IBM z/Architecture CPU Features
A Historical Perspective

Dan Greiner
dgreiner@us.ibm.com
z/Server Architecture
SHARE 117 in Orlando
Session 9220, 10 August 2011, 3:00 pm

IBM Systems and Technology Group (STG)

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.

Topics du Jour:

- A brief history of the CPU architecture leading up to System Z
- Facilities added since the introduction of z/Architecture in 2000
  - z900 (2064) and z800 (2066):
  - z990 (2084) and z890 (2086):
  - z9-109 EC (2094) and z9-109 BC (2096)
  - z10 EC (2097) and z10 BC (2098)
  - z196 (2817)
- Why the new facilities were implemented
- Mostly CPU facilities discussed, not I/O

This presentation will review new CPU facilities that have been added to z/Architecture since its introduction in 2000.

The facilities will be reviewed in (semi) chronological order of introduction, based on the availability of new machine levels.

This presentation deals mostly with CPU facilities … not I/O.

The presentation does not go into bit-level detail of each new instruction. Rather, the presentation attempts to explain why the various facilities were introduced, and how a program might go about exploiting them.

Please refer to my past SHARE presentations on the z9, z10, and z196 for a more detailed description of the recent additions to z/Architecture, or, refer to the most recent *z/Architecture Principles of Operation* (SA22-7832-08).

In the Beginning … System 360 (1964)

- Provided 32-bit arithmetic
  - 16 general-purpose registers
- Provided 24-bit addressing (16 megabytes max.)
  - Real addressing only! No virtual memory
  - More than a few megabytes was quite rare
- Approximately 142 instructions total
  - Some features were optional
    - Decimal instructions (in-storage only)
    - Direct control (specialty I/O for check sorters, &c.)
    - Floating point (with 4 floating-point registers)
    - Protection feature (i.e., storage keys)
- I/O architecture provided a maximum of 7 channels
  - Most machines had far fewer (e.g., 1 byte multiplexer, 2 selector)
  - Enabled by mask bits in the PSW.

Nearly 50 years ago, IBM introduced the System 360 series of processors, providing a common architecture across all models of the product line.

The basic characteristics of the architecture are shown on this slide:

• 32-bit arithmetic using a set of sixteen 32-bit general registers.

• 24-bit real storage addressing, providing support for up to 16 megabytes of main storage. Although seemingly small by 2011 standards, in 1964, 16 megabytes was mind-bogglingly huge … at approximately $1.00 per byte, this was far more than most customers could afford, and physically it would have required hundreds of square feet of costly raised-floor space in the machine room.

Also recall that the original S/360 did not include dynamic address translation (DAT), thus there was no virtual memory. This necessitated a very frugal approach to programming (which seems to have fallen out of fashion in contemporary programming languages), and required some rather awkward programming conventions such as overlays.

The initial *IBM System/360 Principles of Operation* (A22-6821-0) described 141 instructions, however many of these instructions represented optional features in the original machines. One of the features that made the architecture superbly reliable – that of storage-protection keys – was optional in the original S/360!

The number of I/O channels was limited by interruption-enablement bits in the program-status word (PSW): one byte-multiplexer channel (to which unit-record devices such as card readers, punches, printers, and terminals were attached), and up to six selector channels (to which disk and tape devices were attached).

System 370 (1971)

- Provided virtual addressing
  - One or multiple 24-bit virtual spaces
  - Still limited to 24-bit real storage addressing
    - Most CPUs still had (far) less than 16 M-bytes
- Introduced new 16 control registers
- Introduced 13 new instructions
  - Load/store control registers
  - Compare / move long
  - Compare / insert / store character under mask
  - &c.
- I/O subsystem expanded to (up to) 16 channels
  - 32 channel option on some later machines
- Program-event recording
- Monitor-event recording

In 1971, IBM introduced the System 370, continuing the instruction set from the original S/360 (and making many of the optional features standard).

The S/370 added virtual addressing, providing 24-bit addressability which allowed up to 16 megabytes of virtual memory to be accessed with no special tricks required by the application program. Initially, IBM's Single Virtual Storage (SVS) operating system provided a single 16 M-byte virtual space, however soon thereafter, the Multiple Virtual Storage (MVS) operating system provided many 16 M-byte virtual spaces. The CPU was still limited to 24-bit real addresses, thus the maximum amount of real memory still remained 16 M-bytes or less … again, most customers had far less real storage.

One of the initial advantages of virtual storage was the ability to over-commit real storage … potentially stuffing hundreds of megabytes of virtual data into a 16 M-byte (maximum) bag. However, as real storage has become increasingly more affordable over the years (decades, actually), over-committing real storage is of less importance … and many z/OS sites perform very little paging.

The S/360's storage-protection key feature allowed the segregation of data in real storage into one of 16 classes, the access to which was enforced by the hardware. Although a bane to novice programmers who encounter unexpected protection exceptions (that is, a program-interruption code 0004, which z/OS and its predecessors translate into an S-0C4 abend code), the storage-protection keys are one of the features that contribute to the architecture's incredible reliability. Virtual storage – and the ability to segregate data into any number of separate address spaces – is another feature that operating systems can exploit to increase reliability.

To implement virtual storage – and several other features – the architecture introduces 16 new control registers that could be directly manipulated by the control program. This gave the control program much more flexibility in the control of the machine, the virtual context, the enablement of I/O operations (now supporting up to 16 channels), and other new features such as program-event recording (PER).

The S/370 also introduced several new instructions … many of which were used by the control program to manage the hardware, but some of which became instant favorites: MOVE LONG and COMPARE LOGICAL LONG, to name a few. However, you will notice that there were relatively few new instructions (at least, relative to the last decade).

System 370 Enhancements (1978-1982)

- 26-bit real addressing (up to 64 MB)
  - Single virtual address space still limited to 24 bits (16 MB)
- Various specialty instructions
  - MVS assist instructions for obtaining / releasing locks, tracing, SVC assist, FRR manipulation, &c.
  - Mathematical assists (transcendental functions, &c.)
  - High-accuracy arithmetic
- Dual-address space
  - Primary & secondary spaces
  - New instructions:
    - PROGRAM CALL / PROGRAM RETURN
    - INSERT / SET ADDRESS SPACE CONTROL
    - MOVE TO PRIMARY / SECONDARY

In the late 1970s and early 1980s, additional features were added to the S/370 architecture.

26-bit real addressing allowed the machine to exploit 64 megabytes of real storage (after almost 14 years, the machines broke the 16 M-byte real boundary). The virtual space was still limited to 16 M-bytes, but since the operating system could support multiple spaces, 16 M-byte virtual was not considered to be a significant limitation (at the time).

Various special assist instructions were added for use by the MVS operating system. One of the performance bottlenecks in MVS was the tracing of key system functions … special tracing instructions were added to improve the performance of these operations. Another bottleneck was the obtaining and releasing of global and local locks … four specialized instructions were added to address this limitation. Two specialized instructions were added to assist the operation of supervisor-call (SVC) instructions, and adding functional-recovery routines. All of these instructions were privileged operations, available only to the control program.

Around 1981, IBM added the dual-address-space feature … a means by which the program could, with special instructions, access two separate virtual address spaces with one instruction. A (relatively) primitive means of transferring control from a program running in one address space to another (and another … &c) and safely returning was provided, as well as instructions to inspect and set address-space modes. Again, all of these instructions were privileged (or semi-privileged), thus control program assistance was necessary for their use.

370 Extended Architecture (1983)

- 31-bit virtual addressing
  - Single address space providing up to 2 G-bytes
  - Bimodal addressing (1 bit of address stolen to designate 24- or 31-bit addresses)
- 31-bit real addressing (2 gigabytes max.)
  - Most customers still had much less memory
- Entirely new I/O subsystem
  - Up to 256 I/O channels
- New instructions for I/O, storage key manipulation, and program linkage
- Hardware tracing
  - Eliminated significant bottleneck in MP tracing

With the growth of operating-system, database, and application usage, that 16 M-byte virtual constraint (that wasn't such a significant thing back late '70s) had suddenly become a critical limitation by the early 1980s. The 370 Extended Architecture (370/XA) feature provided significant relief … providing 31-bit real and virtual addressing (that is, up to 2 G-bytes … 128x the 16 M-byte limit) while retaining complete compatibility with 24-bit applications.

Why only 31-bit? That was part of the compatibility requirement … the extra bit provided a means by which the program could switch back and forth between 24- and 31-bit addressing without any control-program intervention. As was the case when the original S/360 was introduced, 2 G-bytes of storage was not actually available at the time … the largest real storage sold in the mid 1980s was in the range of a few hundred megabytes.

The entire I/O architecture – now called the channel subsystem – was redesigned, providing support for up to 256 I/O channels (again, more than any customer had for years to come), and around 14 new I/O instructions.

Other new instructions provided for better storage-key management (now on a 4 K-byte boundary instead of 2 K-byte), better cross-memory program linkage, and CPU-specific hardware tracing that significantly reduced the overhead of tracing.

Enterprise Systems Architecture / 370 (1989)

- Introduced access-register translation (ART)
  - Provided the means by which a program could access multiple address spaces with minimum overhead
  - Up to 2,048 2 G-byte address spaces (4 T-bytes)
  - New ***nonprivileged*** instructions for manipulation of ARs
- Introduced the home address space
  - Location for principle task (process) control structures
  - Where to go (i.e., who to blame) when task ABENDs
- Introduced the linkage stack
  - Push-down stack for semi-authorized tasks
  - New instructions for manipulation of linkage stack.

The size of a real or virtual space is often referred to as a vertical limit or a ceiling, and dealing with that limit involved a “horizontal” solution … adding more virtual spaces. Recall that when the 16 M-byte boundary became an upper limit on addressability, one of the solutions was to introduce virtual memory, thus spreading this limit across multiple address spaces. 370/XA again raised the ceiling by providing a 2 G-byte limit, but in just a few short years, that was starting to seem tight. Additionally, the means of accessing data using the dual-address-space techniques was cumbersome and required constant intervention of the control program.

In 1989, IBM introduced the Enterprise Systems Architecture (ESA/370) which provided the access-register mode – a powerful means of dodging the 2 G-byte limit by again expanding horizontally, with up to 2,048 two G-byte space (a mind-boggling maximum of 4 terabytes!). With control program set-up, and very little intervention after that, application programs could easily manipulate storage in multiple address spaces without special semi-privileged instructions. And, the application could switch between any of the spaced it had been authorized to access.

ESA/370 also introduced the home address space, a special space designed to house principle task structures, such that when a program was multiple levels deep in calling programs in other address spaces – and an exception occurred – the control program would know to what task the fault should be attributed (that is, who gets ABENDed.

The program-call mechanism, introduced with the dual-address-space features back in 1981, were greatly expanded … including a hardware-managed linkage stack. [When the original S/360 was designed, (most of) the chief architects were opposed to any sort of push-down stack (thinking it was a characteristic of a microcomputer). Curious that 25 years later, ESA/370 introduces a hardware-supported linkage stack that does much the same thing.]

As has been shown on many of the preceding slides, most of the new features added to the S/360 architecture have thus far been features that the control program exploits. Access registers starts to break this monopoly on improvements, providing the application with a powerful set of new instructions for data access. As will be seen on the next slide, significant architecture changes are added that are available to the application (nonprivileged) programs.

