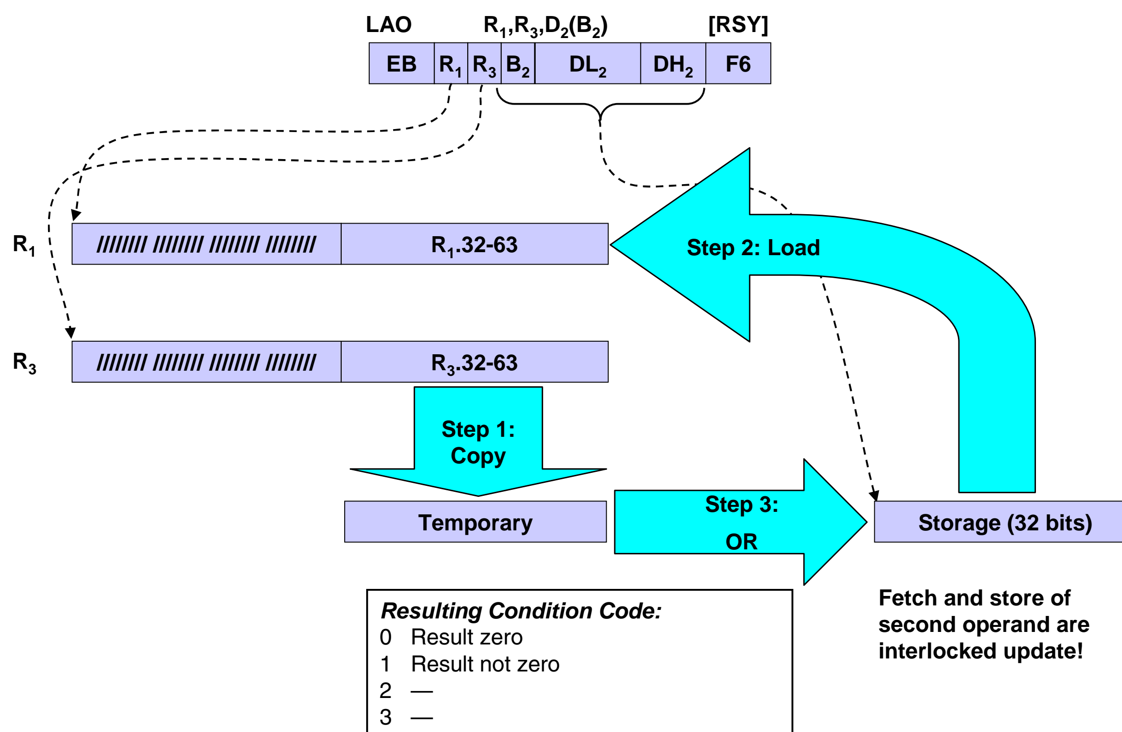
For LOAD AND EXCLUSIVE OR (LAX), the contents of bits 32-63 of the general register designated by the R_3 field of the instruction are preserved in a temporary location in the CPU. Then the word in storage designated by the second-operand location is fetched into bits 32-63 of the general register designated by the R_1 field of the instruction. Finally, the temporary 32-bit value is logically exclusive ORed with the contents of the word in storage, and the result replaces the word in storage. As observed by other CPUs and the channel subsystem, the fetching and storing of the word in storage appear to occur as a block-concurrent interlocked update.

See the description of LOAD AND ADD (LAA) for an explanation of the temporary buffering of the word in the CPU.



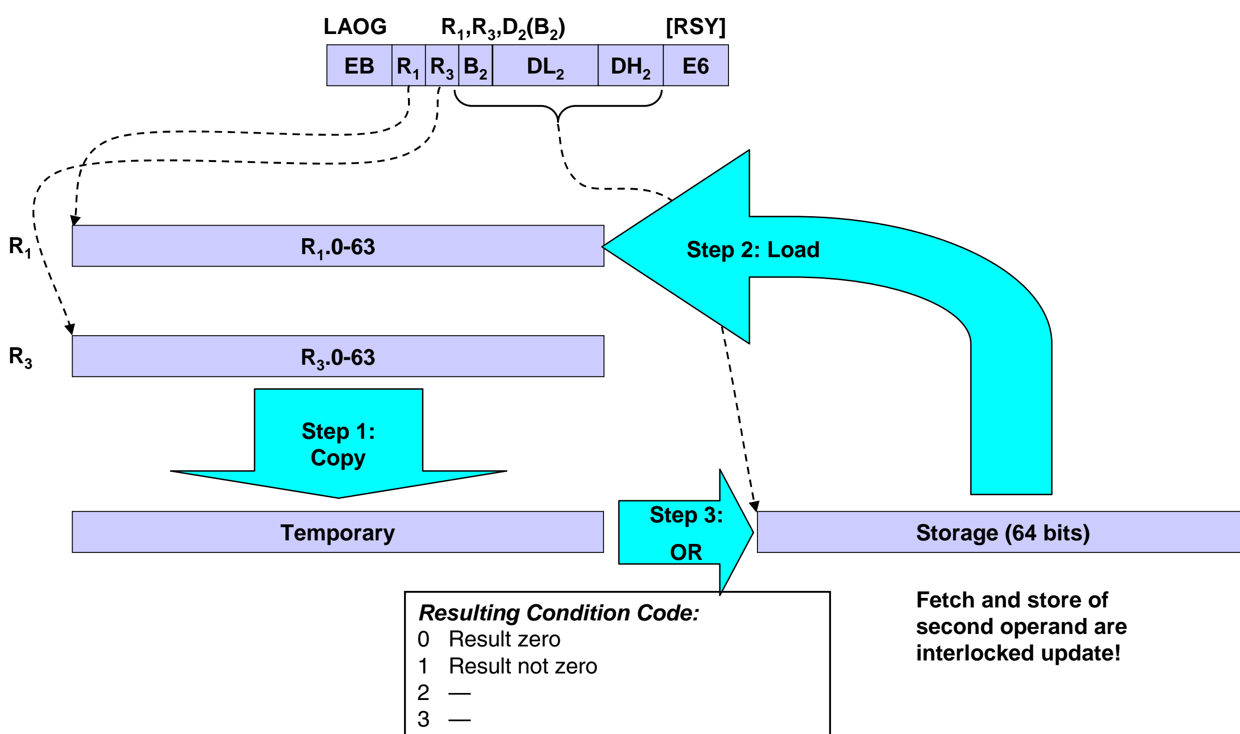
For LOAD AND EXCLUSIVE OR (LAXG), the contents of bits 0-63 of the general register designated by the R_3 field of the instruction are preserved in a temporary location in the CPU. Then the doubleword in storage designated by the second-operand location is fetched into bits 0-63 of the general register designated by the R_1 field of the instruction. Finally, the temporary 64-bit value is logically exclusive ORed with the contents of the doubleword in storage, and the result replaces the doubleword in storage. As observed by other CPUs and the channel subsystem, the fetching and storing of the doubleword in storage appear to occur as a block-concurrent interlocked update.

See the description of LOAD AND ADD (LAA) for an explanation of the temporary buffering of the word in the CPU.



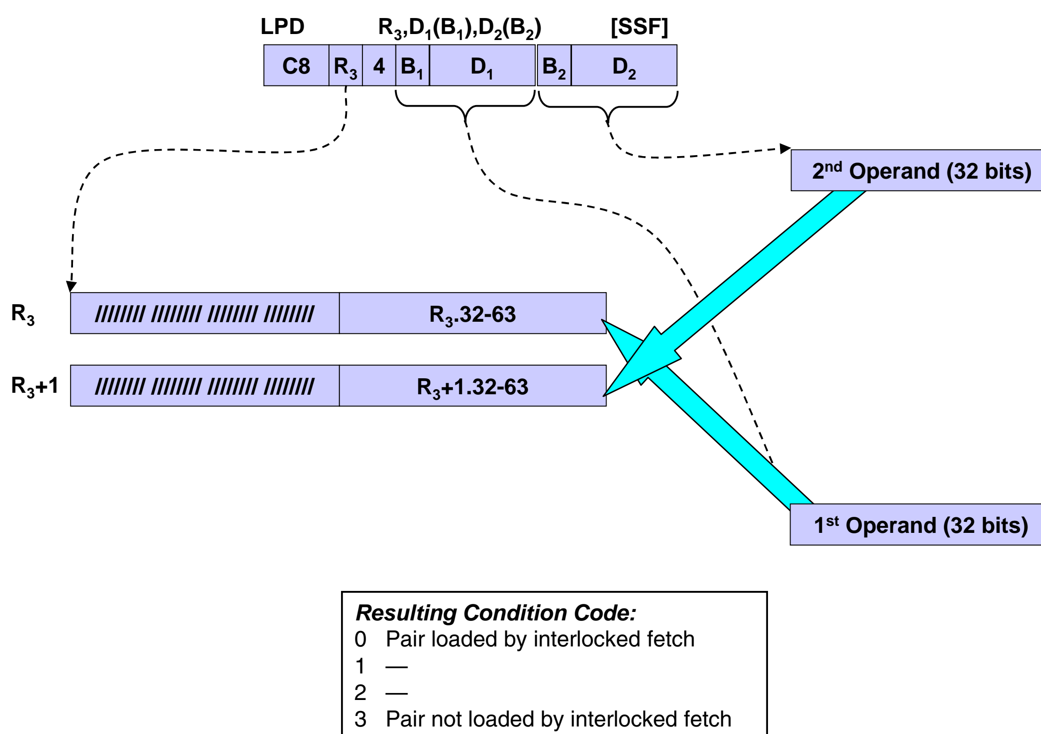
For LOAD AND OR (LAO), the contents of bits 32-63 of the general register designated by the R_3 field of the instruction are preserved in a temporary location in the CPU. Then the word in storage designated by the second-operand location is fetched into bits 32-63 of the general register designated by the R_1 field of the instruction. Finally, the temporary 32-bit value is logically ORed with the contents of the word in storage, and the result replaces the word in storage. As observed by other CPUs and the channel subsystem, the fetching and storing of the word in storage appear to occur as a block-concurrent interlocked update.

See the description of LOAD AND ADD (LAA) for an explanation of the temporary buffering of the word in the CPU.



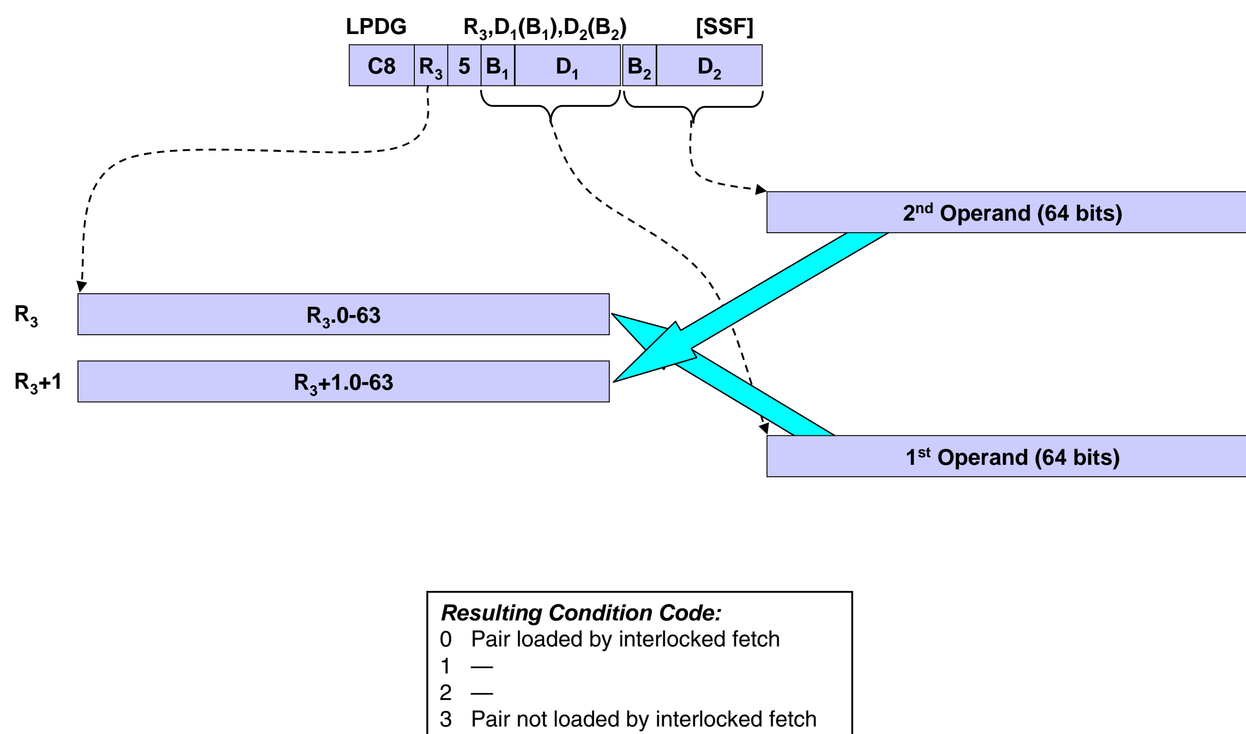
For LOAD AND OR (LAOG), the contents of bits 0-63 of the general register designated by the R_3 field of the instruction are preserved in a temporary location in the CPU. Then the doubleword in storage designated by the second-operand location is fetched into bits 0-63 of the general register designated by the R_1 field of the instruction. Finally, the temporary 64-bit value is logically ORed with the contents of the doubleword in storage, and the result replaces the doubleword in storage. As observed by other CPUs and the channel subsystem, the fetching and storing of the doubleword in storage appear to occur as a block-concurrent interlocked update.