Enterprise Systems Architecture / 390 (1990-1999)

- **Numerous new facilities:**
  - **Arithmetic instructions with 16-bit immediate operands**
  - **Branch instructions with 16-bit relative-branch location**
  - **Binary-floating-point instructions (IEEE standard)**
    - **95 new instructions and floating-point control register**
  - **Compression facility**
  - **Check-sum instruction**
  - **Extended-translation instructions (Unicode™ conversion)**
  - **Sorting-assist instructions**
  - **String-manipulation instructions**
- **Enhancements to other facilities (PC, PR, &c).**

During the 1990s, various facilities were added to the architecture to improve application programs:

A broad range of instructions having 16-bit immediate operands were added. The advantage was that a separate storage reference (other than the actual instruction itself) was not necessary to perform basic arithmetic and logical operations. For example, ADD IMMEDIATE allowed the addition of a signed 16-bit number to a register. This reduced storage access, thus minimizing unnecessary cache pollution by literal values.

Similarly, the relative-branching facility provided the means by which a program could perform forward or backward branches of up to 64K … relative to the current instruction address in the PSW (rather than from a base register). This provided a significant increase in the range of branching (classic branching was limited to an unsigned 12-bit displacement). The relative-branching facility provided base-register constraint relief, and, with judicious use, made it possible to write some programs that did not require a base register at all.

The binary-floating-point facility provided 95 new instructions that conformed to the emerging IEEE floating-point standard (common in microprocessor platforms). This provided increased precision, with a smaller exponent “step” than in the classic S/360 floating-point number representation (now called hexadecimal floating point).

Numerous other new features were added, including a hardware-assisted compression facility, a check-sum facility in support of TCP/IP protocols, instructions for performing translation of Unicode character representations, manipulation of character strings (including Unicode strings) and assisting with DFSORT utilities. Other features were added to the linkage instructions (including the new PROGRAM RETURN instruction) for improved reliability in cross-memory applications.

z/Architecture (2000)

- General registers grew to 64 bits
  - Existing 32-bit instructions retained
    - Used rightmost 32 bits of the 64-bit registers
  - Large suite of 64-bit analogues added
  - 163 new instructions (139 general, 11 control, 12 FP)
- Provides 64-bit virtual address space
  - Up to 16 exabytes in a single space
    - 18,446,744,073,709,551,616 bytes
  - With ART, up to 2,048 spaces (2^{75} bytes)
    - 37,778,931,862,957,161,709,568 bytes
  - Trimodal addressing: 24-, 31-, or 64-bit
- Provides 64-bit real
  - Current models offer a few terabytes

We're starting to see a trend: Addressing hits a ceiling (16 M-byte), and then expands horizontally (multiple address spaces). Addressing is expanded, and again hits a ceiling (2 G-bytes), and then expands horizontally (access-register mode).

With z/Architecture, the number of addressing bits is more than doubled (to 64 from 31), and the number of addressable bytes increases by a factor of 8 billion – all while retaining compatibility with existing 24- and 31-bit applications. Sixty-four bit addressing provides support for both real and virtual spaces of sixteen exabytes! And with access-register translation, that's a maximum of thirty seven sextillion, seven hundred seventy eight quintillion, nine hundred thirty one quadrillion, eight hundred sixty two trillion, nine hundred fifty seven billion, one hundred sixty one million, seven hundred nine thousand, five hundred sixty eight bytes.

As with earlier extensions, the application program can switch between 24-, 31-, and 64-bit addressing without control program intervention. Unlike most of the preceding architecture enhancements that favored control-program features, z/Architecture provides 139 new general instructions … most of which are 64-it analogs of existing 32-bit arithmetic and logical operations.

As seen with prior systems, current models of z/Architecture-capable machines offer much less than the maximum amount of memory addressable by the system.

New Facilities in z/Architecture 2000-2007

Thus far, we have had a whirlwind review of the architecture that lead to System Z. The following slides will discuss enhancements that have been added to z/Architecture over the past 10 years.

New Facilities in the Z800 & Z900 - Extended-Translation Facility 2:

- Performs operations on double-byte, ASCII, and decimal data.
- Provides support for manipulation of Unicode strings.
- Includes the following instructions:

Name	Mne-monic	Op-code
COMPARE LOGICAL LONG UNICODE	CLCLU	EB8F
MOVE LONG UNICODE	MVCLU	EB8E
PACK ASCII	PKA	E9
PACK UNICODE	PKU	E1
TEST DECIMAL	TP	EBC0
TRANSLATE ONE TO ONE	TROO	B993
TRANSLATE ONE TO TWO	TROT	B992
TRANSLATE TWO TO ONE	TRTO	B991
TRANSLATE TWO TO TWO	TRTT	B990
UNPACK ASCII	UNPKA	EA
UNPACK UNICODE	UNPKU	E2

The extended-translation facility (now called **extended-translation facility 1**) was introduced in ESA/390 in 1999. It included the CONVERT UNICODE TO UTF-8, CONVERT UTF-8 TO UNICODE and TRANSLATE EXTENDED instructions.

To assist with the increasing handling of multinational character encodings – particularly those found in web pages – the **extended-translation-facility 2** provided the additional instructions shown in this slide.

Two instructions -- COMPARE LOGICAL LONG UNICODE and MOVE LONG UNICODE -- provide similar function to that of COMPARE LOGICAL LONG EXTENDED and MOVE LONG EXTENDED, respectively, except that the former pair provides 2 byte padding and requires the length to be a multiple of two.

Four instructions – PACK ASCII, PACK UNICODE, UNPACK ASCII, and UNPACK UNICODE – provide an efficient means of packing and unpacking data having character representations other than EBCDIC.

The four TRANSLATE N TO N instructions support translating between single-byte and double-byte character encodings.

•Each instruction was designed to translate until a particular test character was encountered.

•The two TRANSLATE TWO TO N instructions required that the translation table be on a 4K boundary.

The two characteristics enumerated above were not viewed as attractive features by some developers, and the ETF2-Enhancement Facility was later added to address these issues.

TEST DECIMAL was somewhat of an orphan, packaged into this facility. It tests for a valid packed-decimal digits and sign.

New Facilities in the Z800 & Z900 - HFP Multiply-and-Add / Multiply-and-Subtract Facility:

- Provides improved performance for hexadecimal floating-point numbers
- May be used in place of MULTIPLY followed by ADD (or SUBTRACT)
- Key to many improved mathematics functions
- Includes the following instructions:

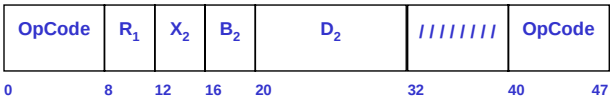
Name	Mne-monic	Op-code
MULTIPLY AND ADD	MAD	ED3E
MULTIPLY AND ADD	MADR	B33E
MULTIPLY AND ADD	MAE	ED2E
MULTIPLY AND ADD	MAER	B32E
MULTIPLY AND SUBTRACT	MSD	ED3F
MULTIPLY AND SUBTRACT	MSDR	B33F
MULTIPLY AND SUBTRACT	MSE	ED2F
MULTIPLY AND SUBTRACT	MSER	B32F

MULTIPLY AND ADD and MULTIPLY AND SUBTRACT appeared in the architecture in earlier machines in the form of mathematical assists. With the advent of the binary floating-point (BFP) facility in ESA/390, MULTIPLY AND ADD / SUBTRACT were included as standard BFP instructions.

To provide equivalent function for the hexadecimal floating-point (HFP) formats, MULTIPLY AND ADD / SUBTRACT instructions were provided for HFP.

New Facilities in Z800 & Z900 - Long-Displacement Facility (1):

- **Exploits previously-unused byte in RXE-format instruction:**
  - **RXE-format instructions introduced with binary-floating point in ESA/390**



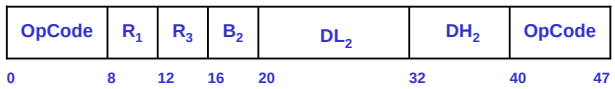
  - **RXE (and RSE) used extensively to implement z/Architecture opcodes (e.g., 64-bit instructions)**
  - **Bits 32-39 of the instruction reserved in the new formats**

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

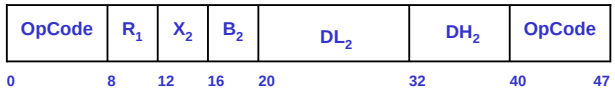
With the advent of z/Architecture, the RS instruction format was extended in a similar manner to form the RSE format. The RSE and RXE instruction formats were used extensively in implementing the new 64-bit architecture.

New Facilities in Z800 & Z900) - Long-Displacement Facility (2):

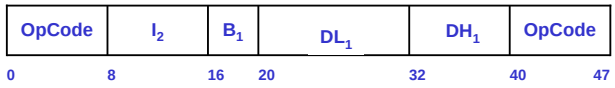
- **Extends 12-bit unsigned displacement to 20-bit signed displacement:**
  - **RSY:**



  - **RXY:**



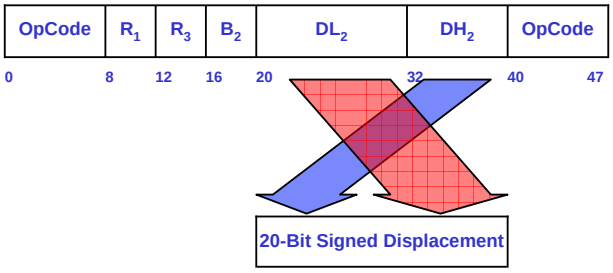
  - **SIY:**



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

New Facilities in Z800 & Z900 - Long-Displacement Facility (3):

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



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

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

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

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

New Facilities in Z800 & Z900 - Long-Displacement Facility (4):

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

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

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

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

New Facilities in Z800 & Z900 - Long-Displacement Facility (5):

- **Advantages of long displacement**
  - **Reduce the number of base registers required to address data**
  - **Allows for non-zero-based structures**
    - **Structures with prefix**
    - **Certain stack models**
  - **Opportunity for significant performance improvement**
    - **Packing chained structured together**
    - **Reduced address-generation interlocks (AGI)**
- **WARNING: Performance of long-displacement facility on Z800 & Z900 is suboptimal!**

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

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

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

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

New Facilities in z890 / z990:

- DAT-Enhancement Facility 1 (June 2003)
- High-Performance Long-Displacement Facility (June 2003)
- Message-Security Assist (June 2003)
  - Five new instructions to perform CPU crypto operations
  - Five query functions
  - Two functions for message digest based on secure hash algorithm (SHA-1)
  - Data-encryption-algorithm (DEA) functions
  - More to follow in future machines

The z890 and z990 introduced a high-performance version of long-displacement. As discussed earlier, the version of long displacement provided on the z800 and z900 was done in Millicode.

The z890 and z990 also introduced the message-security assist, comprising five new multi-function instructions. Each instruction includes a query function that reports which additional functions are implemented for the instruction.

The basic assist includes a basic data-encryption-algorithm (DEA) function two secure-hash-algorithm functions (SHA-1). The assist is designed such that additional functions can be added in the future. A subsequent slide provides more details.

New Facilities in z890 & z990 GA3:

- **Extended-translation facility 3 (May 2004)**
  - ►Performs operations on Unicode and Unicode-Transformation-Format (UTF) characters
  - ►Provides right-to-left TRT

Name	Mne-monic	Op-code
CONVERT UTF-16 TO UTF-32	CU24	B9B1
CONVERT UTF-32 TO UTF-16	CU42	B9B3
CONVERT UTF-32 TO UTF-8	CU41	B9B2
CONVERT UTF-8 TO UTF-32	CU14	B9B0
SEARCH STRING UNICODE	SRSTU	B9BE
TRANSLATE AND TEST REVERSED	TRTR	D0

- **ASN-and-LX-reuse facility (May 2004)**
  - ►Allows safe reuse of ASN
  - ►Expands PC Number

Name	Mne-monic	Op-code
EXTRACT PRIMARY ASN & INSTANCE	EPAIR	B99A
EXTRACT SECONDARY ASN & INSTANCE	ESAIR	B99B
PROGRAM TRANSFER W / INSTANCE	PTI	B99E
SET SECONDARY ASN W / INSTANCE	SSAIR	B99F

The extended-translation facility 3 provides additional instructions for processing for multiple-byte character representations such as Unicode.

In addition to the two Unicode conversion instructions added by the extended-translation facility 1, four additional instructions are added, providing a full complement of conversion options between 8-bit, 16-bit, and 32-bit Unicode representations. The two ETF-1 instructions CUTFU and CUUTF are given new mnemonics for consistency with the four new instructions.

SEARCH STRING UNICODE (SRSTU) provides an analog to the SEARCH STRING (SRST) instruction, except that SRSTU searches for a two-byte pattern.

TRANSLATE AND TEST REVERSE (TRTR) provides an analog to TRANSLATE AND TEST (TRT), except that TRTR operates in a right-to-left manner.

The ASN-and-LX-reuse facility (ALRF) extends the usability of address-space-number (ASN) and linkage-index (LX) translations by providing a separate instance number of (ASTEIN or LSTESN). Prior to ARLF, ASNs and LXs could not safely be reused due to the risk of a stale value being translated. The instance number (along with new checking and exceptions) assures that the ASN and/or LX can be reused.

The facility also expands the size of the PC number used by PROGRAM CALL to a 32-bit number (from a 20 bit number).

New Facilities in z9-109:

- Compare-and-swap-and-store facility
- Conditional-SSKE facility
- DAT-Enhancement Facility 2 (LPTEA)
- Decimal-floating-point facilities
- ETF2-enhancement facility
- ETF3-enhancement facility
- Extended-immediate facility
- Extract-CPU-time facility
- HFP-unnormalized extension
- Message-security-assist extension 1
- Move-with-optional-specifications facility
- Store-clock-fast facility
- Store-facility-list-extended facility
- TOD-clock-steering facility

The z9-109 series of processors introduced a substantial number of new facilities as enumerated on this slide. Each of these will be discussed in more detail on subsequent slides.

New Features in z9-109 GA2 - Decimal Floating Point (1):

- **Why decimal floating point?**
  - **Given the normal supply of fingers, most people have a very particular affinity for base-10 arithmetic**
  - **Imprecise hex/binary representation of common decimal fractions (like $^{1}/_{10}$)**
  - **Accurate conversions between decimal and hex/binary is tricky**
    - **Rounding even trickier**
  - **Emerging standard, expected to have impact on web applications**
- **What's needed?**
  - **Intuitive arithmetic**
  - **Exact representation of most decimal numbers**
  - **Better handling of rounding**
  - **Handle integers, fixed point, and floating point**

22

Hexadecimal floating point has prevailed for over 43 years; the "new" binary floating point was introduced approximately 10 years ago. Why then, do we need yet another floating-point number representation?

Among other things, it comes down to the normal allotment of fingers – base 10 arithmetic is the form in which most people are trained. Other compelling factors include:

- There are imprecise binary representations of very common decimal fractions such as 1/10.
- Accurate conversion between binary formats (used by the CPU) and decimal formats (used in external representation) is tricky; rounding is even trickier.
- Decimal floating point (DFP) is an emerging standard that is used in web applications today.

Decimal floating point provides for more intuitive arithmetic processing, more exact representation of most decimal numbers, better (well, more options for) rounding, and a means by which integers, fixed-point and floating-point can be represented.

But from the machine's perspective, it's a little trickier …

New Features in z9-109 GA2 - Decimal Floating Point (2):

- **DFP number representation:**
  - **value = $(-1)^{sign}$ x coefficient x $10^{exponent}$**
- **Truly bizarre encoding:**
  - **Sign (S)**
  - **Combination field (CF)**
    - **Two bits of biased exponent**
    - **Leftmost digit (LMD) of coefficient**
    - **Encoding of Infinity and Not-a-Number (NaN)**
      - **All 3 of the above mashed into 5 bits**
    - **Biased exponent continuation field (BXCF)**
      - **Remainder of biased exponent**
      - **Includes controls for quiet or signaling NaNs**
      - **Size depends on precision**
  - **Encoded Trailing Significand (ETS)**
    - **Remaining portion of the coefficient**
    - **Size depends on precision**

The number representation of a DFP number is as shown on this slide:

value = $(-1)^{sign}$ x coefficient x $10^{exponent}$

The encoding of a DFP value is rather unusual too, as shown on the slide.

▪Perhaps the most recognizable field is a sign bit.

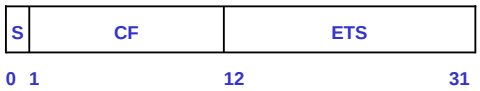
▪The sign bit is followed by a five bit field, aptly named the combination field – a highly encoded entity containing three separate components, including the leftmost digit (LMD) of the DFP number, two bits of the biased exponent field (more to follow), and an encoded indication of infinity or not-a-number (NaN).

▪Depending on the format of the DFP value, the biased exponent continuation (BXC) field contains either 6, 8, or 12 remaining bits of biased exponent. In more recent versions of the architecture, the BXC field is considered to be a part of the combination field.

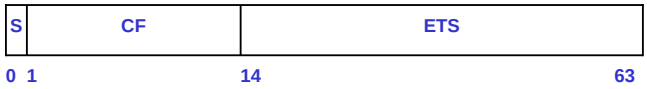
▪Depending on the format of the DFP value, the encoded trailing significant (ETS) field contains either 20, 50, or 110 additional bits of the coefficient. This is cleverly encoded such that three decimal digits fit within 10 bits (with 24 possible decimal aliases).

New Features in z9-109 GA2 - Decimal Floating Point (2):

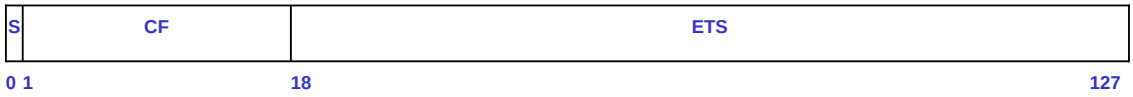
- **Short Format:**



- **Long Format:**



- **Extended Format:**



- **Encoded-trailing-significand (ETS) field encoded with 3 decimal digits in 10 bits**

The three formats of a DFP floating-point number are shown on this slide.

Note that both the combination field and the encoded-trailing-significand field vary in size from one format to another.

New Features in z9-109 GA2 - Decimal Floating Point (3):

Summary of DFP Formats

Property	Short	Long	Extended
Format length (bits)	32	64	128
Combination length (bits)	11	13	17
Encoded trailing significand length (bits)	20	50	110
Precision (digits), p	7	16	34
Maximum left-units-view (LUV) exponent (Emax)	96	384	6144
Minimum left-units-view (LUV) exponent (Emin)	-95	-383	-6143
Left-units-view (LUV) bias	95	383	6143
Maximum right-units-view (RUV) exponent (Qmax)	90	369	6111
Minimum right-units-view (RUV) exponent (Qmin)	-101	-398	-6176
Right-units-view (RUV) bias	101	398	6176
Maximum biased exponent	191	767	12,287
Largest (in magnitude) normal number, Nmax	$(10^7 - 1) \times 10^{90}$	$(10^{16} - 1) \times 10^{369}$	$(10^{34} - 1) \times 10^{6111}$
Smallest (in magnitude) normal number, Nmin	1×10^{-95}	1×10^{-383}	1×10^{-6143}
Smallest (in magnitude) subnormal number, Dmin	1×10^{-101}	1×10^{-398}	1×10^{-6176}

This slide enumerates the various characteristics of the three DFP formats.

New Features in z9-109 GA2 - Decimal Floating Point (4):

- **DFP Facility:**
  - **54 problem-state instructions for:**
    - **Adding**
    - **Comparing**
    - **Converting to / from fixed, signed BCD, unsigned BCD**
    - **Dividing**
    - **Extracting exponent, significance**
    - **Loading and testing, lengthening, rounding**
    - **Multiplying**
    - **Quantizing**
    - **[Re]rounding**
    - **Shifting**
    - **Subtracting**
    - **Testing data classes**
    - **Testing data groups**

The basic DFP facility includes 54 problem-state instructions that perform the operations enumerated on this slide. These instructions were added to the *z/Architecture Principles of Operation* (SA22-7832-05) in Chapter 20. Note that numerous other changes were made to Chapters 9, 18, and 19 in support of DFP.

The majority of DFP instructions are of one of the register-to-register formats (RRE, RRF or RRR), operating on values that have been preloaded into floating-point registers. Exceptions are the SHIFT instructions (RXF format) and the TEST DATA CLASS / GROUP instructions (RXE format).

New Features in z9-109 GA2 - Decimal Floating Point (5):

- **Control via:**
  - **Additional-floating-point (AFP) control (CR0.45)**
  - **Floating-point-control (FPC) register**
    - **New DFP rounding-mode control with 8 options.**
- **z9-109 implementation done in Firmware (with hardware assists)**
  - **Compared to web-based emulations written in C (or Java), performance is superior.**
  - **All-hardware implementation in System z10 and above**

As with the binary-floating-point facility, control of DFP is by means of the additional-floating-point (AFP) control (bit 45 of control register 0) and the floating-point-control (FPC) register.

A new DFP rounding mode has been added to the FPC register (bits 25-27), with 8 rounding options:

000 – round to nearest with ties to even

001 – round toward 0

010 – round toward positive infinity

011 – round toward negative infinity

100 – round to nearest with ties away from zero

101 – round to nearest with ties toward zero

110 – round away from 0

111 – round to prepare for shorter precision

The initial (z9-109) implementation of DFP is done in processor firmware using specialized hardware assists. Although this is not as fast as an all-hardware implementation, it is significantly faster than web-based emulations written in Java or C++. Future implementations are planned to be all hardware.

New Features in z9-109 GA2 - DFP Support Facilities:

- **DFP Rounding Facility:**
  - **SET DFP ROUNDING MODE instruction (and associated rounding modes in FPC register).**
- **Floating-Point-Support-Sign-Handling Facility:**
  - **COPY SIGN**
  - **LOAD COMPLEMENT**
  - **LOAD NEGATIVE**
  - **LOAD POSITIVE**
- **FPR-GR-Transfer Facility:**
  - **LOAD FPR FROM GR**
  - **LOAD GR FROM FPR**
- **IEEE-Exception-Simulation Facility:**
  - **SET FPC AND SIGNAL**
  - **LOAD FPC AND SIGNAL**
- **PERFORM FLOATING POINT OPERATION (PFPO)**
  - **Conversion to/from HFP / BFP / DFP**
  - **Numerous rounding-mode options**

Several floating-point-support facilities were introduced coincident with the DFP facility, as shown on this slide. These are all documented in Chapter 9 of the *z/Architecture Principles of Operation* (SA22-7832-05).