See the description of LOAD AND ADD (LAA) for an explanation of the temporary buffering of the word in the CPU.



For LOAD PAIR DISJOINT (LPD), the first and second operands are two distinct words in storage. The first and second operands are fetched into bits 32-63 of the even-odd general register pair designated by the R_3 field of the instruction; the first operand is fetched into the even-numbered register, and the second operand is fetched into the odd-numbered register.

The condition code is set based on whether the pair of words were fetched without alteration by other CPUs or the channel subsystem. CC0 means that neither word was altered during the fetching; CC3 means that one of the words was altered.



For LOAD PAIR DISJOINT (LPDG), the first and second operands are two distinct doublewords in storage. The first and second operands are fetched into bits 0-63 of the even-odd general register pair designated by the R_3 field of the instruction; the first operand is fetched into the even-numbered register, and the second operand is fetched into the odd-numbered register.

The condition code is set based on whether the pair of doublewords were fetched without alteration by other CPUs or the channel subsystem. CC0 means that neither doubleword was altered during the fetching; CC3 means that one of the doublewords was altered.

Load/Store-on-Condition Facility

- Provides means by which load and store operations may be executed conditionally
- M_3 field determines action based on current PSW condition code
  - RSY format – M_3 field takes place of X_2 field
  - HLASM provides extended mnemonics to use in lieu of the M_3 field
    - E, L, H, NE, NH, NL
- Installation of the load/store-on-condition facility (& al.) indicated by facility bit 45

The load-and-store-on-condition facility provides a means of executing a load or store, subject to the control of the condition code. Therefore, no branch instruction(s) are necessary to select the various code paths that effect the loading or storing. Consider the following code fragment that implements a min() function for four storage parameters:

```
          LG     15,PARM1
          CG     15,PARM2
          JNL    A
          LG     15,PARM2
A         CG     15,PARM3
          JNL    B
          LG     15,PARM3
B         CG     15,PARM4
          JNL    C
          LG     15,PARM4
C         …
```

With the load-and-store-on-condition facility, equivalent function can be realized without all the branching instructions, as follows

```
          LG     15,PARM1
          CG     15,PARM2
          LOCG   15,PARM2,B'0100'      (or LOCGL 15,PARM2)
          CG     15,PARM3
          LOCG   15,PARM3,B'0100'      (or LOCGL 15,PARM3)
          CG     15,PARM4
          LOCG   15,PARM4,B'0100'      (or LOCGL 15,PARM4)
```

Load/Store-on-Condition Facility

Instruction	Mnemonic	OpCode	1st Operand	2nd Operand	3rd Operand
LOAD ON CONDITION	LOCR	B9F2	R_1.32-63	R_2.32-63	Condition Mask
LOAD ON CONDITION	LOCGR	B9E2	R_1.0-63	R_2.0-63	Condition Mask
LOAD ON CONDITION	LOC	EBF2	R_1.32-63	S20 [32 bits]	Condition Mask
LOAD ON CONDITION	LOCG	EBE2	R_1.0-63	S20 [64 bits]	Condition Mask
STORE ON CONDITION	STOC	EBF3	R_1.32-63	S20 [32 bits]	Condition Mask
STORE ON CONDITION	STOCG	EBE3	R_1.0-63	S20 [64 bits]	Condition Mask

Explanation:

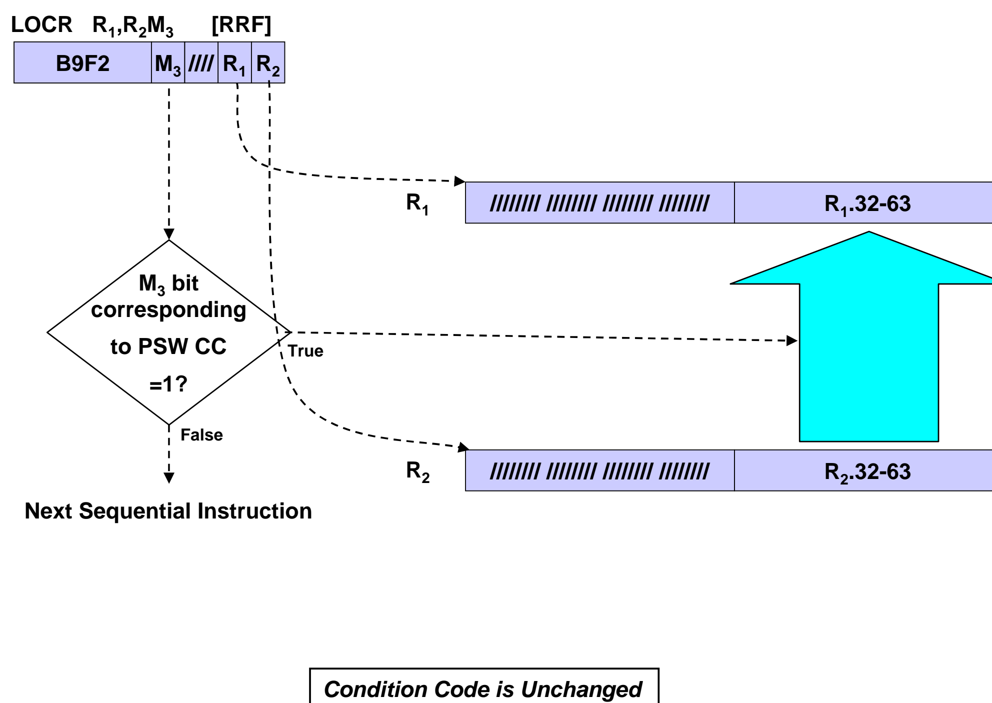
R_n Register operand ‘n’

S20 Storage operand designated by base register with 20-bit signed long displacement

For LOAD ON CONDITION, there are two forms of second operand: one source is a register and the other is a storage operand. For STORE ON CONDITION, the second operand is a storage operand. For each of these, there is an instruction that operates on 32-bit values and one that operates on 64-bit values.

As noted on the previous slide, the High-Level Assembler implements extended mnemonics for the load-and-store-on-condition facility. The extended mnemonic is formed by adding a suffix to one of the six basic mnemonics. When an extended mnemonic is coded, the conditional mask operand (the M_3 field) is not coded.

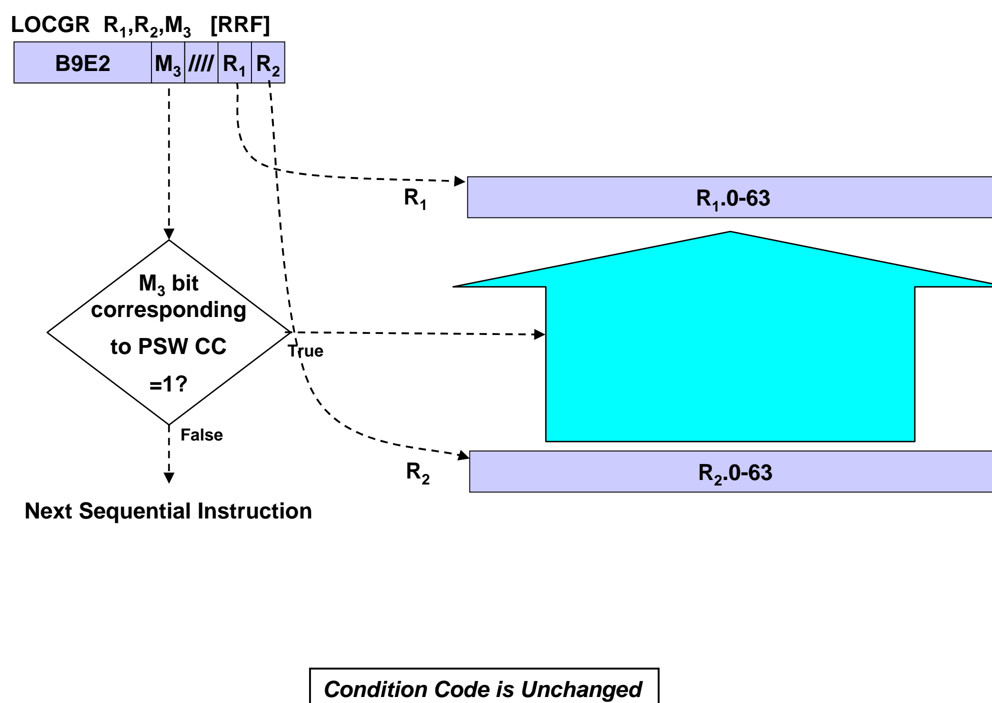
The extended mnemonics represent the conditions that would be expected after a comparison operation: E, H, L, NE, NH, and NL. As the expected usage is following a compare instruction, HLASM does not provide extended mnemonics for other conditions (particularly CC3). However, the programmer can specify these conditions by using the M_3 field.



This slide illustrates the operation of LOAD ON CONDITION (LOCR).

If the condition specified in the M_3 field of the instruction (or specified by the extended mnemonic) is true, bits 32-63 of the general register specified by the R_2 field of the instruction are copied into the corresponding bits of the general register specified by the R_1 field; bits 0-31 of the register specified by the R_1 field remain unchanged.

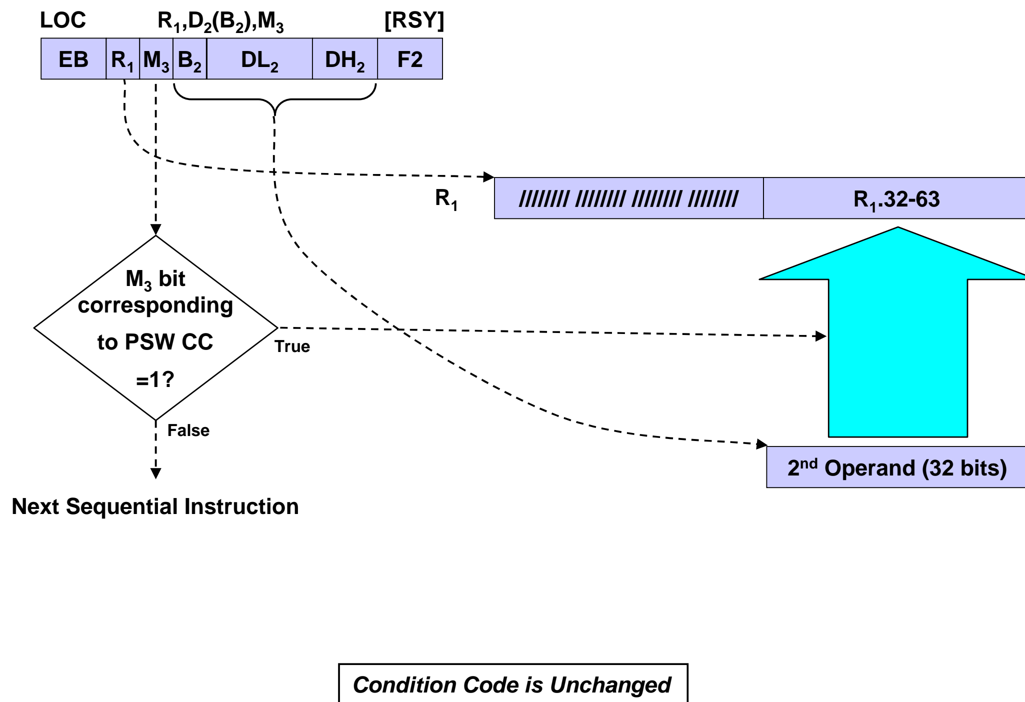
If the condition specified by the M_3 field (or extended mnemonic) is not true, all bits in the general register specified by the R_1 field remain unchanged.



This slide illustrates the operation of LOAD ON CONDITION (LOCGR).

If the condition specified in the M_3 field of the instruction (or specified by the extended mnemonic) is true, bits 0-63 of the general register specified by the R_2 field of the instruction are copied into the corresponding bits of the general register specified by the R_1 field.

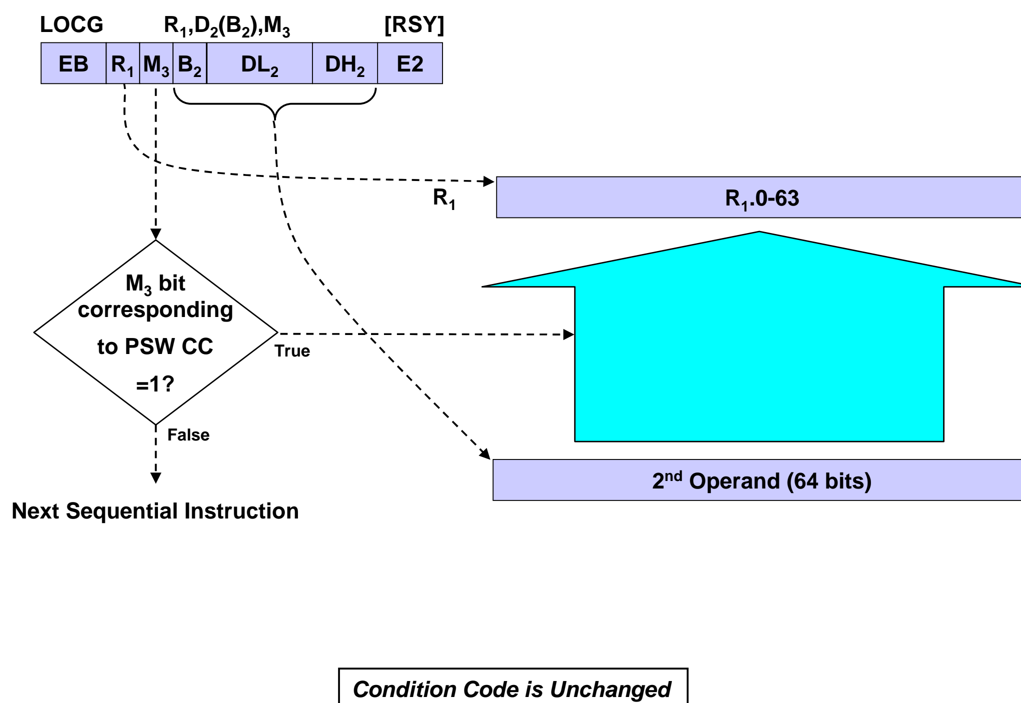
If the condition specified by the M_3 field (or extended mnemonic) is not true, all bits in the general register specified by the R_1 field remain unchanged.