As mentioned in the notes for slide 23, the majority of DFP instructions operate on floating-point registers. The standard floating-point-support LOAD instructions – LD, LDY, LE, and LEY – can be used to load the FP registers from storage; LDR, LER, and LXR can be used to copy from one FPR to another. The FPR-GPR transfer facility provides the means by which data can be interchanged between FP registers and general registers.

The IEEE-exception-simulation facility provides a means of testing IEEE exception conditions by software simulation – without actually forcing the arithmetic condition to occur.

New Features in z9-109 - Extended-Immediate Facility (1):

- **Adds numerous 32-bit immediate-operand instructions**
  - ADD IMMEDIATE (AFI, AGFI)
  - ADD LOGICAL IMMEDIATE (ALFI, ALGFI)
  - AND IMMEDIATE (NIHF, NILF)
  - COMPARE IMMEDIATE (CFI, CGFI)
  - COMPARE LOGICAL IMMEDIATE (CLFI, CLGFI)
  - EXCLUSIVE OR IMMEDIATE (XIHF, XILF)
  - INSERT IMMEDIATE (IIHF, IILF)
  - LOAD IMMEDIATE (LGFI)
  - LOAD LOGICAL IMMEDIATE (LLIHF, LLILF)
  - OR IMMEDIATE (OIHF, OILF)
  - SUBTRACT LOGICAL IMMEDIATE (SLFI, SLGFI)
- **Minimizes need for constants & literal pools**

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

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

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

New Features in z9-109 - Extended-Immediate Facility (2):

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

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

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

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

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

New Features in z9-109 GA2 - Extract-CPU-Time Facility:

- **Original STORE CPU TIMER (STPT) is a privileged instruction**
  - **z/OS TIMEUSED service routine:**
    - **Extracts CPU timer for problem-state programs**
    - **Takes hundreds of CPU cycles (PC, lock, disable, observe, enable, unlock , PR)**
    - **Significantly skews measurements**
- **EXTRACT CPU TIME (ECTG) instruction:**
  - **Problem-state instruction**
  - **Can provide most data provided by TIMEUSED**
  - **Substantially less overhead**
  - **May facilitate better measurement of module flow, instruction sequences, micro-accounting, &c.**
  - **Requires access to SCP-maintained fields:**
    - **Task-time used**
    - **CPU timer at last dispatch**
    - **Scaling factor for secondary CPUs.**

Accurate observations of CPU utilization require the inspection of the CPU timer. However, the STORE PROCESSOR TIMER (STPT) instruction is privileged and is not available to the application program. The primary reason for this restriction is that, without serialized observation of other data maintained by the operating system, the results of STPT are meaningless. This is because, between one observation and the next, the program may be interrupted, and the operating system may alter the contents of the CPU timer without the program's knowledge.

z/OS provides the TIMEUSED service for extracting a task's accumulated CPU time, however older versions of the service routine are invoked by means of PROGRAM CALL and consumes hundreds of cycles. For a program that needs to measure the CPU time of a small code fragment, the overhead of the TIMEUSED service may obscure what is actually being measured.

The EXTRACT CPU TIME (ECTG) instruction provides a means by which a problem-state application program can inspect the CPU timer and other SCP-maintained timer-related fields as a single unit of operation. This provides a task with a means of efficiently determining total elapsed CPU time from which other timing-related data can be derived.

The z/OS TIMEUSED service has been adapted to use the ECTG instruction; TIMEUSED may now be entered via a branch rather than PROGRAM CALL.

New Features in z9-109 - HFP Unnormalized Extension:

- Adds unnormalized versions of MULTIPLY and MULTIPLY AND ADD to hexadecimal floating point
- Useful in multi-precision and crypto applications
- All are RRF-format ops.

Name	Mne-monic	Op-code
MULTIPLY UNNORMALIZED	MYR	B33B
MULTIPLY UNNORMALIZED	MYHR	B33D
MULTIPLY UNNORMALIZED	MRLR	B339
MULTIPLY UNNORMALIZED	MY	ED3B
MULTIPLY UNNORMALIZED	MYH	ED3D
MULTIPLY UNNORMALIZED	MYL	ED39
MULTIPLY AND ADD UNNORMALIZED	MAXWR	B33A
MULTIPLY AND ADD UNNORMALIZED	MAWR	B33C
MULTIPLY AND ADD UNNORMALIZED	MAYWR	B338
MULTIPLY AND ADD UNNORMALIZED	MAXW	ED3A
MULTIPLY AND ADD UNNORMALIZED	MAW	ED3C
MULTIPLY AND ADD UNNORMALIZED	MAYW	ED38

Slide 5 discussed the addition of hexadecimal-floating-point (HFP) multiply-and-add and multiply-and-subtract instructions. Similar functions were already provided for binary floating point.

The HFP unnormalized extension facility adds unnormalized versions of HFP MULTIPLY and MULTIPLY AND ADD instructions.

New Features in z9-109 - Message-Security-Assist Extension 1:

- Adds two 256-bit secure-hash-algorithm (SHA) functions:
  - COMPUTE INTERMEDIATE MESSAGE DIGEST
  - COMPUTE FINAL MESSAGE DIGEST
- Adds two 128-bit advanced-encryption-standard (AES) functions:
  - CIPHER MESSAGE WITH CHAINING
  - CIPHER MESSAGE (sans chaining)
- Adds 64-bit pseudo-random-number-generation facility
  - CIPHER MESSAGE WITH CHAINING
- All are extensions to message-security assist added in z990 GA1.

As noted on slide 13, there may be future extensions to the Message-Security Assist. The MSA-Extension-Assist Extension 1 is the first of these.

MSA-X1 includes the following enhancements:

- For the CIPHER MESSAGE and CIPHER MESSAGE WITH CHAINING instructions, two 128-bit advanced-encryption-standard (AES) operations are added.
- For CIPHER MESSAGE WITH CHAINING, a 64-bit pseudo-random-number-generation facility is added.
- For COMPUTE INTERMEDIATE MESSAGE DIGEST and COMPUTE FINAL MESSAGE DIGEST, two 256-bit secure-hash-algorithm functions are added.

New Features in z9-109 GA2 - Move-with-Optional-Specifications Facility:

- MVCOS provides über MOVE CHARACTER
  - True length specified in a register (no need for EXECUTE)
  - Moves up to 4K in one execution
  - Moves from any address-space control (ASC) to any other
  - Moves from any key to any other
  - Key and ASC for source and destination may be explicitly program-specified or use current-PSW values
  - May be faster than MOVE LONG for 4K-byte moves, but …
  - Will likely be slower than executed MVC for < 256-byte move.
- Equivalent to MVCP, MVCS, MVCDK, MVCSK, MVCK
  - Except the above are limited to 256 bytes; MVCOS is not.
  - Available to problem-state code (subject to PSW key mask)

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

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

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

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

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

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

New Features in z9-109 - Store-Clock-Fast Facility:

- **STORE CLOCK FAST allows storing of full-resolution 64-bit clock**
  - **Same clock value may be seen by the same or other CPUs multiple times.**
  - **Should be used only by applications that can tolerate it**
    - **Duplicate time stamps means exact sequence cannot be determined in an MP environment!**
  - **STCKF (B27C hex) can replace STCK (B205 hex)**
- **TRACE instruction can also use STCKF format**
  - **Subject to control register 0, bit 32.**
- **Facility is not required if program switches to STORE CLOCK EXTENDED and TRACE (TRACG)**

STORE CLOCK FAST (STCKF) allows the storing of a full-resolution 64-bit clock value. The rules for monotonic advancement are relaxed such that the same time stamp may be stored multiple times by one or more CPUs. However, the clock will **never** appear to run backwards.

STCKF should only be used by applications that can tolerate observing the same clock value twice. The potential for duplicate TOD values being stored may mean that the program cannot determine which event occurred first (unless some other form of serialization is used).

A portion of the TOD clock is also stored in trace records created by the TRACE (32-bit) instruction. Subject to bit 32 of control register 0, TRACE can be directed to store a result as if STCKF had been used (when CR0.32 is zero, it's business as usual, thus assuring compatibility with earlier programs).

When STORE CLOCK EXTENDED is used (or the 64-bit TRACG), the store-clock-fast facility is unnecessary.

New Features in z9-109 - Store-Facility-List-Extended Facility:

- **Original z/Architecture STORE FACILITY LIST (STFL):**
  - **Privileged operation (requires supervisor state)**
  - **Stores a list of facility bits at real location 200 (C8 hex)**
    - **Inaccessible unless OS maps real frame 0 to virtual page 0**
      - ☺ **- Z/OS does**
      - ☹ **- Linux doesn't**
  - **Limited to 32 facilities (one word – potentially extendable)**
- **STORE FACILITY LIST EXTENDED (STFLE)**
  - **General instruction (problem state)**
  - **Stores the results in a program-specified location and length**
  - **Up to 16,384 facilities may be indicated**
  - **Maps the first 32 facilities the same as STFL**

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

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

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

STORE FACILITY LIST EXTENDED (STFLE) is intended to address all of these concerns, as follows:

- STFLE is a general instruction, thus any application can execute it.
- The results are stored in a program-specified location.
- Up to 16K of facility indications may be indicated (256 doublewords; 1 bit per facility indication).
- The first 32 facility indications are identical to that provided by STFL.

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

New Facilities in the System z10 2008

The following slides will discuss the following facilities that were introduced on the z800 and z900 processors:

- Extended-translation facility 2
- HFP Multiply-and-Add / Subtract
- Long Displacement

General Instructions Extension Facility

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

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

For presentation purposes, the 72 instructions of the GIEF are divided into several categories, based on the instructions' characteristics. In some cases, an instruction's category characteristics overlap – for example, PREFETCH DATA RELATIVE LONG, which is described in the cache-cognizance category – also has the relative-long-second-operand characteristic. When such ambiguity exists, the instruction is described under the category that represents a best fit.

Note: In subsequent slides, the notation "SSS…SSS" means that the operand is sign-extended to the left to match the size of the other operand. Similarly, notation "000…000" means that the operand is zero-extended to the left to match the size of the other operand.

General-Instructions Extension Facility: Cache Cognizance Instructions

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

Explanation:

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

Three instructions fall into the cache-cognizance category:

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

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

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

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

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

The COMPARE AND BRANCH instructions combine a compare operation and, if the specified condition is met, a branch operation, in a single instruction. When the specified condition is not met, execution continues with the next sequential instruction. Many forms of the instruction are provided based on these characteristics:

- Numeric attribute: signed versus unsigned
- Operand size: 32-bit versus 64-bit
- Second-operand location: register versus immediate field
- Branch designation: base and 12-bit displacement versus 16-bit signed relative

As can be seen in the subsequent slides, the instructions have a rich set of operands. For the instruction formats with the comparand (second operand) in an immediate field, there is only room for an 8-bit value.