This slide illustrates the operation of LOAD ON CONDITION (LOC).

If the condition specified in the M_3 field of the instruction (or specified by the extended mnemonic) is true, the four bytes designated by the second-operand location are copied into bits 32-63 of the general register specified by the R_1 field; bits 0-31 of the register remain unchanged.

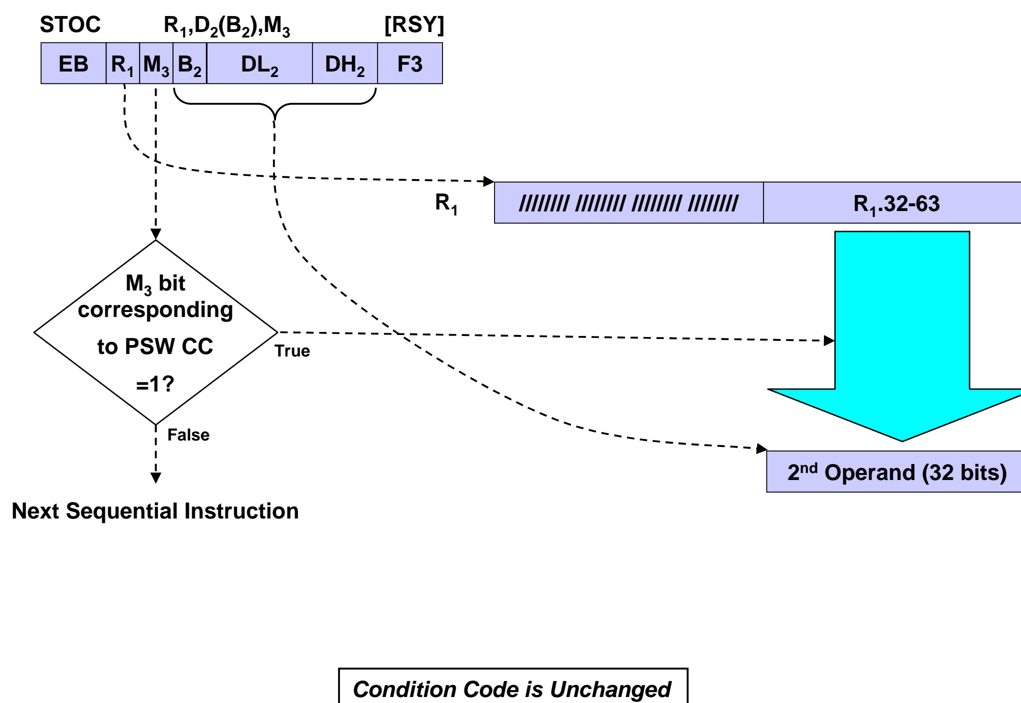
If the condition specified by the M_3 field (or extended mnemonic) is not true, all bits in the general register specified by the R_1 field remain unchanged.



This slide illustrates the operation of LOAD ON CONDITION (LOCG).

If the condition specified in the M_3 field of the instruction (or specified by the extended mnemonic) is true, the eight bytes designated by the second-operand location are copied into bits 0-63 of the general register specified by the R_1 field.

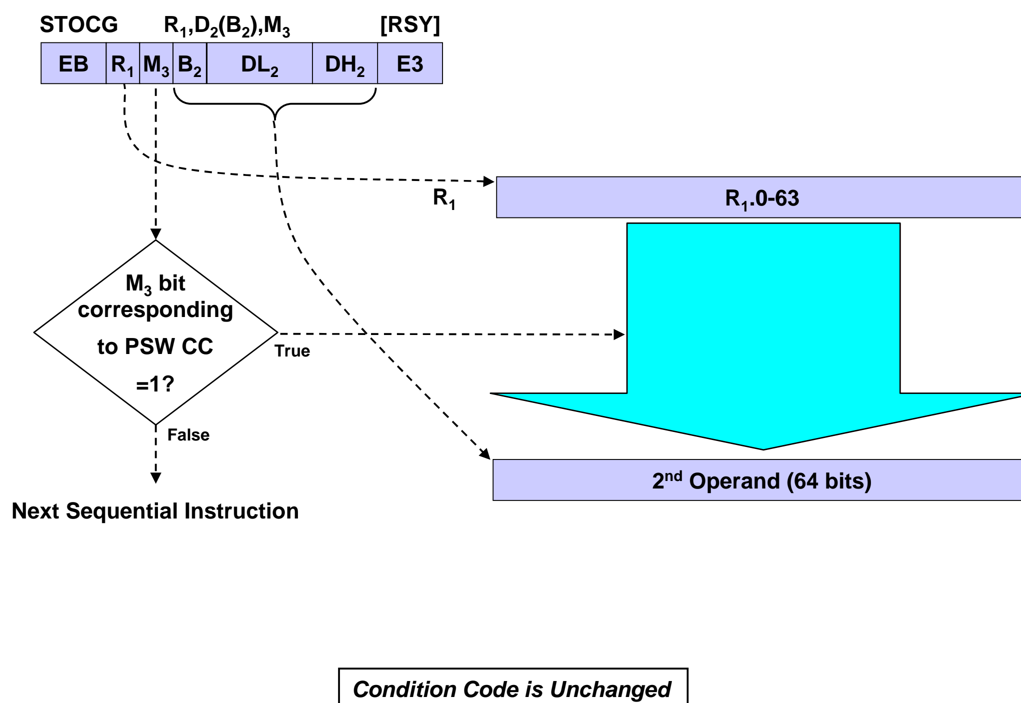
If the condition specified by the M_3 field (or extended mnemonic) is not true, all bits in the general register specified by the R_1 field remain unchanged.



This slide illustrates the operation of STORE ON CONDITION (STOC).

If the condition specified in the M_3 field of the instruction (or specified by the extended mnemonic) is true, bits 32-63 of the general register specified by the R_1 field are stored at the four-byte second-operand location.

If the condition specified by the M_3 field (or extended mnemonic) is not true, no store operation occurs.



This slide illustrates the operation of STORE ON CONDITION (STOCG).

If the condition specified in the M_3 field of the instruction (or specified by the extended mnemonic) is true, bits 0-63 of the general register specified by the R_1 field are stored at the eight-byte second-operand location

If the condition specified by the M_3 field (or extended mnemonic) is not true, no store operation occurs.

Distinct-Operands Facility (1)

- **Suite of instructions to provide nondestructive analogs to existing destructive instructions**
  - **Target register is separate from source registers**
  - **Nondestructive instructions provided for:**

ADD	**OR**
ADD LOGICAL	**SHIFT LEFT**
ADD LOG. w/SIGN. IMMED.	**SHIFT RIGHT**
AND	**SUBTRACT**
EXCLUSIVE OR	**SUBTRACT LOGICAL**

- **Intended to provide register-constraint relief for compilers**
- **Installation of the distinct-operands facility (& al.) indicated by facility bit 45**

Beginning with the original System/360, the architecture has a long tradition of performing arithmetic or logical operations on two source operands, and then replacing one of the source operands with the result. This was completely understandable for RR-format instructions, where the instruction format only had room for two registers.

With the advent of newer instruction formats, there is sufficient space for separate source and target operand specifications. z/Architecture began exploiting this with the 64-bit shift operations, and the decimal-floating-point facility extended the practice by having the results of floating point computations placed in a register that can be distinct from the two source registers.

Having a separate destination operand register provides greater flexibility to compiler designers and assembler programmers. When a source operand needs to be preserved, extra instructions are not needed to perform a copying operation.

The distinct-operands facility introduces a series of arithmetic and logical instructions that have a result register that can be distinct from any of the source operands. For all of the instructions, the first (result) and third (source) operands are in a register; depending on the instruction, the second operand is a register, immediate field, or storage-type operand.

All of the distinct-operand-facility instructions have a suffix of “K” in the mnemonic.

Distinct-Operands Facility (2):

Instruction	Mnemonic	OpCode	1st Operand	2nd Operand	3rd Operand
ADD	ARK	B9F8	R_1.32-63	R_2.32-63	R_3.32-63
ADD	AGRK	B9E8	R_1.0-63	R_2.0-63	R_3.0-63
ADD IMMEDIATE	AHIK	ECD8	R_1.32-63	I_2	R_3.32-63
ADD IMMEDIATE	AGHIK	ECD9	R_1.0-63	I_2	R_3.0-63
ADD LOGICAL	ALRK	B9FA	R_1.32-63	R_2.32-63	R_3.32-63
ADD LOGICAL	ALGRK	B9EA	R_1.0-63	R_2.0-63	R_3.0-63
ADD LOGICAL WITH SIGNED IMMEDIATE	ALHSIK	ECDA	R_1.32-63	I_2	R_3.32-63
ADD LOGICAL WITH SIGNED IMMEDIATE	ALGHSIK	ECDB	R_1.0-63	I_2	R_3.0-63
AND	NRK	B9F4	R_1.32-63	R_2.32-63	R_3.32-63
AND	NGRK	B9E4	R_1.0-63	R_2.0-63	R_3.0-63

Explanation:

I_2 Second operand is a 16-bit signed immediate value

R_n Register operand ‘n’

This slide introduces the various ADD and AND instructions in the distinct-operand facility.

For the ADD instructions, the second operand is either a register or immediate field. For the AND, OR, and XOR instructions, the second operand is always a register.

Distinct-Operands Facility (3):

Instruction	Mnemonic	OpCode	1st Operand	2nd Operand	3rd Operand
EXCLUSIVE OR	XRK	B9F7	R_1.32-63	R_2.32-63	R_3.32-63
EXCLUSIVE OR	XGRK	B9E7	R_1.0-63	R_2.0-63	R_3.0-63
OR	ORK	B9F6	R_1.32-63	R_2.32-63	R_3.32-63
OR	OGRK	B9E6	R_1.0-63	R_2.0-63	R_3.0-63
SHIFT LEFT SINGLE	SLAK	EBDD	R_1.32-63	S20	R_3.32-63
SHIFT LEFT SINGLE LOGICAL	SLLK	EBDF	R_1.32-63	S20	R_3.32-63
SHIFT RIGHT SINGLE	SRAK	EBDC	R_1.32-63	S20	R_3.32-63
SHIFT RIGHT SINGLE LOGICAL	SRLK	EBDE	R_1.32-63	S20	R_3.32-63
SUBTRACT	SRK	B9F9	R_1.32-63	R_2.32-63	R_3.32-63
SUBTRACT	SGRK	B9E9	R_1.0-63	R_2.0-63	R_3.0-63
SUBTRACT LOGICAL	SLRK	B9FB	R_1.32-63	R_2.32-63	R_3.32-63
SUBTRACT LOGICAL	SLGRK	B9EB	R_1.0-63	R_2.0-63	R_3.0-63

Explanation:

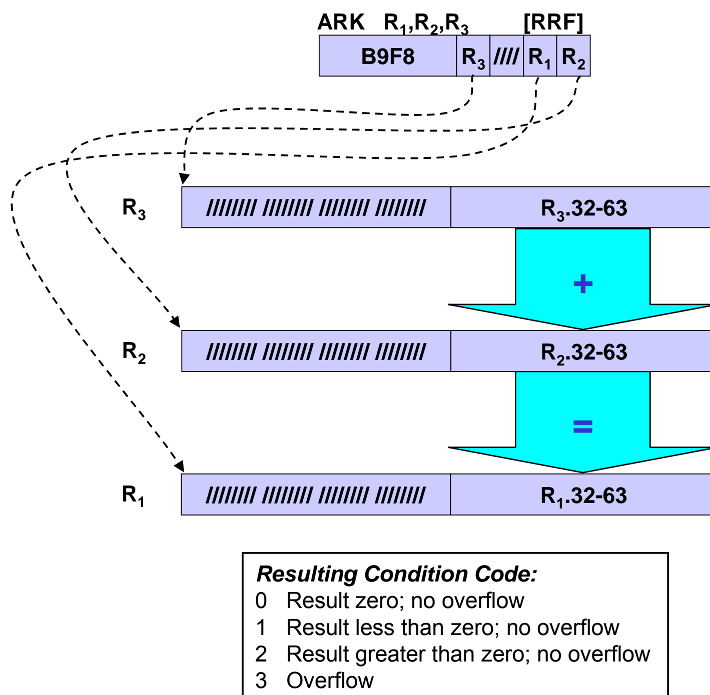
I_2 Second operand is a 16-bit signed immediate value

R_n Register operand 'n'

S20 Address designated by base register with 20-bit signed long displacement

This slide enumerates the remaining instructions in the distinct-operand facility.

For the SHIFT instructions, the second operand is not used to access storage; rather, the rightmost six bits of the second-operand address form the shift amount (just like any other shift operation).

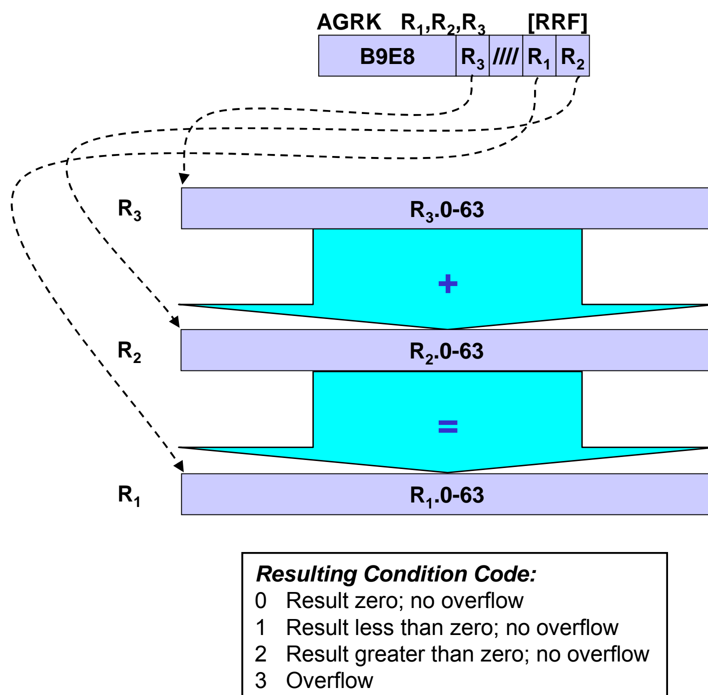


For ADD (ARK), the second operand is added to the third operand, and the result is placed in the first operand. Each operand occupies the rightmost 32 bits (bits 32-63) of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 field, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The condition code is set as with all signed addition instructions.

ADD (AGRK)

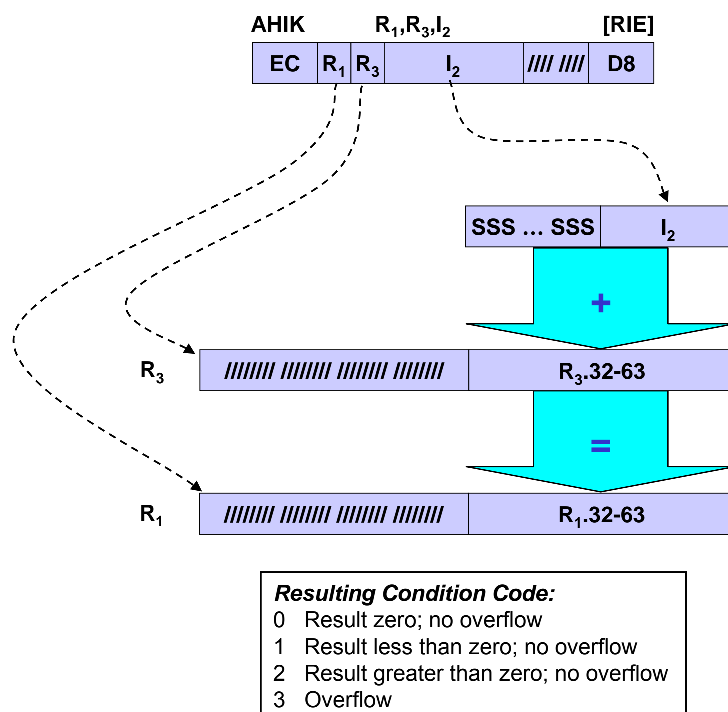


For ADD (AGRK), the second operand is added to the third operand, and the result is placed in the first operand. Each operand occupies all 64 bits of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 field, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged.

The condition code is set as with all signed addition instructions.

ADD IMMEDIATE (AHIK)

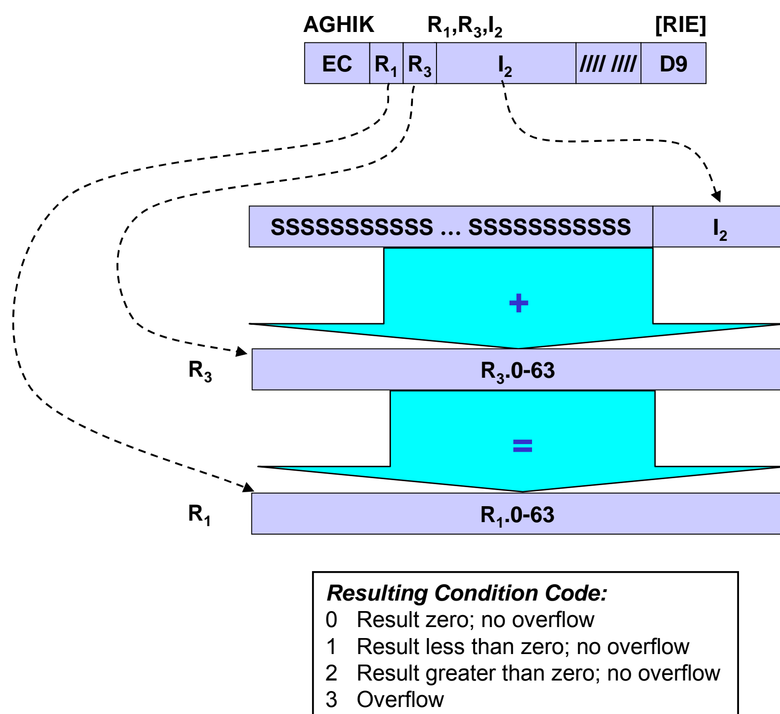


For ADD IMMEDIATE (AHIK), the 16-bit signed binary integer in the I_2 field of the instruction is sign extended on the left to form a 32-bit signed value which is added to the third operand. The result of this addition is placed in the first operand. The first and third operands occupy the rightmost 32 bits (bits 32-63) of the general registers designated by the R_1 and R_3 fields of the instruction, respectively.

Unless the R_1 field designates the same register as the R_3 field, the contents of the general register designated by the R_3 field remains unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The condition code is set as with all signed addition instructions.

ADD IMMEDIATE (AGHIK)

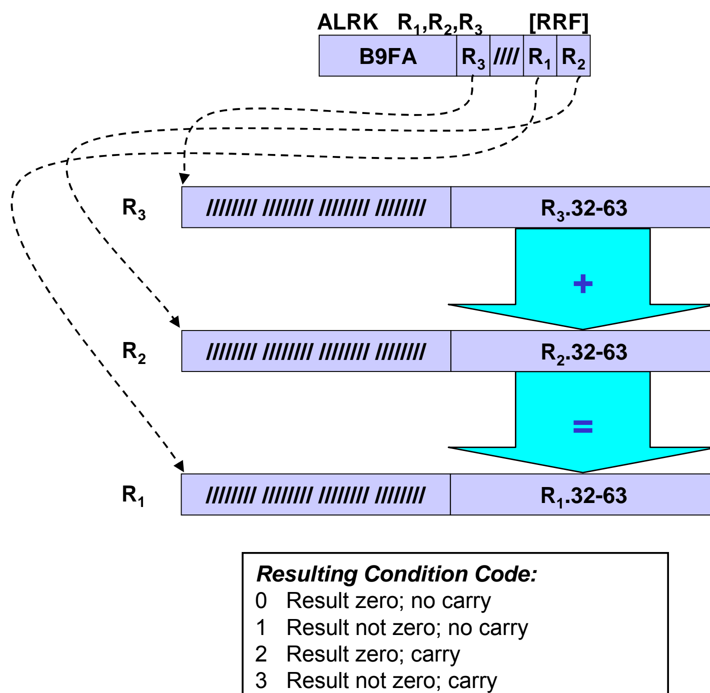


For ADD IMMEDIATE (AGHIK), the 16-bit signed binary integer in the I_2 field of the instruction is sign extended on the left to form a 64-bit signed value which is added to the third operand. The result of this addition is placed in the first operand. The first and third operands occupy all 64 bits of the general registers designated by the R_1 and R_3 fields of the instruction, respectively.

Unless the R_1 field designates the same register as the R_3 field, the contents of the general register designated by the R_3 field remains unchanged.

The condition code is set as with all signed addition instructions.

ADD LOGICAL (ALRK)



Resulting Condition Code:
0 Result zero; no carry
1 Result not zero; no carry
2 Result zero; carry
3 Result not zero; carry

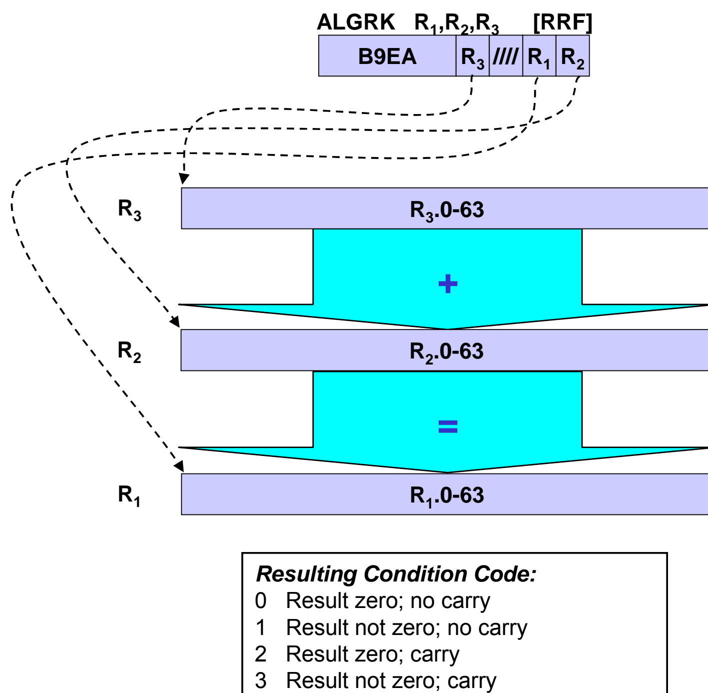
For ADD LOGICAL (ALRK), the second operand is added to the third operand, and the result is placed in the first operand. Each operand occupies the rightmost 32 bits (bits 32-63) of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 field, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The condition code is set as with all unsigned addition instructions.

IBM

ADD LOGICAL (ALGRK)

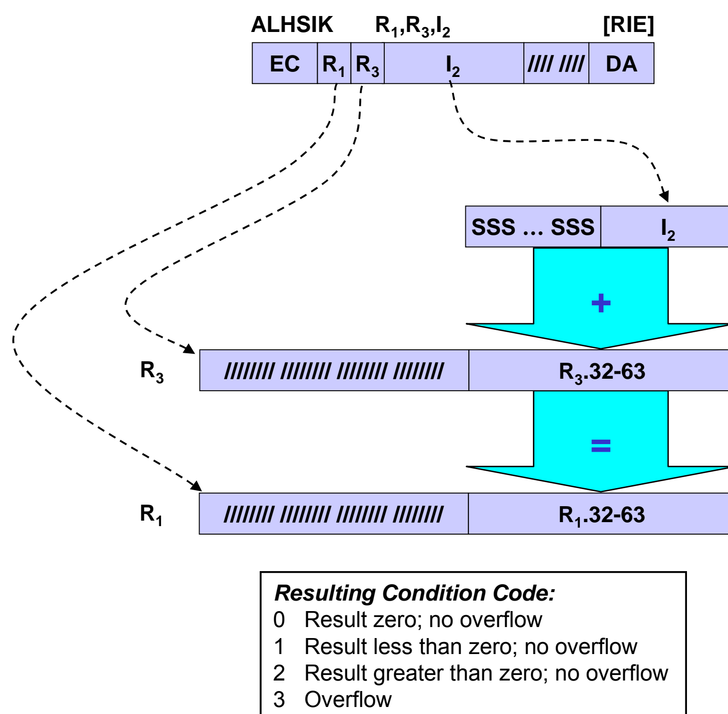


For ADD LOGICAL (ALGRK), the second operand is added to the third operand, and the result is placed in the first operand. Each operand occupies all 64 bits of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_3 field, the contents of the general register designated by the R_3 field remains unchanged.

The condition code is set as with all unsigned addition instructions.

ADD LOGICAL WITH SIGNED IMMEDIATE (ALHSIK)

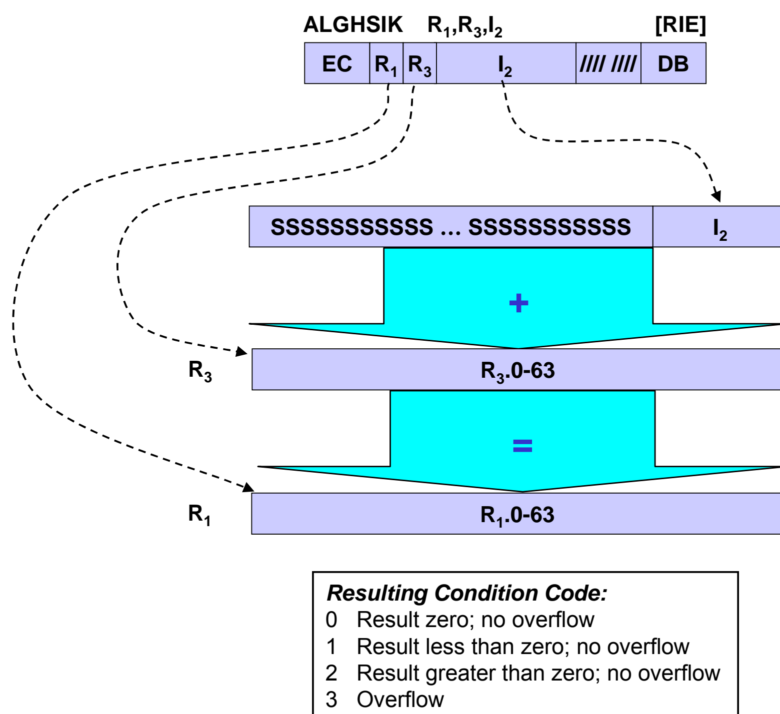


For ADD LOGICAL WITH SIGNED IMMEDIATE (ALHSIK), the 16-bit signed binary integer in the I_2 field of the instruction is sign extended on the left to form a 32-bit signed value which is added to the third operand. The result of this addition is placed in the first operand. The first and third operands occupy the rightmost 32 bits (bits 32-63) of the general registers designated by the R_1 and R_3 fields of the instruction, respectively.