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

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

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

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

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

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

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

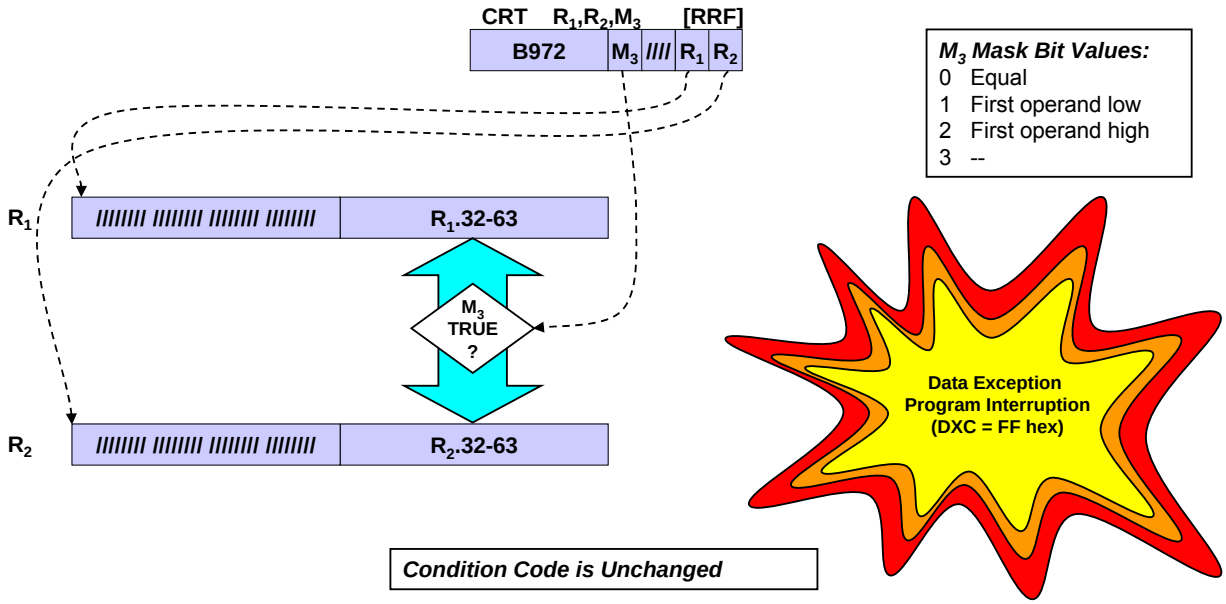
Because there is no branch location required (as in COMPARE AND BRANCH), the instruction formats with an immediate-field comparand provide a 16-bit value.

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

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

COMPARE AND TRAP (CRT)

(32-bit register signed operands)



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

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

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

General-Instructions Extension Facility: Immediate Second-Operand Instructions

Instruction	Mne-monic	Op-Code	First Operand			Second Operand		
			Location	Size	Sign	Location	Size	Sign
ADD IMMEDIATE	ASI	EB6A	S(20)	32	S	Immediate	8	S
ADD IMMEDIATE	AGSI	EB7A	S(20)	64	S	Immediate	8	S
ADD LOGICAL WITH SIGNED IMMEDIATE	ALSI	EB6E	S(20)	32	U	Immediate	8	S
ADD LOGICAL WITH SIGNED IMMEDIATE	ALGSI	EB7E	S(20)	64	U	Immediate	8	S
COMPARE HALFWORD IMMEDIATE	CHHSI	E554	S(12)	16	S	Immediate	16	S
COMPARE HALFWORD IMMEDIATE	CHSI	E55C	S(12)	32	S	Immediate	16	S
COMPARE HALFWORD IMMEDIATE	CGHSI	E558	S(12)	64	S	Immediate	16	S
COMPARE LOGICAL IMMEDIATE	CLHHSI	E555	S(12)	16	U	Immediate	16	U
COMPARE LOGICAL IMMEDIATE	CLFHSI	E55D	S(12)	32	U	Immediate	16	U
COMPARE LOGICAL IMMEDIATE	CLGHSI	E556	S(12)	64	U	Immediate	16	U
MOVE [IMMEDIATE]	MVHHI	E544	S(12)	16	S	Immediate	16	S
MOVE [IMMEDIATE]	MVHI	E54C	S(12)	32	S	Immediate	16	S
MOVE [IMMEDIATE]	MVGHI	E548	S(12)	64	S	Immediate	16	S
MULTIPLY SINGLE IMMEDIATE	MSFI	C21	Register	32	S	Immediate	32	S
MULTIPLY SINGLE IMMEDIATE	MSGFI	C20	Register	64	S	Immediate	32	S

A large variety of instructions are added with the second operand specified as an immediate field.

Of particular note is that for ADD IMMEDIATE, ADD LOGICAL WITH SIGNED IMMEDIATE, COMPARE HALFWORD IMMEDIATE, COMPARE LOGICAL IMMEDIATE, and MOVE (the first 15 instructions listed), the first operand is a ***storage location*** designated by a base register and 12-bit unsigned displacement field. More notes on ADD LOGICAL WITH SIGNED IMMEDIATE appear with that slide.

There are previously-existing forms of MULTIPLY with an immediate operand. The two forms shown here provide the immediate multiplier to the MULTIPLY SINGLE instructions.

General-Instructions Extension Facility: Relative-Long Second Operands (1 of 2)

Instruction	Mne-monic	Op-Code	First Operand		Second Operand	
			Location	Size	Location	Size‡
COMPARE RELATIVE LONG	CRL	C6D	Register	32	RL	32
COMPARE RELATIVE LONG	CGFRL	C6C	Register	64	RL	32
COMPARE RELATIVE LONG	CGRL	C68	Register	64	RL	64
COMPARE HALFWORD RELATIVE LONG	CHRL	C65	Register	32	RL	16
COMPARE HALFWORD RELATIVE LONG	CGHRL	C64	Register	64	RL	16
COMPARE LOGICAL RELATIVE LONG	CLRL	C6F	Register	32	RL	32
COMPARE LOGICAL RELATIVE LONG	CLGFRL	C6E	Register	64	RL	32
COMPARE LOGICAL RELATIVE LONG	CLGRL	C6A	Register	64	RL	64
COMPARE LOGICAL RELATIVE LONG	CLHRL	C67	Register	32	RL	16
COMPARE LOGICAL RELATIVE LONG	CLGHRL	C66	Register	64	RL	16

Explanation:

‡ Operand must be aligned on an integral boundary; otherwise, a specification exception is recognized!

RL Relative-long operand; 32-bit immediate value, multiplied by two and added to the current instruction address, provides the storage location of the operand

GIEF introduces variations of existing compare, load, and store instructions in which the second operand is specified as a relative-long storage location. As stated with PFDRL, relative-long storage specifications are new with the System z10 EC; they are similar to the relative and relative-long branch-address specifications, already present in the architecture.

For a relative-long storage specification, the immediate field is a signed value that designates the number of halfwords added to the current instruction address to form the address of the storage location (subject to the current addressing mode). More succinctly:

$$storage\_location = I_2 \times 2 + current\_instruction\_address$$

Because the I_2 field is signed, the storage location may precede the current instruction address if I_2 is negative. The storage location is always in the same address space from which instructions are being fetched.

See alignment notes accompanying the following slide.

General-Instructions Extension Facility: Relative-Long Second Operands (2 of 2)

Instruction	Mne-monic	Op-Code	First Operand		Second Operand	
			Location	Size	Location	Size‡
LOAD RELATIVE LONG	LRL	C4D	Register	32	RL	32
LOAD RELATIVE LONG	LGFRL	C4C	Register	64	RL	32
LOAD RELATIVE LONG	LGRL	C48	Register	64	RL	64
LOAD HALFWORD RELATIVE LONG	LHRL	C45	Register	32	RL	16
LOAD HALFWORD RELATIVE LONG	LGHRL	C44	Register	64	RL	16
LOAD LOGICAL RELATIVE LONG	LLGFRL	C4E	Register	64	RL	32
LOAD LOGICAL HALFWORD RELATIVE LONG	LLHRL	C42	Register	32	RL	16
LOAD LOGICAL HALFWORD RELATIVE LONG	LLGHRL	C46	Register	64	RL	16
STORE RELATIVE LONG	STRL	C4F	Register	32	RL	32
STORE RELATIVE LONG	STGRL	C4B	Register	64	RL	64
STORE HALFWORD RELATIVE LONG	STHRL	C47	Register	16	RL	16

Explanation:

‡ **Operand must be aligned on an integral boundary; otherwise, a specification exception is recognized!**

RL **Relative-long operand; 32-bit immediate value, multiplied by two and added to the current instruction address, provides the storage location of the operand**

Note Well! The original IBM S/360 imposed integral-alignment requirements for various storage operands. For example, a LOAD instruction required that the storage operand be aligned on a fullword boundary; otherwise, a specification exception was recognized. With the advent of the S/370, most of these restrictions were relaxed such that no particular alignment (usually) required.

With the GIEF instructions having relative-long second operands, the alignment restrictions are reinstated. The second operand for each of these instructions must be aligned on an integral boundary corresponding to the operand's size. For example, for LOAD RELATIVE LONG (LGRL), the second operand must be aligned on a doubleword boundary; for LRL, the second operand must be aligned on a fullword boundary; otherwise, a specification exception is recognized. Because the I_2 field designates a number of halfwords, second operands that are 16 bit in length are necessarily aligned on a halfword.

Why – after all these years of alignment emancipation – was such a restriction reintroduced? As we noted in the introduction, a primary goal of GEIF was high performance. The alignment of these operands is necessary to achieve the desired performance.

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

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

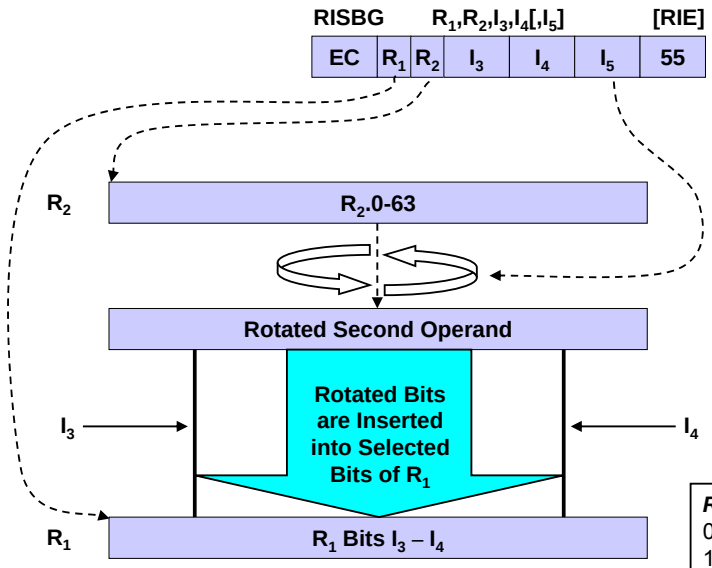
Explanation:

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

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

The following slides provide additional details.