Unless the R_1 field designates the same register as the R_3 field, the contents of the general register designated by the R_3 field remains unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The condition code is set as with all unsigned addition instructions.

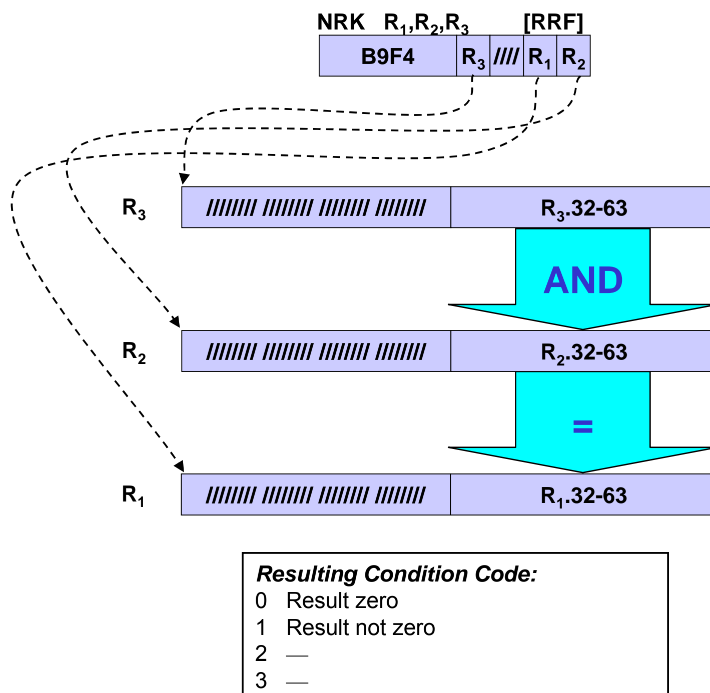
ADD LOGICAL WITH SIGNED IMMEDIATE (ALGHSIK)



For ADD LOGICAL WITH SIGNED IMMEDIATE (ALGHSIK), the 16-bit signed binary integer in the I_2 field of the instruction is sign extended on the left to form a 64-bit signed value which is added to the third operand. The result of this addition is placed in the first operand. The first and third operands occupy all 64 bits of the general registers designated by the R_1 and R_3 fields of the instruction, respectively.

Unless the R_1 field designates the same register as the R_3 field, the contents of the general register designated by the R_3 field remains unchanged.

The condition code is set as with all unsigned addition instructions.

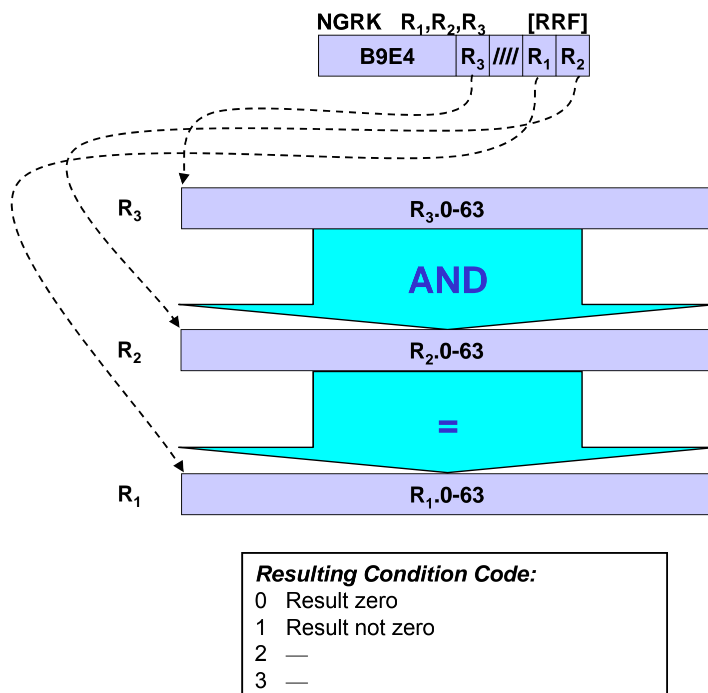


For AND (NRK), the second operand is logically ANDed with the third operand, and the result is placed in the first operand. Each operand occupies the rightmost 32 bits (bits 32-63) of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 field, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The condition code is set as with all common logical-operation instructions.

AND (NGRK)

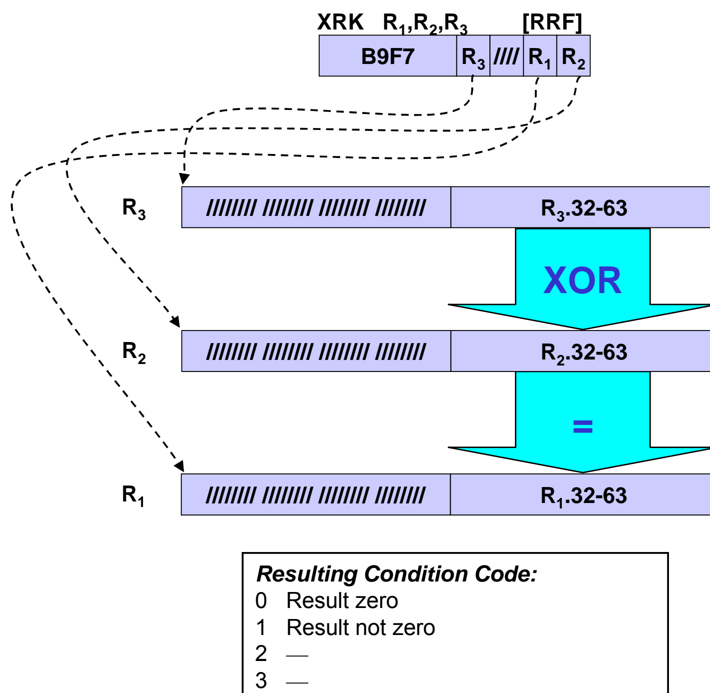


For AND (NGRK), the second operand is logically ANDed with the third operand, and the result is placed in the first operand. Each operand occupies all 64 bits of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 field, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged.

The condition code is set as with all common logical-operation instructions.

EXCLUSIVE OR (XRK)

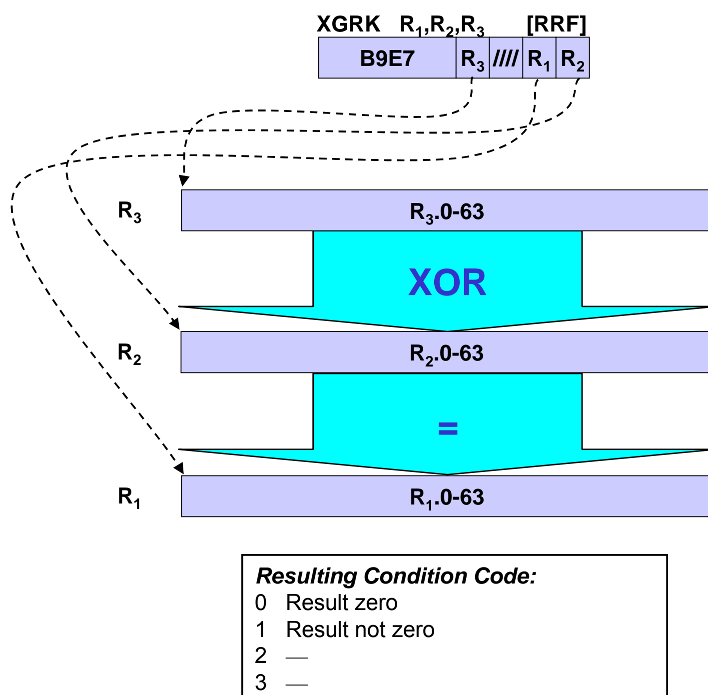


For EXCLUSIVE OR (XRK), the second operand is logically exclusive-ORed with the third operand, and the result is placed in the first operand. Each operand occupies the rightmost 32 bits (bits 32-63) of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 field, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The condition code is set as with all common logical-operation instructions.

EXCLUSIVE OR (XGRK)



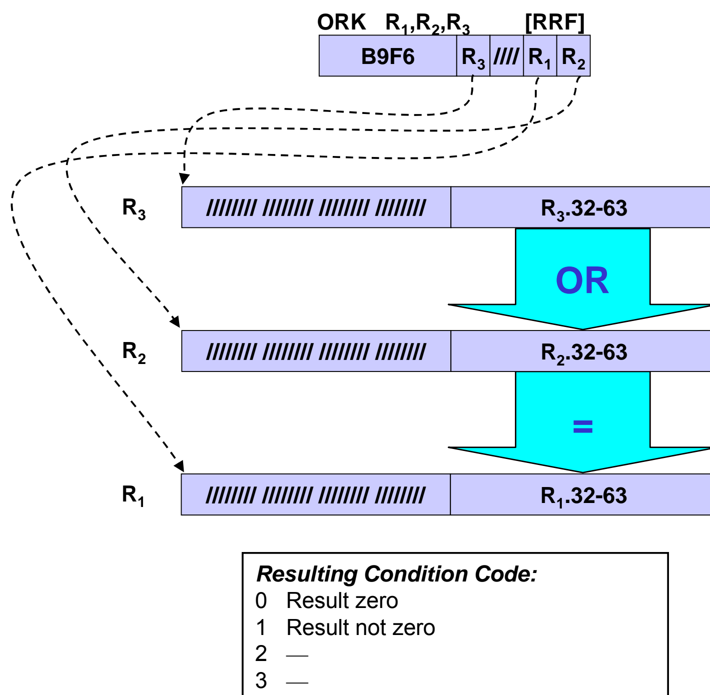
Resulting Condition Code:
0 Result zero
1 Result not zero
2 —
3 —

For EXCLUSIVE OR (XGRK), the second operand is logically exclusive-ORed with the third operand, and the result is placed in the first operand. Each operand occupies all 64 bits of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 field, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged.

The condition code is set as with all common logical-operation instructions.

OR (ORK)

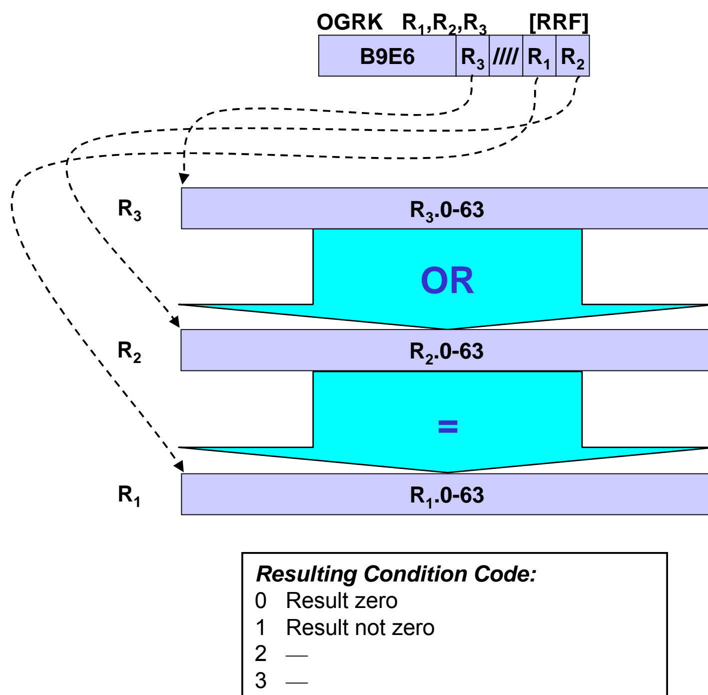


For OR (XRK), the second operand is logically ORed with the third operand, and the result is placed in the first operand. Each operand occupies the rightmost 32 bits (bits 32-63) of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 field, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The condition code is set as with all common logical-operation instructions.

OR (OGRK)



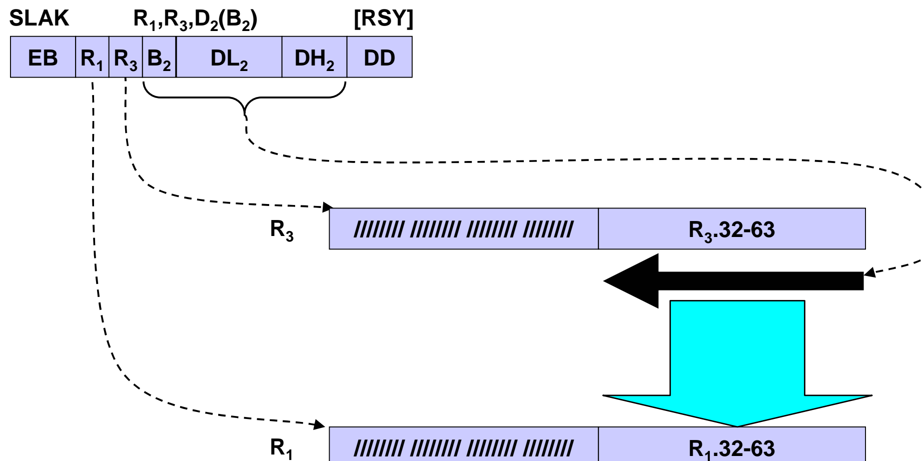
Resulting Condition Code:
0 Result zero
1 Result not zero
2 —
3 —

For OR (XGRK), the second operand is logically ORed with the third operand, and the result is placed in the first operand. Each operand occupies all 64 bits of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 field, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged.