ROTATE THEN INSERT SELECTED BITS (RISBG)



Remaining bits of R_1 either:

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

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

Resulting Condition Code:

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

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

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

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

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

General-Instructions Extension Facility: Miscellany

Instruction	Mne-monic	Op-Code	First Operand		Second Operand	
			Location	Size	Location	Size
COMPARE HALFWORD	CGH	E334	Register	64	S(20)	16
LOAD ADDRESS EXTENDED	LAEY	E375	Register	24/31/64	S(20)	N/A
LOAD AND TEST	LTGF	E332	Register	64	S(20)	32
MULTIPLY	MFY	E35C	Register	64←32	S(20)	32
MULTIPLY HALFWORD	MHY	E37C	Register	32	S(20)	16

Explanation:

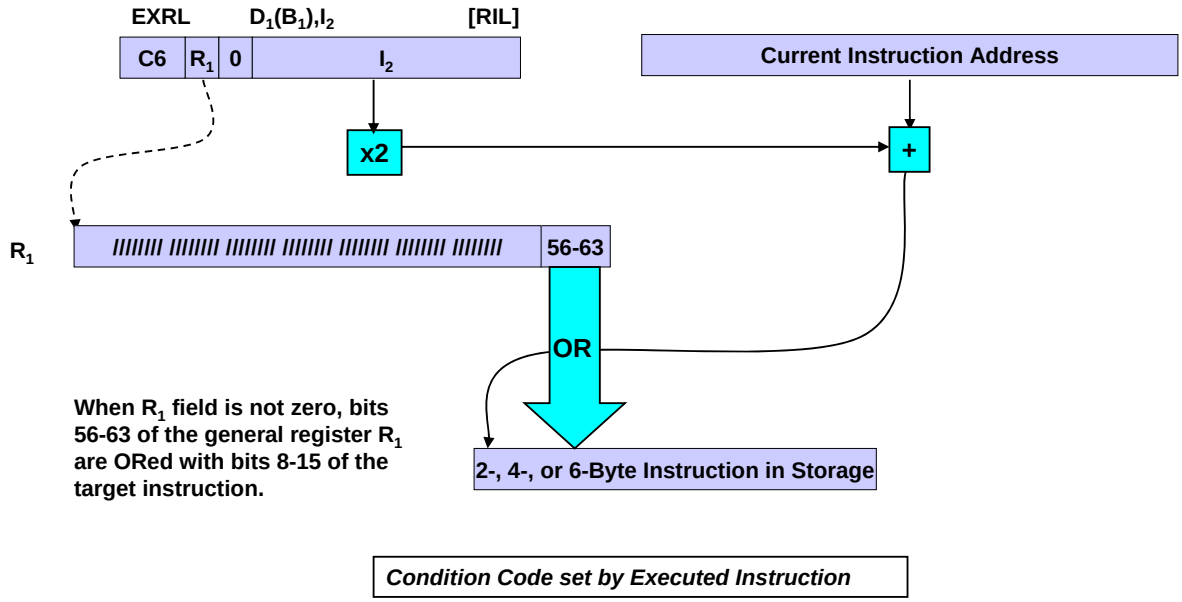
N/A	Not applicable
S(20)	Storage operand addressed using base, index, and 20-bit signed displacement.

GIEF includes a handful of miscellaneous other instructions which were added primarily to make the architecture (somewhat) more orthogonal with existing instructions, as shown on the following five slides.

There are no particular alignment requirements for the storage operands of any of these instructions.

Execute-Extensions Facility:

- Provides EXECUTE RELATIVE LONG (EXRL)



The execute-extensions facility provides the EXECUTE RELATIVE LONG (EXRL) instruction, which operates in a similar manner to that of the EXECUTE (EX) instruction.

EXECUTE (EX) uses a base and index register and a 12-bit unsigned displacement to designate the location of the target instruction.

EXECUTE RELATIVE LONG (EXRL) uses a relative-long storage specification, where the immediate field is a signed value that designates the number of halfwords added to the current instruction address to form the address of the storage location (subject to the current addressing mode). More succinctly:

storage_location = I_2 x 2 + current_instruction_address

Because the I_2 field is signed, the storage location may precede the current instruction address if I_2 is negative. The storage location is always in the same address space from which instructions are being fetched.

Parsing-Enhancement Facility:

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

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

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

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

Message-Security Assist Extension 2:

- **Message-Security-Assist Extension 2:**
  - **Adds advanced-encryption-standard (AES) functions for message ciphering**
    - **AES-192 algorithm**
    - **AES-256 algorithm**
    - **With or without chaining**
- **MSA SHA-512 Facility.**
  - **Adds two functions for generating intermediate- and final-message digests**
    - **SHA-512 algorithm**
    - **Also, SHA-384 (same as SHA-512 algorithm, but with different initial chaining values).**

The message-security assist was introduced in June 2003, with the message-security assist extension 1 added in September 2005.

The message-security assist extension 2 provides additional advanced-encryption-standard (AES) functions for 192- or 256-bit encryption and decryption.

Additionally, the secure-hash algorithm (SHA) has been extended to use the SHA-512 algorithm (512-bit).

Enhanced-DAT Facility:

- Current z/Architecture has 4K-byte page frames.
  - TLB entry for each translation
  - TLB space is very costly, thus limited scope with larger memory.
- Enhanced-DAT facility implements substantially-larger frame size
  - Similar features in other architectures
  - Implemented 1 megabyte frames
  - Other future possibilities, 2GB? 4TB??, 8PB???
- Assumes operating system does *not* do a lot of paging (segmenting?)
- New and changed control instructions:
  - PERFORM FRAME MANAGEMENT FUNCTION – clear frame and/or set key
  - LOAD PAGE TABLE ENTRY ADDRESS – Additional results returned.
  - LOAD REAL ADDRESS – Additional results returned.
  - SET STORAGE KEY EXTENDED - multiple-block mode (set up to 1M-byte)
- Facility enabled by control register 0, bit 40:
  - 0 – Classic DAT
  - 1 – Enhanced DAT enabled

Ever since 370/XA was introduced, the dynamic-address-translation (DAT) process has used 4 K-byte pages and 1 M-byte segments. z/Architecture added 2 G-byte regions. To improve the performance of DAT, a translation-lookaside buffer may cache the results of translations.

For systems that long-term fix large blocks of data (such as megabytes), maintaining TLB entries at the page (4 K-byte) level is inefficient. The enhanced-DAT facility addresses this by allowing DAT to result in a 4 K-byte page or 1 M-byte segment. This is accomplished by extensions made to the existing DAT-table entries, as shown on the following slide.

The facility modifies the behavior of the three control instructions listed, and introduces a new PERFORM FRAME MANAGEMENT FUNCTION, described shortly.

When the facility is installed, it may be enabled or disabled by a newly-defined bit in control register 0.

CPU-Measurement Facilities

- **Set-program-parameters facility**
  - **SET PROGRAM PARAMETERS (SPP) privileged operation**
  - **Used by the CPU-measurement sampling facility**
  - **Facility bit 40 indicates presence of the facility**
- **CPU-measurement facility**
  - **CPU-measurement counter facility**
  - **CPU-measurement sampling facility**
- **Documentation:**
  - ***The Set-Program-Parameter and CPU-Measurement Facilities*** **(SA23-2260-00),** http://publibfi.boulder.ibm.com/epubs/pdf/a2322600.pdf
  - ***The CPU-Measurement Facility Extended Counters Definition for z10*** **(SA23-2261-00),** http://publibfi.boulder.ibm.com/epubs/pdf/a2322610.pdf

The set-program-parameter facility and the CPU-measurement facilities were introduced in the second generation of the system z10 in November of 2008.

The set-program-parameter facility comprises the SET PROGRAM PARAMETERS instruction and one 64-bit program-parameter register that is used by the two components of the CPU-measurement facility: the counter facility and the sampling facility. Bit 40 of the facility list stored by STORE FACILITY LIST EXTENDED indicates the presence of the set-program-parameters facility.

The CPU-measurement facility has two sub-facilities – the the counter facility and the sampling facility – both of which are described on the subsequent slides.

These facilities are not described in the *z/Architecture Principles of Operation*, rather the architecture is described in two separate documents as listed on this slide. The documents are available only in soft copy from the web.

New Facilities in the IBM zEnterprise 196 (2010)

The following slides will discuss the following facilities that were introduced on the z800 and z900 processors:

- Extended-translation facility 2
- HFP Multiply-and-Add / Subtract
- Long Displacement

High-Word Facility (1)

- Suite of instructions to manipulate bits 0-31 of a 64-bit general-purpose register (GPR)
- For purposes of address-generation interlock (AGI), the 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.

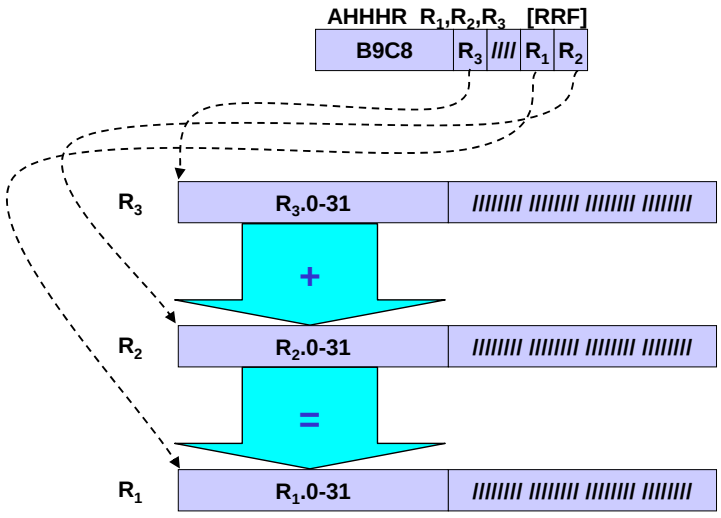
Also note that, of necessity, certain characters in the mnemonics have become a bit overloaded. The rookie programmer will likely find using the high-word facility challenging. We hope the benefits will be worth it.

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)



Resulting Condition Code:

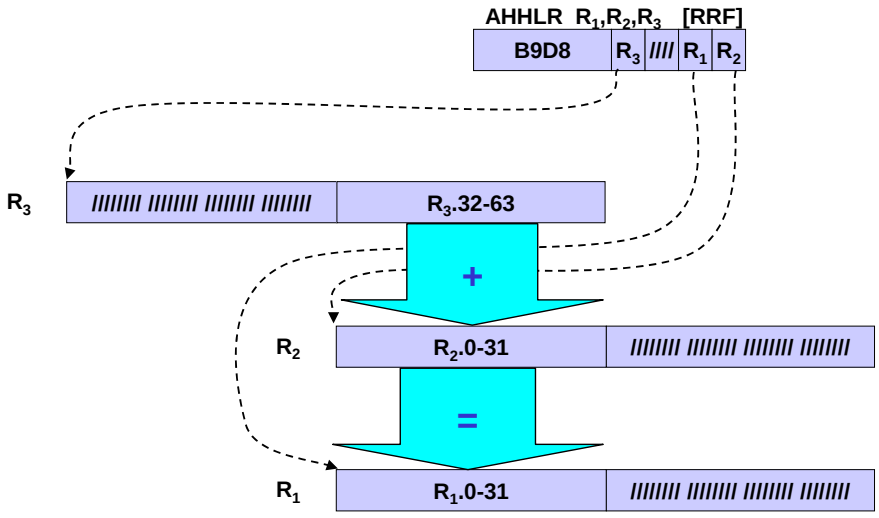
0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

For 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)



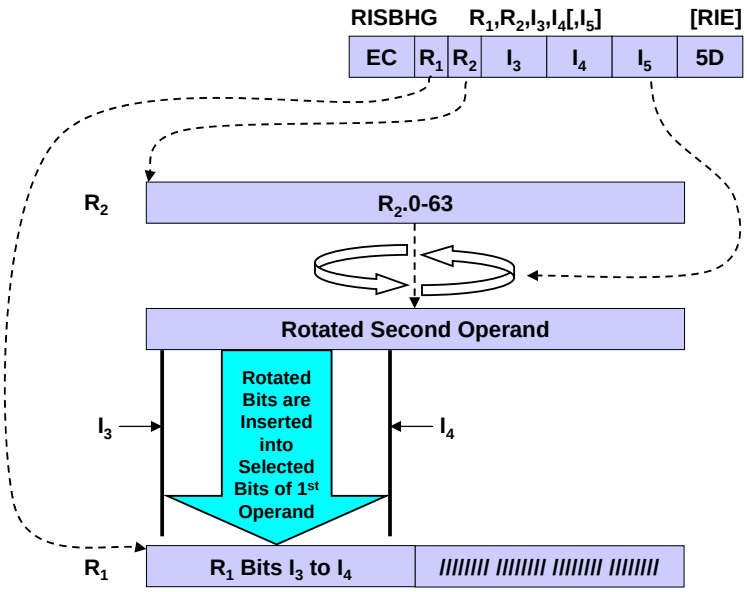
Resulting Condition Code:
0 Result zero; no overflow
1 Result less than zero; no overflow
2 Result greater than zero; no overflow
3 Overflow

ADD 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.

ROTATE THEN INSERT SELECTED BITS HIGH (RISBHG)



Remaining bits of 1st operand either:

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

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

Bits 32-63 of R_1 unchanged

Condition Code is Unchanged!

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.

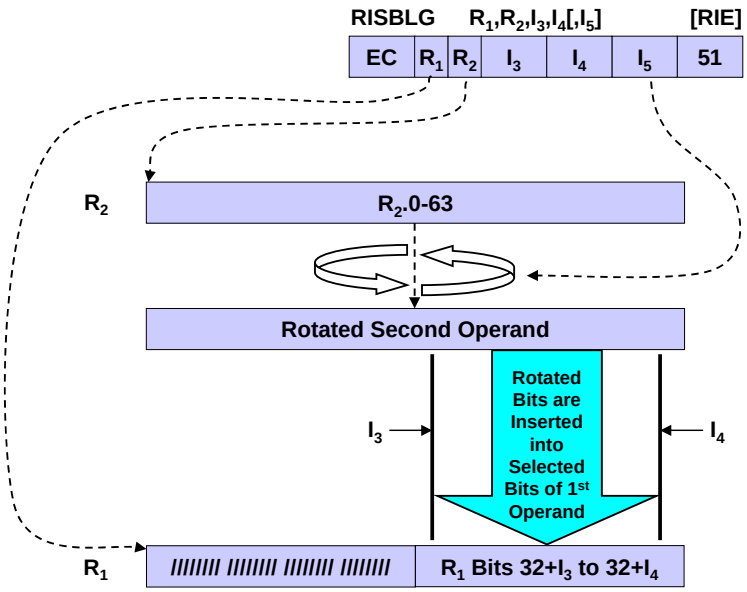
All 64 bits of the second operand are rotated to the left by the number of bits specified in the fifth operand (note, if the fifth operand is coded as a negative value, the rotation appears to occur to the right).

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)



Remaining bits of 1st operand either:

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

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

Bits 0-31 of R_1 unchanged

Condition Code is Unchanged!

61

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: 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).

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:

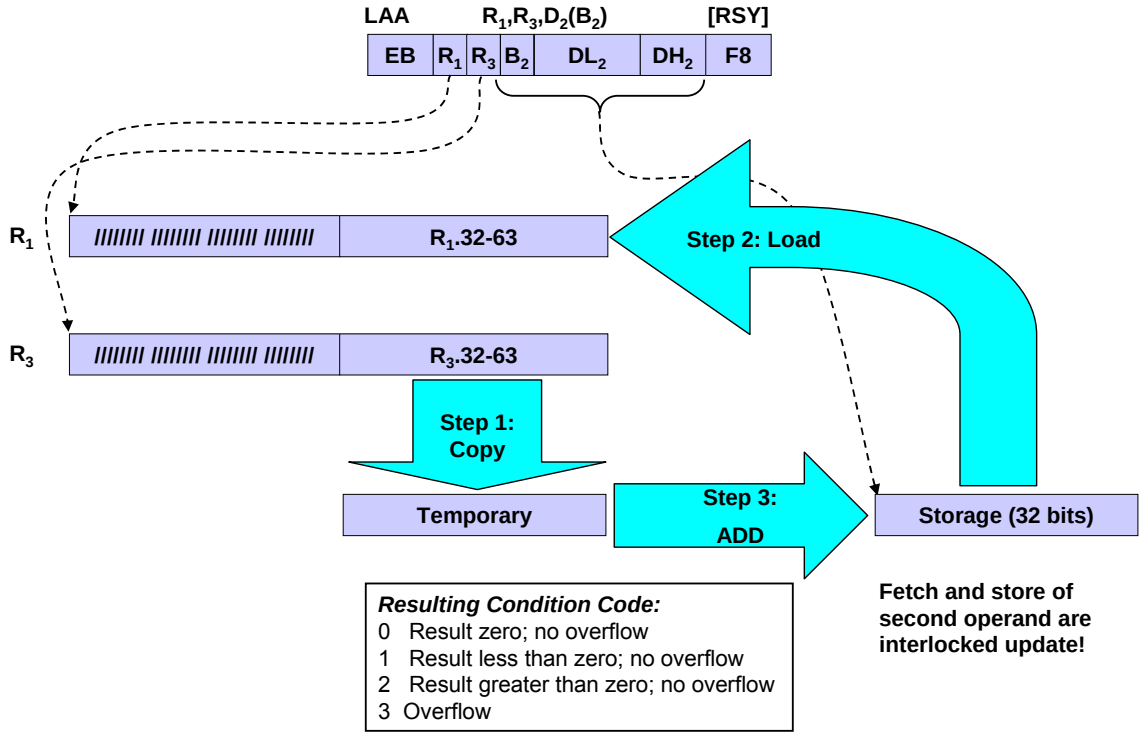
1. The second-operand storage location is fetched.
2. An operation is performed using the contents of the third-operand register and the storage location, with the result being placed into the storage location. The access of the storage location (beginning with the fetch in step 1, through the store 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.

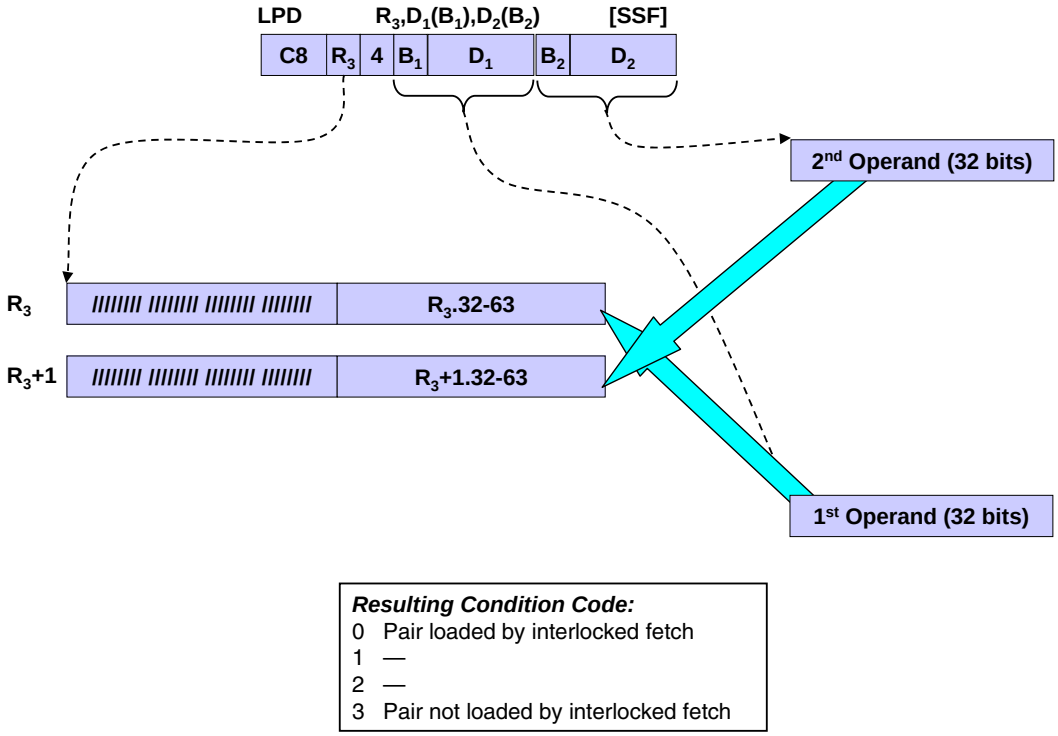
LOAD AND ADD (LAA)



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).

LOAD PAIR DISJOINT (LPD)



Resulting Condition Code:

0 Pair loaded by interlocked fetch
1 —
2 —
3 Pair not loaded by interlocked fetch

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.

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:

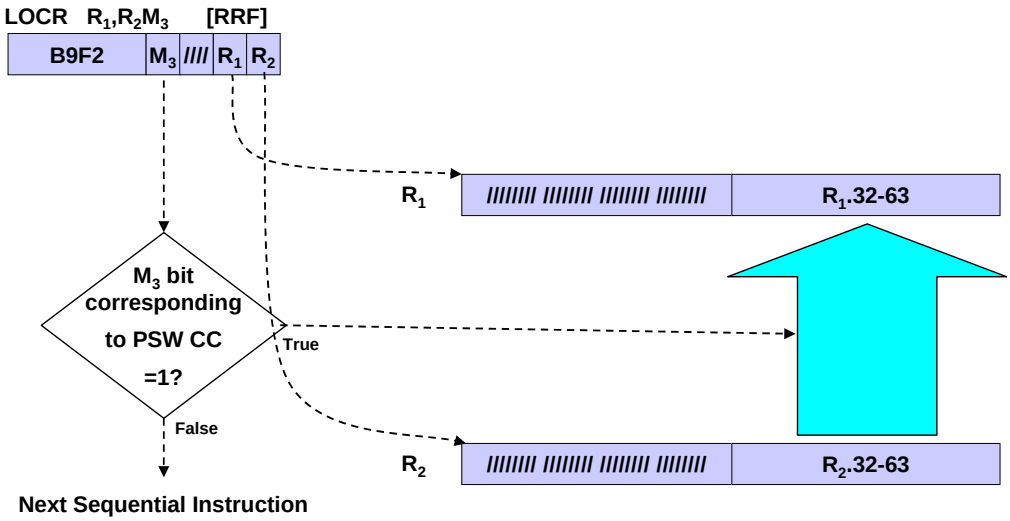
R_n	Register operand 'n'
S20	Storage operand designated by base register with 20-bit signed long displacement

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.