The condition code is set as with all common logical-operation instructions.

SHIFT LEFT SINGLE (SLAK)



Resulting Condition Code:
0 Result zero (no overflow)
1 Result less-than zero (no overflow)
2 Result greater-than zero (no overflow)
3 Overflow

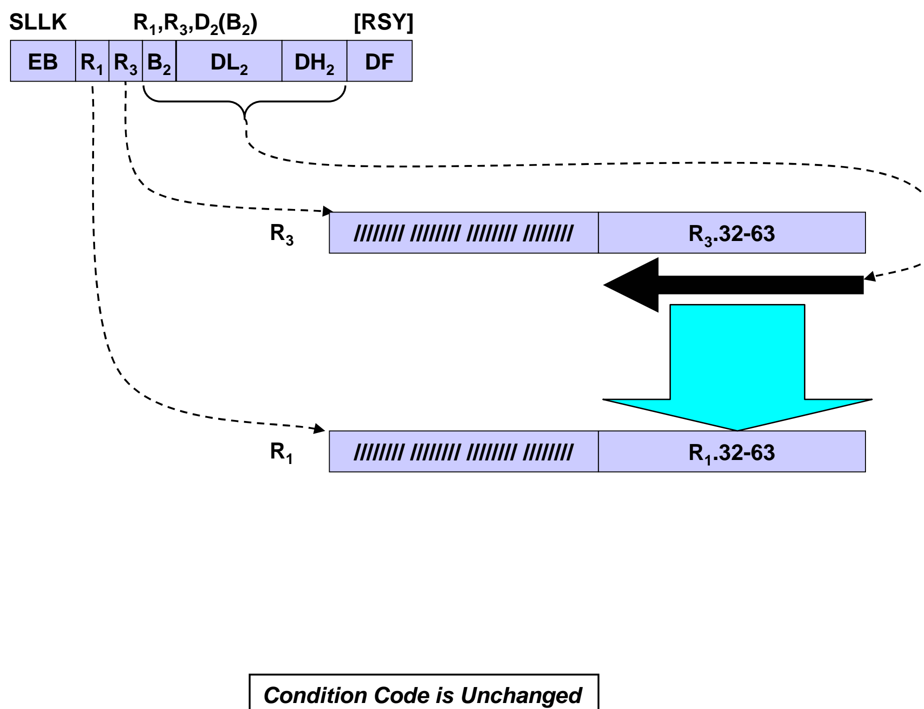
For SHIFT LEFT SINGLE (SLAK), the 31-bit numeric part of the third operand is shifted left by the number of bits specified by the second-operand address, and the result is placed in the first operand. Zeros are supplied to the vacated bit positions on the right. The first and third operands are 32-bit signed binary integers in bits 32-63 of the respective registers, with the sign in bit position 32.

Unless the R_1 field designates the same register as the R_3 field, the contents of the general register designated by the R_3 field remains unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The second-operand address is not used to address data; rather, its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

The condition code is set based on whether the results are negative, zero, positive, or cause an overflow. An overflow occurs if one or more bits are shifted left out of bit position 33; if the fixed-point-overflow mask bit in the PSW is one, a fixed-point-overflow program interruption occurs.

SHIFT LEFT SINGLE LOGICAL (SLLK)

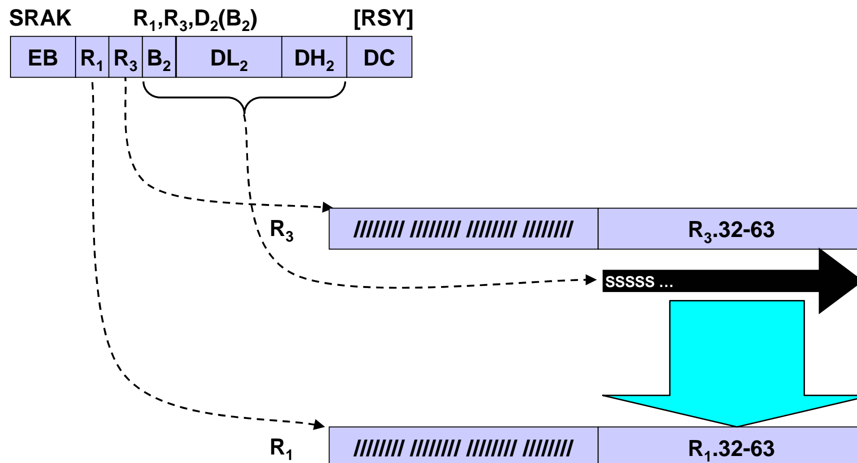


For SHIFT LEFT SINGLE LOGICAL (SLLK), the third operand is shifted left by the number of bits specified by the second-operand address, and the result is placed in the first operand. Zeros are supplied to the vacated bit positions on the right. The first and third operands are 32-bit unsigned binary integers occupying the rightmost 32 bits (bits 32-63) of the general registers designated by the R_1 and R_3 fields of the instruction, respectively.

Unless the R_1 field designates the same register as the R_3 field, the contents of the general register designated by the R_3 field remains unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The second-operand address is not used to address data; rather, its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

SHIFT RIGHT SINGLE (SRAK)



Resulting Condition Code:

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

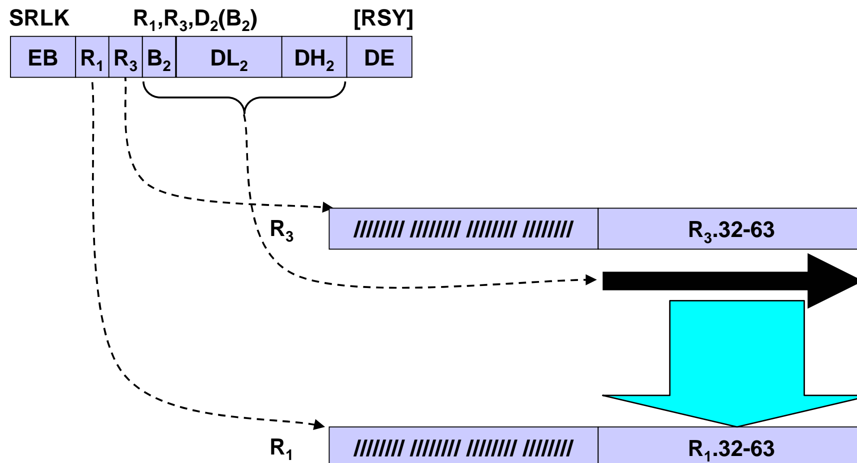
For SHIFT RIGHT SINGLE (SRAK), the 31-bit integer portion of the third operand is shifted right by the number of bits specified by the second-operand address, and the result is placed in the first operand. The first and third operands are 32-bit signed binary integers in bits 32-63 of the respective registers, with the sign in bit position 32. The sign bit is supplied to the vacated bit positions on the left.

Unless the R_1 field designates the same register as the R_3 field, the contents of the general register designated by the R_3 field remains unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The second-operand address is not used to address data; rather, its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

The condition code is set based on whether the results are negative, zero, or positive.

SHIFT RIGHT SINGLE LOGICAL (SRLK)



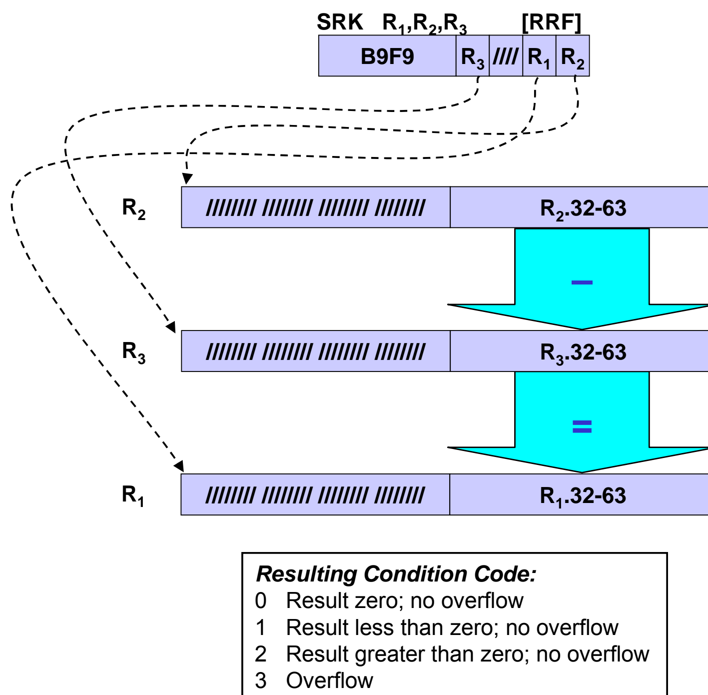
Condition Code is Unchanged

For SHIFT RIGHT SINGLE LOGICAL (SRLK), the third operand is shifted right by the number of bits specified by the second-operand address, and the result is placed in the first operand. Zeros are supplied to the vacated bit positions on the left. The first and third operands are 32-bit unsigned binary integers occupying the rightmost 32 bits (bits 32-63) of the general registers designated by the R_1 and R_3 fields of the instruction, respectively.

Unless the R_1 field designates the same register as the R_3 field, the contents of the general register designated by the R_3 field remains unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The second-operand address is not used to address data; rather, its rightmost six bits indicate the number of bit positions to be shifted. The remainder of the address is ignored.

SUBTRACT (SRK)

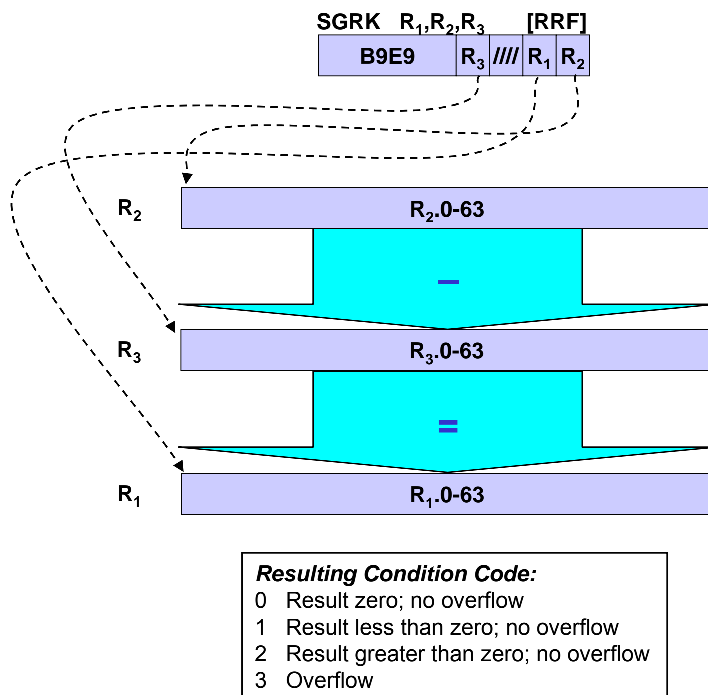


For SUBTRACT (SRK), the third operand is subtracted from the second operand, and the result is placed in the first operand. Each operand occupies the rightmost 32 bits (bits 32-63) of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 fields, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The condition code is set as with all signed subtraction instructions.

SUBTRACT (SGRK)

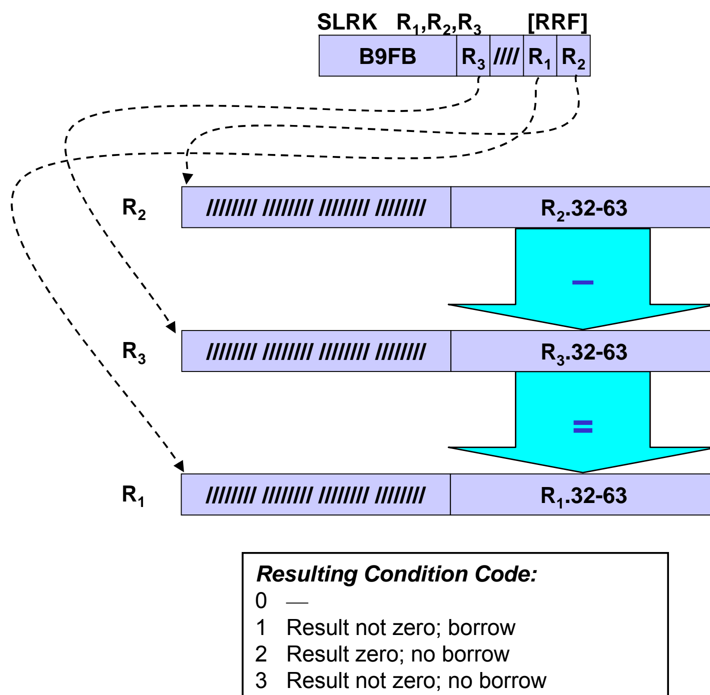


For SUBTRACT (SGRK), the third operand is subtracted from the second operand, and the result is placed in the first operand. Each operand occupies all 64 bits of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 fields, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged.

The condition code is set as with all signed subtraction instructions.