LOAD ON CONDITION (LOCR)



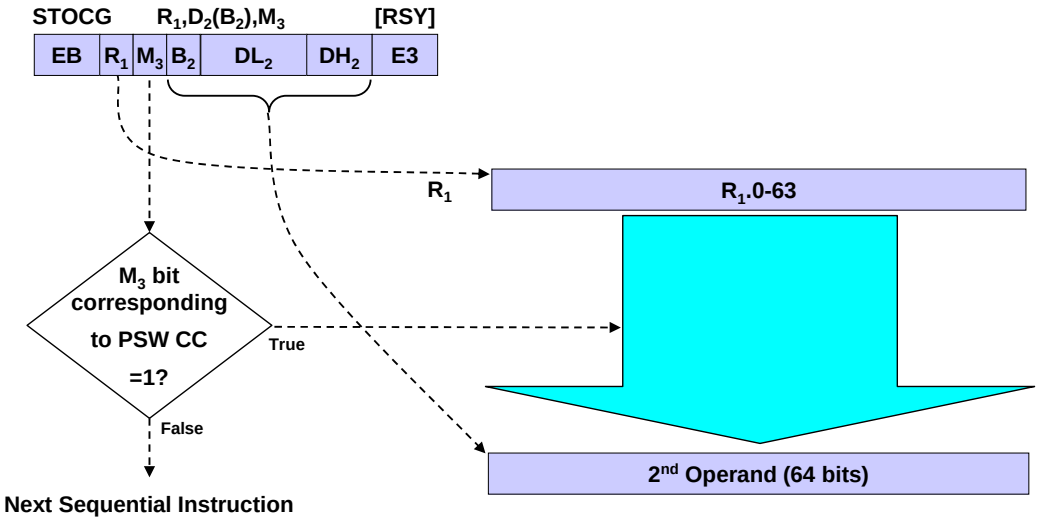
Condition Code is Unchanged

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.

STORE ON CONDITION (STOCG)



Condition Code is Unchanged

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

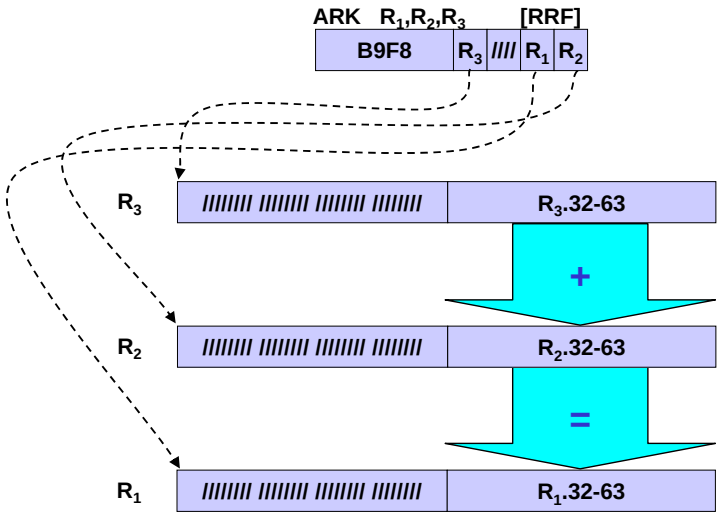
Explanation:

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).

ADD (ARK)



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

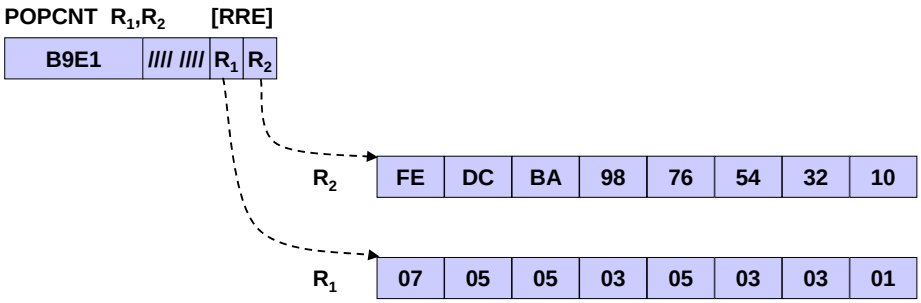
For ADD (ARK), the second operand is added to the third operand, and the result is placed in the first operand. Each operand occupies the rightmost 32 bits (bits 32-63) of the general register designated by the corresponding R field of the instruction.

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.

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**

76

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
- New privileged PERFORM CRYPTOGRAPHIC KEY MANAGEMENT OPERATION (PCKMO) instruction for importing clear keys
- Modifies existing KM, KMC, & KMAC ops to use encrypted keys

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.

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 Cipher-text-stealing (XTS) mode
- New Instructions:
  - CIPHER MESSAGE WITH CFB (KMF)
  - CIPHER MESSAGE WITH COUNTER (KMCTR)
  - CIPHER MESSAGE WITH OFB (KMO)
  - PERFORM CRYPTOGRAPHIC COMPUTATION (PCC)
- New function for existing KM, KIMD, & KMAC instructions

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.

Miscellaneous Enhancements (1)

- **Enhancements to general instructions:**
  - **Fast BCR Serialization Facility**
  - **Enhanced-Monitor Facility**
  - **CMPSC-Enhancement Facility**
- **Enhancement to control instructions:**
  - **IPTE-Range Facility**
  - **Nonquiescing Key-Setting Facility**
  - **Reset-Reference-Bits-Multiple Facility**

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 1110 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.

Summary

- S/360 architecture has evolved to provide numerous enhancements:
  - Additional addressing (24 → 31 → 64-bit addresses)
  - Larger binary data (32 → 64-bit registers)
  - Two additional floating-point representations (BFP & DFP)
  - Advanced program-linkage operations
  - Ability to access multiple address spaces simultaneously
  - Much broader I/O capacity
  - Numerous diagnostic features
  - Additional instructions for compiler efficiency
    - Register constraint relief
    - Cache optimization
    - General performance benefit
- z/Architecture now includes 924 instructions !!
- All while retaining application-program compatibility with the original S/360 instruction set

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).

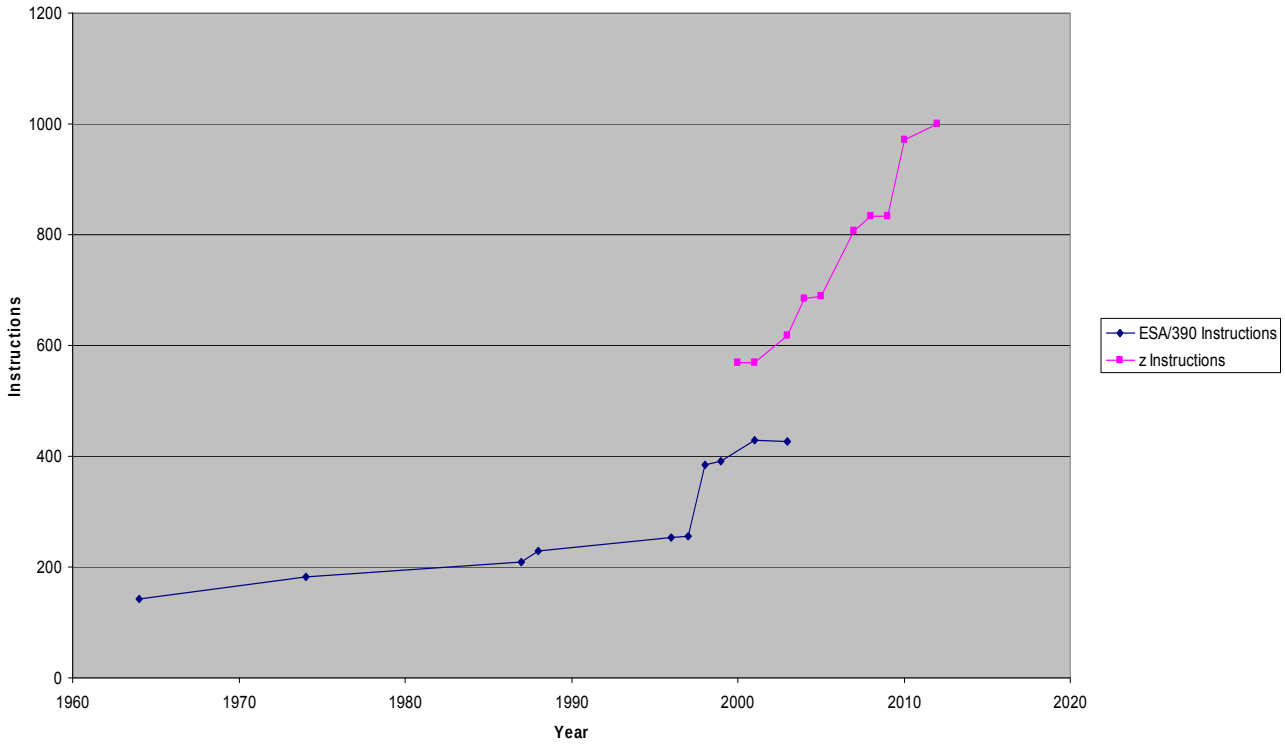
The IBM zEnterprise introduces a wide variety of new CPU facilities – some of which are simply designed to provide new or extended functions – however most of these facilities are designed to provide improved performance.

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.

z/Architecture Instruction Growth



85

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).

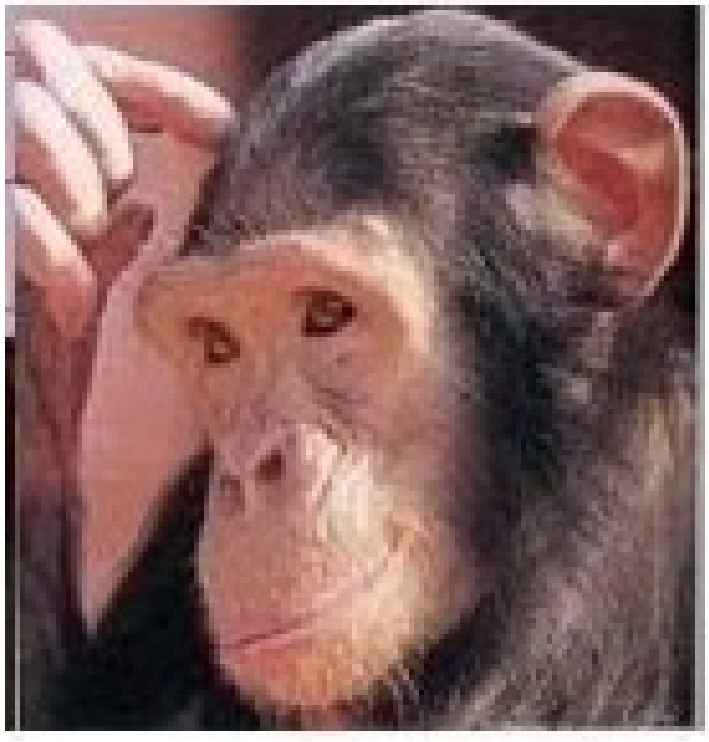
The IBM zEnterprise introduces a wide variety of new CPU facilities – some of which are simply designed to provide new or extended functions – however most of these facilities are designed to provide improved performance.

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?



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.