SUBTRACT LOGICAL (SLRK)

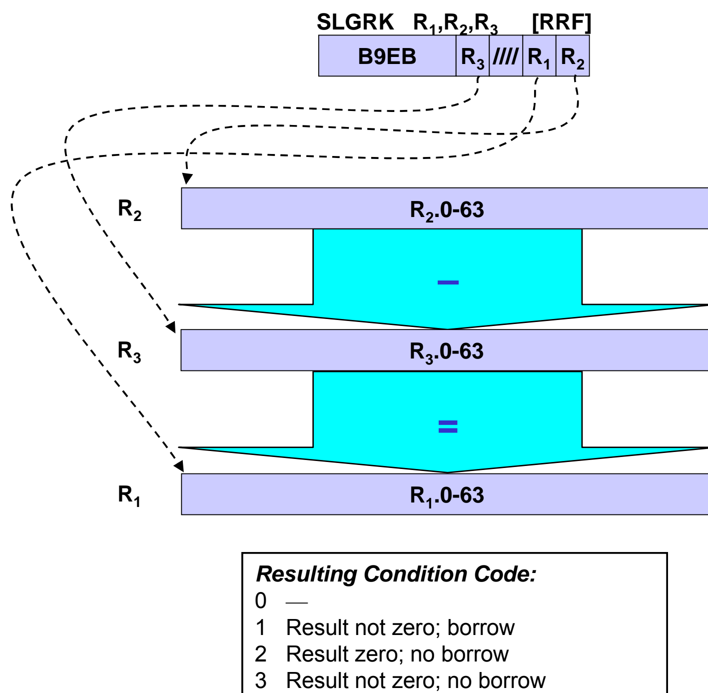


For SUBTRACT LOGICAL (SLRK), the third operand is subtracted from the second operand, and the result is placed in the first operand. Each operand occupies the rightmost 32 bits (bits 32-63) of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 field, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged. The contents of bit positions 0-31 of the general register designated by the R_1 field always remains unchanged.

The condition code is set as with all unsigned subtraction instructions.

SUBTRACT LOGICAL (SLGRK)



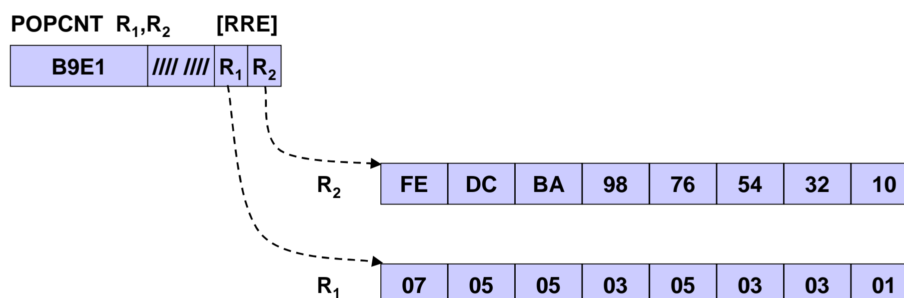
For SUBTRACT LOGICAL (SLGRK), the third operand is subtracted from the second operand, and the result is placed in the first operand. Each operand occupies all 64 bits of the general register designated by the corresponding R field of the instruction.

Unless the R_1 field designates the same register as the R_2 or R_3 fields, the contents of the general registers designated by the R_2 and R_3 fields remain unchanged.

The condition code is set as with all unsigned subtraction instructions.

Population-Count Facility

- Instruction for determining the number of one bits in each of the eight bytes of a GR
- Installation of the population-count facility (& al.) indicated by facility bit 45



- To tabulate one bits in a register, post processing is required, e.g.,

```
POPCNT  8,15
MSG     8,=X'0101010101010101'
SRLG    8,8,56
```


The POPULATION COUNT instruction is useful for determining the number of one bits contained in each byte of a 64-bit register. For each byte in the register designated by the R_2 field of the instruction, POPCNT places an 8-bit count of the number of one bits into the corresponding byte of the general register designated by the R_1 field of the instruction.

POPCNT may be useful in applications that use bit maps to indicate the presence, validity, or availability of some group of resources. An example of such bit-map usage may be found in Appendix A of the *z/Architecture Principles of Operation* (SA22-7832) in the programming example for the FIND LEFTMOST ONE instruction.

POPCNT provides only an indication of one bits for each byte. If the application needs to know the number of one bits in larger units, it must perform its own post processing. The example shown illustrates a clever way of summing the eight bytes, however on some models, the MULTIPLY SINGLE instruction may be slower than a group of instructions, for example:

```
POPCNT 8,15
AHHLR  8,8,8
SLLG   9,8,16
ALGR   8,9
SLLG   9,8,8
ALGR   8,9
SRLG   8,8,56
```

This sequence of instructions can easily be adapted to produce a count of one bits per halfword or per word.

Floating-Point Extension Facility (1)

Extensions to BFP and DFP instructions:

- **New BFP rounding mode:**
  - Round to prepare for shorter precision
  - Control in the floating-point control register (FPCR)
- **New DFP quantum exception:**
  - New mask and flag controls in the FPCR
- **New IEEE inexact-exception control (Xxc)**
  - Alternate forms of many BFP and DFP instructions with new M_4 field
- **New BFP and DFP instructions for converting to/from fixed-point**
  - CONVERT FROM LOGICAL
  - CONVERT TO LOGICAL

The floating-point extension facility provides enhancements to the binary-floating-point (BFP) and decimal-floating-point (DFP) facilities. BFP was added to the architecture late in the life of ESA/390 (circa 1998); DFP was added in the System z9-109 (circa 2005).

For BFP, a new rounding mode – round to prepare for shorter precision – is provided. The new rounding mode may be controlled by means of a new bit in the floating-point control register, or by means of the M_3 field in alternate forms of the CONVERT FROM FIXED, CONVERT TO FIXED, LOAD FP INTEGER, and LOAD ROUNDED instructions.

For most computational DFP operations, a new quantum exception-exception condition exists whenever the delivered DFP result is inexact, or when the result is exact and finite but the delivered quantum differs from the preferred quantum. The quantum-exception condition also applies to the DIVIDE, LOAD FP INTEGER, QUANTIZE, and REROUND instructions, but for somewhat different causes. Whether or not the quantum-exception condition results in an interruption is controlled and indicated by a new mask and flag bit, respectively, in the floating-point control register.

For both BFP and DFP, a new M_4 field has been added to certain alternate forms of instructions to control the IEEE inexact-exception condition.

Finally, both BFP and DFP have new instructions, CONVERT FROM LOGICAL and CONVERT TO LOGICAL, for converting between unsigned binary integers and the respective floating-point formats.

Floating-Point Extension Facility (2)
New BFP Instructions

Instruction	Mnemonic	Format	Opcode
CONVERT FROM LOGICAL (Extended BFP ← 32)	CXLFBR	RRF	B392
CONVERT FROM LOGICAL (Long BFP ← 32)	CDLFBR	RRF	B391
CONVERT FROM LOGICAL (Short BFP ← 32)	CELFBR	RRF	B390
CONVERT FROM LOGICAL (Extended BFP ← 64)	CXLGBR	RRF	B3A2
CONVERT FROM LOGICAL (Long BFP ← 64)	CDLGBR	RRF	B3A1
CONVERT FROM LOGICAL (Short BFP ← 64)	CELGBR	RRF	B3A0
CONVERT TO LOGICAL (32 ← Extended BFP)	CLFXBR	RRF	B39E
CONVERT TO LOGICAL (32 ← Long BFP)	CLFDBR	RRF	B39D
CONVERT TO LOGICAL (32 ← Short BFP)	CLFEBR	RRF	B39C
CONVERT TO LOGICAL (64 ← Extended BFP)	CLGXBR	RRF	B3AE
CONVERT TO LOGICAL (64 ← Long BFP)	CLGDBR	RRF	B3AD
CONVERT TO LOGICAL (64 ← Short BFP)	CLGEBR	RRF	B3AC
SET BFP ROUNDING MODE	SRNMB	S	B2B8

Note: All instructions are documented in Chapter 19 except for SRNMB which is documented in Chapter 9. SRNMB is a complete superset of the functionality of SRNM.

This slide illustrates the new BFP instructions.

The majority of the instructions are various forms of the CONVERT FROM LOGICAL and CONVERT TO LOGICAL instructions. CONVERT FROM LOGICAL converts an unsigned binary integer in the second operand to a binary-floating-point value that is placed in the first operand. CONVERT TO LOGICAL rounds a binary-floating-point value in the second operand to an integer value and then converts it to fixed-point format in the first operand.

SET BFP ROUNDING MODE (SRNM) was the original instruction to set the 2-bit BFP rounding mode in the floating-point control register (FPCR). The new SRNMB instruction sets the 3-bit BFP rounding mode in the FPCR. SRNMB is a complete superset of the functionality of SRNM (SRNM is now deprecated.)

Floating-Point Extension Facility (3)
Alternate Forms of BFP Instructions

Instruction	Mnemonic	Format	Opcode
CONVERT FROM FIXED (Extended BFP ← 32)	CXFBRA	RRF	B396
CONVERT FROM FIXED (Long BFP ← 32)	CDFBRA	RRF	B395
CONVERT FROM FIXED (Short BFP ← 32)	CEFBRA	RRF	B394
CONVERT FROM FIXED (Extended BFP ← 64)	CXGBRA	RRF	B3A6
CONVERT FROM FIXED (Long BFP ← 64)	CDGBRA	RRF	B3A5
CONVERT FROM FIXED (Short BFP ← 64)	CEGBRA	RRF	B3A4
CONVERT TO FIXED (32 ← Extended BFP)	CFXBRA	RRF	B39A
CONVERT TO FIXED (32 ← Long BFP)	CFDBRA	RRF	B399
CONVERT TO FIXED (32 ← Short BFP)	CFEBRA	RRF	B398
CONVERT TO FIXED (64 ← Extended BFP)	CGXBRA	RRF	B3AA
CONVERT TO FIXED (64 ← Long BFP)	CGDBRA	RRF	B3A9
CONVERT TO FIXED (64 ← Short BFP)	CGEBRA	RRF	B3A8
LOAD FP INTEGER (Extended BFP)	FIXBRA	RRF	B347
LOAD FP INTEGER (Long BFP)	FIDBRA	RRF	B35F
LOAD FP INTEGER (Short BFP)	FIEBRA	RRF	B357
LOAD ROUNDED (Long BFP ← Extended)	LDXBRA	RRF	B345
LOAD ROUNDED (Short BFP ← Extended)	LEXBRA	RRF	B346
LOAD ROUNDED (Short BFP ← Long)	LEDBRA	RRF	B344

This slide illustrates alternate forms of existing BFP instructions, as indicated by the “A” suffix on the mnemonic. The actual operation codes for these instructions are identical to those generated from mnemonics without the A, but the High-Level Assembler recognizes new operands when the “A” suffix is present.

For CONVERT FROM FIXED and LOAD ROUNDED, the alternate-mnemonic forms add both an M_3 and M_4 operand. The M_3 operand provides a rounding control, and the M_4 operand provides the IEEE-inexact-exception control. For CONVERT TO FIXED and LOAD FP INTEGER, a rounding control is already provided in the form of the M_3 field, but the new M_4 operand provides the IEEE-inexact-exception control. For each of these instructions, and for DIVIDE TO INTEGER, the new rounding method (round to prepare for shorter precision) may be specified.

Floating-Point Extension Facility (4)
New DFP Instructions

Instruction	Mnemonic	Format	Opcode
CONVERT FROM FIXED (Extended DFP ← 32)	CXFTR	RRF	B959
CONVERT FROM FIXED (Long DFP ← 32)	CDFTR	RRF	B951
CONVERT FROM LOGICAL (Extended DFP ← 32)	CXLFTR	RRF	B95B
CONVERT FROM LOGICAL (Long DFP ← 32)	CDLFTR	RRF	B953
CONVERT FROM LOGICAL (Extended DFP ← 64)	CXLGTR	RRF	B95A
CONVERT FROM LOGICAL (Long DFP ← 64)	CDLGTR	RRF	B952
CONVERT TO FIXED (32 ← Extended DFP)	CFXTR	RRF	B949
CONVERT TO FIXED (32 ← Long DFP)	CFDTR	RRF	B941
CONVERT TO LOGICAL (32 ← Extended DFP)	CLFXTR	RRF	B94B
CONVERT TO LOGICAL (32 ← Long DFP)	CLFDTR	RRF	B943
CONVERT TO LOGICAL (64 ← Extended DFP)	CLGXTR	RRF	B94A
CONVERT TO LOGICAL (64 ← Long DFP)	CLGDTR	RRF	B942

This slide illustrates the new DFP instructions.

As with BFP, the new DFP instructions are various forms of the CONVERT FROM LOGICAL and CONVERT TO LOGICAL instructions. CONVERT FROM LOGICAL converts an unsigned binary integer in the second operand to a decimal-floating-point value that is placed in the first operand. CONVERT TO LOGICAL rounds a decimal-floating-point value in the second operand to an integer value and then converts it to unsigned fixed-point format in the first operand.

Floating-Point Extension Facility (5)
Alternate Forms of DFP Instructions

Instruction	Mnemonic	Format	Opcode
ADD (Extended DFP)	AXTRA	RRF	B3DA
ADD (Long DFP)	ADTRA	RRF	B3D2
CONVERT FROM FIXED (Extended DFP ← 64)	CXGTRA	RRF	B3F9
CONVERT FROM FIXED (Long DFP ← 64)	CDGTRA	RRF	B3F1
CONVERT TO FIXED (64 ← Extended DFP)	CGXTRA	RRF	B3E9
CONVERT TO FIXED (64 ← Long DFP)	CGDTRA	RRF	B3E1
DIVIDE (Extended DFP)	DXTRA	RRF	B3D9
DIVIDE (Long DFP)	DDTRA	RRF	B3D1
MULTIPLY (Extended DFP)	MXTRA	RRF	B3D8
MULTIPLY (Long DFP)	MDTRA	RRF	B3D0
SUBTRACT (Extended DFP)	SXTRA	RRF	B3DB
SUBTRACT (Long DFP)	SDTRA	RRF	B3D3

This slide illustrates alternate forms of existing DFP instructions, as indicated by the “A” suffix on the mnemonic. The actual operation codes for these instructions are identical to those generated from mnemonics without the A, but the High-Level Assembler recognizes new operands when the “A” suffix is present.

For the arithmetic operations, ADD, DIVIDE, MULTIPLY, and SUBTRACT, a new M_4 operand is provided to control the rounding mode of the result.

For CONVERT FROM FIXED, a new M_3 operand is provided to control the rounding mode of the result, and a new M_4 operand provides the IEEE-inexact-exception control.

For CONVERT TO FIXED, a new M_4 operand provides the IEEE-inexact-exception control.

Also, for all DFP instructions for which a rounding mode exists in the base architecture (i.e., the M_3 field of CONVERT TO FIXED, LOAD FP INTEGER, LOAD ROUNDED, QUANTIZE, and REROUND), additional rounding methods are available.

Message-Security Assist Extension 3 (MSA-X3)

- Protects user cryptographic keys by encrypting them under machine-generated wrapping keys:
  - 256-Bit AES Wrapping-Key Register
  - 256-Bit AES Wrapping-Key Verification-Pattern Register
  - 192-Bit DEA Wrapping-Key Register
  - 192-Bit DEA Wrapping-Key Verification-Pattern Register
- MSA-X3 available on the System z10 GA3 (November 2009)
- Installation of MSA-X3 indicated by facility bit 76

The message-security assist was introduced in the System z10 at general-availability level 3 (November 2009). Although it is not new in the z196, we'll devote a few slides to it, as it hasn't been published before.

MSA-X3 provides a means to protect user cryptographic keys by encrypting them under machine-generated wrapping keys. When this extension is installed, two wrapping keys are provided for each configuration: one for protecting user DEA keys and another for protecting user AES keys. The wrapping keys reside in the machine so that, with an appropriate setting of controls, no clear value of user cryptographic keys is observed anywhere in the system by any program.

The message-security-assist extension 3 may be available on models implementing the message-security assist. The extension provides the following features:

• A 256-Bit AES Wrapping-Key Register: The register contents are used to protect user AES keys.

• A 256-Bit AES Wrapping-Key Verification-Pattern Register: The register contents are used to identify the version of the AES wrapping key.

• A 192-Bit DEA Wrapping-Key Register: The register contents are used to protect user DEA keys.

• A 192-Bit DEA Wrapping-Key Verification-Pattern Register: The register contents are used to identify the version of the DEA wrapping key.

A new section has been added to the back of the General Instructions chapter of the *z/Architecture Principles of Operation* describing the protection of cryptographic keys.

MSA-X3 New Instruction

- PERFORM CRYPTOGRAPHIC KEY MANAGEMENT OPERATION (PCKMO)
  - Functions:
    - Query
    - Encrypt DEA Key
    - Encrypt TDEA 128 Key
    - Encrypt TDEA 192 Key
    - Encrypt AES 128 Key
    - Encrypt AES 192 Key
    - Encrypt AES 256 Key
  - Provide a means of importing clear cryptographic keys
  - Privileged operation

PERFORM CRYPTOGRAHPIC KEY MANAGEMENT OPERATION (PCKMO) is a control (privileged) instruction that provides a means of importing clear cryptographic keys.

MSA-X3 New Functions for Existing Instructions

- **CIPHER MESSAGE (KM),**
- **CIPHER MESSAGE WITH CHAINING (KMC), and**
- **COMPUTE MESSAGE AUTHENTICATION CODE (KMAC)**
  - **New Functions for each instruction:**
    - **Encrypted DEA Key (KM, KMC, & KMAC)**
    - **Encrypted TDEA 128 Key (KM, KMC, & KMAC)**
    - **Encrypted TDEA 192 Key (KM, KMC, & KMAC)**
    - **Encrypted AES 128 Key (KM & KMC only)**
    - **Encrypted AES 192 Key (KM & KMC only)**
    - **Encrypted AES 256 Key (KM & KMC only)**
  - **New functions use encrypted cryptographic key**

New functions are also added to the existing CIPHER MESSAGE (KM), CIPHER MESSAGE WITH CHANING (KMC), and COMPUTE MESSAGE AUTHENTICATION CODE (KMAC) instructions that allow encryption to be performed using the *encrypted* keys.

Message-Security Assist Extension 4 (MSA-X4)

- Provides support for:
  - Cipher-feedback (CFB) mode
  - Output-feedback (OFB) mode
  - Counter (CTR) mode
- Provides primitives to facilitate support of:
  - Cipher-based message-authentication (CMAC) mode
  - Counter with cipher-block chaining – message authentication code (CCM) mode
  - Galois/counter mode
  - XEX-based Tweaked CodeBook mode with CipherText Stealing (XTS) mode
- Installation of MSA-X4 indicated by facility bit 77
- Requires MSA-3 to be installed

The message-security-assist extension 4 (MSA-X4) is introduced with the IBM zEnterprise 196. It requires that the MSA-X3 facility also be installed.

MSA X4 provides support for cipher feedback (CFB) mode, output feedback (OFB) mode, and counter (CTR) mode of encryption and decryption. Additionally, primitive operations are provided to facility the support for the cipher-based message-authentication (CMAC) mode, the counter with cipher-block-chaining message-authentication code (CMM) mode, the Galois/counter mode, and the XTS mode.

MSA-X4 New Instructions

- CIPHER MESSAGE WITH CFB (KMF)
- CIPHER MESSAGE WITH COUNTER (KMCTR)
- CIPHER MESSAGE WITH OFB (KMO)
  - Functions:
    - Query
    - DEA
    - TDEA
    - TDEA 192
    - Encrypted DEA
    - Encrypted TDEA
    - Encrypted TDEA 192
    - AES 128
    - AES 192
    - AES 256
    - Encrypted AES 128
    - Encrypted AES 192
    - Encrypted AES 256

MSA-X4 introduces four new instructions, three of which are enumerated on this slide:

• CIPHER MESSAGE WITH CFB (KMF) [cipher feedback mode]

• CIPHER MESSAGE WITH COUNTER (KMCTR) [counter mode]

• CIPHER MESSAGE WITH OFB (KMO) [output feedback mode]

Each of these instructions provides a common suite of functions listed. For each basic type of function, there is a corresponding encrypted-key version.

MSA-X4 New Instructions

- PERFORM CRYPTOGRAPHIC COMPUTATION (PCC)
  - Functions:
    - Query
    - Compute last block CMAC using DEA
    - Compute last block CMAC using TDEA 128
    - Compute last block CMAC using TDEA 192
    - Compute last block CMAC using encrypted DEA
    - Compute last block CMAC using encrypted TDEA 128
    - Compute last block CMAC using encrypted TDEA 192
    - Compute last block CMAC using AES 128
    - Compute last block CMAC using AES 192
    - Compute last block CMAC using AES 256
    - Compute last block CMAC using encrypted AES 128
    - Compute last block CMAC using encrypted AES 192
    - Compute last block CMAC using encrypted AES 256
    - Compute XTS parameter using AES 128
    - Compute XTS parameter using AES 256
    - Compute XTS parameter using encrypted AES 128
    - Compute XTS parameter using encrypted AES 256

The fourth of the new MSA-X4 instructions describes the new PERFORM CRYPTOGRAPHIC COMPUTATION (PCC) instruction. This instruction provides the primitive operations to cipher-based-message-authentication-code mode and XTS mode.

MSA-X4 New Functions for Existing Instructions

- CIPHER MESSAGE (KM)
  - XTS AES 128
  - XTS AES 256
  - Encrypted XTS AES 128
  - Encrypted XTS AES 256
- COMPUTE INTERMEDIATE MESSAGE DIGEST (KIMD)
  - GHASH
- COMPUTE MESSAGE AUTHENTICATION CODE (KMAC)
  - AES 128
  - AES 192
  - AES 256

MSA-X4 also adds new functions to existing message-security-assist instructions.

For CIPHER MESSAGE (KM), functions supporting the XTS and encrypted XTS modes are provided.

For COMPUTE INTERMEDIATE MESSAGE DIGEST (KIMD), a function is provided in support of the Galois/counter mode hashing.

For COMPUTE MESSAGE AUTHENTICATION CODE (KMAC), three new advanced-encryption-standard (AES) functions are provided.

Miscellaneous Enhancements (1)

- **Fast BCR Serialization Facility**
  - **BCR 15,0 – performs serialization & checkpoint sync.**
  - **BCR 14,0 – performs serialization only**
  - **Installation of the fast-BCR-serialization facility (& al.) indicated by facility bit 45**
- **Enhanced-Monitor Facility**
  - **MONITOR CALL can now do counting without program interruption**
  - **Requires O/S-supplied counting array in home AS**
  - **Enabled by CR8 bits 16-31**
  - **Installation of the enhanced-monitor facility indicated by facility bit 36**
- **CMPSC-Enhancement Facility**
  - **Provides zero-padding control – may improve performance**
  - **Installation of the CMPSC-enhancement facility indicated by facility bit 47**

The enhancements described on this slide are changes to existing general instructions to provide improved performance or new function.

For as long as I can remember, the BRANCH ON CONDITION (BCR) instruction caused serialization and checkpoint synchronization to occur when the M_1 and R_2 fields of the instruction contain 1111 and 0000 binary, respectively. Without getting into tedious details of machine-check recovery, there may be situations where a programs wants to effect a serialization operation, but doesn't care about checkpoint synchronization. A new form of BCR will cause serialization only when the M_1 and R_2 fields of the instruction contain <u>1110</u> and 0000 binary, respectively.

MONITOR CALL provides a means by which a program can – with operating-system assistance – cause monitor-event program interruptions to occur during the execution of a program. The O/S can use these interruptions to count, measure, or otherwise observe the execution of the program. If the O/S does not enable the monitor class specified in the MC instruction (via control register 8), the instruction is effectively a no-op. This type of program measurement is expensive and tends to perturb the condition being measured. The enhanced-monitor facility provides a means by which MONITOR CALL can be used to effect the counting of events in a program – without a program interruption and (other than set-up of a counting array) without operating-system intervention.

COMPRESSION CALL is performed by a specialized component in the CPU that operates best when processing – and storing – data in larger chunks than just a byte. A new zero-padding control on the CMPSC instruction allows the instruction to operate in this more efficient manner when storing the last bytes of a result. The default zero-padding-control value of zero causes the CMPSC instruction to operate as originally defined to ensure complete compatibility with the original architecture, however we recommend that all users of CMPSC set the zero-padding control to one for potential improved performance.

The enhancements listed on this slide are all tweaks to control instructions.

As originally defined, INVALIDATE PAGE TABLE ENTRY (IPTE) sets the invalid bit to one in a PTE, and then signals all CPUs in the configuration to purge (at least) that entry from their translation-lookaside buffers (TLBs). Signaling and waiting for the acknowledgement of the TLB-purging was a time-consuming operation, especially if a large number of PTEs were being invalidated in bulk. The IPTE-range facility provides a new operand to the instruction that designates the number of PTEs to be invalidated. This allows the instruction to signal other CPUs to invalidate a block of contiguous PTEs, rather than once per PTE.

The SET STORAGE KEY (SSKE) instruction signal other CPUs of changes to the storage key to ensure that all CPUs observe a consistent key value. A signal to change the key may cause other CPUs to become quiesced to ensure that it is not accessing the storage in which the key is being changed. However, in certain situations the O/S may be able to ensure that other CPUs are not accessing the block (e.g., when a block is not mapped to a virtual address space). In such situations, performance may be improved by bypassing the quiesce operation. The nonquiescing-SSKE provides a new control to the SSKE instruction to cause the quiescing operation to be skipped. For compatibility purposes, the default (0) value of the control is to cause quiescing.

The reset-reference-bits-multiple facility provides a new control to the RESET REFERENCE BITS EXTENDED (RRBE) instruction, to allow it to reset the reference bits for multiple contiguous blocks of real storage with one execution of the instruction.

Summary

- The IBM zEnterprise 196 provides a broad range of new facilities to improve performance and function:
  - High-word facility (30 instructions)
  - Interlocked-access facility (12 instructions)
  - Load/store-on-condition facility (6 instructions)
  - Distinct-operands facility (22 instructions)
  - Population-count facility (1 instruction)
  - Enhanced-floating-point facility (25 new, 30 changed instructions)
  - MSA-X4 facility (4 new, 3 changed instructions, new functions)
  - Etc.
- Potential for:
  - Significant performance improvement
  - Enhanced capabilities
  - Simpler code

The old saw about effective presentations states, “tell them what you’re going to say, say it, tell them what you’ve just said!” We’re at the third point of that teaching, and as the past 99 slides illustrate, I’ve said a lot (or if you’re reading these slides, you’ve read a lot).

The IBM zEnterprise introduces a wide variety of new CPU facilities – some of which are simply designed to provide new or extended functions – however most of these facilities are designed to provide improved performance.

There is the potential that in exploiting these new instructions, significant performance improvement may be realized. The high-word and distinct-operand facilities may provide register-constraint relief to certain applications. The interlocked-access and load-and-store-on-condition facilities may provide reduced instruction path length – the interlocked-access facility is particularly useful in MP applications.

The enhanced-floating-point facility provides additional function for floating-point applications, and the MSA-X3 and MSA-X4 facilities provide powerful operations for cryptographic and security applications.

In addition to improved performance and function, exploitation of these facilities may yield simpler code paths, thus making program execution faster and program debugging easier.

Questions?

SHARE 116

103

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.