The Long Life of COBOL and the Mainframe.

John Rankin

June 20th – 22nd, 2024
VM Workshop 2024
Richmond, Virginia

CSI INTERNATIONAL

Overview

- Early History of the Industry
- Development of COBOL
- Adoption of COBOL as a strategy by IBM
- Hardware enhancements focused on COBOL

IBM and the Seven Dwarfs

- IBM – IBM 704
  - ✓ **B**urroughs – Datatron 205
  - ✓ **U**NIVAC (Sperry Rand) – UNIVAC I and II
  - ✓ **N**CR – NCR 304
  - ✓ **C**ontrol Data Corporation – CDC 1604
  - ✓ **H**oneywell – DATAmatic 1000
  - ✓ General Electric – GE 200, GE 400, GE 600 (sold to Honeywell)
  - ✓ RCA – Spectra 70 (sold to UNIVAC)

The state of programming in 1954

- Almost all programming was done in Assembler or Machine Code
- Hand coding was part of the art of programming
- Compilers were about to become a thing
- IBM’s John Backus begins to develop FORTRAN for the 704
  - ✓ Backus sees programing as algebraic
- UNIVAC’s Grace Hopper begins to develop FLOW-MATIC
  - ✓ Hopper sees programming as English like
- Burroughs’ Mary Hawes is working as a senior product planner
  - ✓ Hawes helped develop B-0 which became FLOW-MATIC

FORTRAN I

- John Backus requests permission from IBM (Cuthbert Hurd) in late 1953
- Development team is assembled in early 1954
- The first manual is produced October 15, 1956
- The objectives:
  - ✓ Produce object code that functions as efficiently as hand coding.
  - ✓ Compile a program without using too many resources
  - ✓ Improve the speed of the programming process
  - ✓ Make the IBM 704 more desirable
  - ✓ Very little concern was put into the language design
- The first release April 8, 1957

CSI INTERNATIONAL

FORTRAN I, a tower of Babel

- John Backus only planned his compiler for the IBM 704
- Every computer manufacturer produced versions of FORTRAN
- FORTRAN I was incomplete in several major areas:
    - ✓ Each manufacturer created incompatible extensions
    - ✓ A program written in FORTRAN needed to be rewritten on new machines
- The reasons for diversity:
    - ✓ Too little work was put into the language design
    - ✓ Had been built only for IBM, so other companies needed changes
    - ✓ The documentation allowed early competition
- Dozens of incompatible FORTRAN compilers were produced

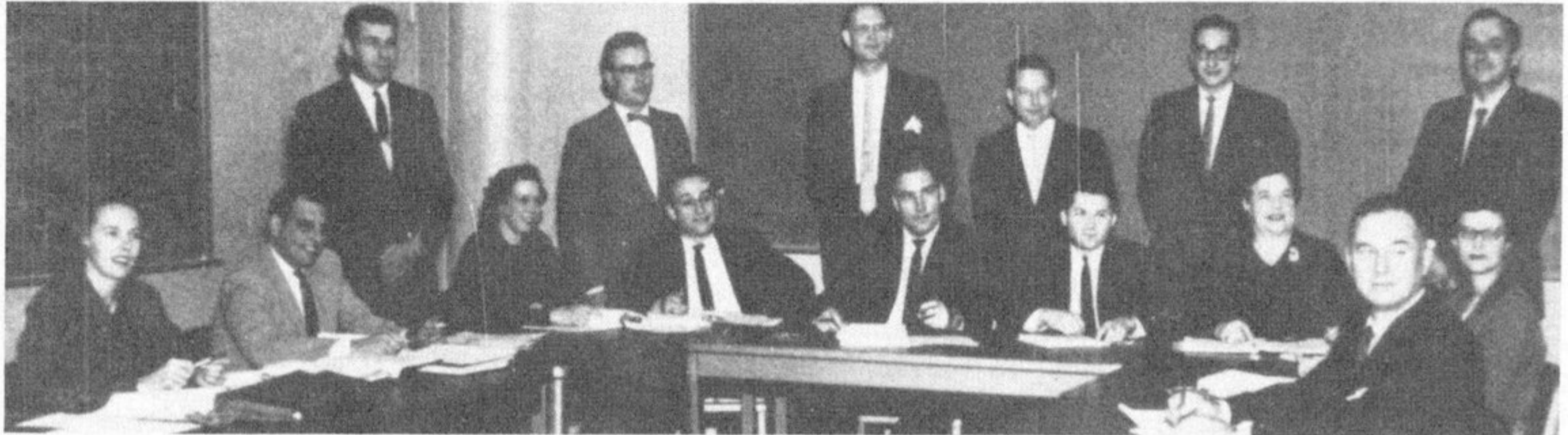
A meeting at the Pentagon

- In May of 1959, a meeting was held at the Pentagon
- A number of computer manufacturers and industry leaders
- Setup the Conference on Data Systems Languages
- Purpose:
  - ✓ Create a Common Business Oriented Language (COBOL)
  - ✓ Truly portable from different machines
  - ✓ To resolve the government problem of massive diversity in computing
- Thoughts of the members:
  - ✓ Hopper, Only direction from the government could allow all these companies to work together without creating a restraint of trade issue
  - ✓ Hawes, Real business management needed to be clear and understood

First Report

- April 1960 the first CODASYL report on COBOL
- Short Range Committee
  - ✓ Air Materiel Command, U.S. Air Force
  - ✓ Bureau of Standards, Department of Commerce
  - ✓ Computer Science Corporation
  - ✓ Datamatic Division, Minneapolis – Honeywell Corporation
  - ✓ David Taylor Model Basin, Bureau of Ships, U.S. Navy
  - ✓ ElectroData Division, Burroughs Corporation
  - ✓ International Business Machines Corporation
  - ✓ Radio Corporation of America
  - ✓ Remington-Rand Division of Sperry-Rand, Inc.
  - ✓ Sylvania Electric Products, Inc.

Members of the Short-Range Committee in 1959



Members of the Short-Range Committee in 1959 (some members are missing from the picture). *From the left, seated*: Gertrude Tierney (IBM), William Logan (Burroughs), Frances (Betty) Holberton (David Taylor Model Basin), Daniel Goldstein (Univac), Joseph Wegstein (National Bureau of Standards), Howard Bromberg (RCA), Mary Hawes (Burroughs), Benjamin Cheydleur (RCA), Jean Sammet (Sylvania). *Standing*: Alfred Asch (U.S. Air Force), unknown, William Selden (IBM), Charles Gaudette (Minneapolis-Honeywell), Norman Discount (RCA), Vernon Reeves (Sylvania). Photograph from *Management and Business Automation*, March 1960, p. 24.

Competitions for COBOL firsts

- The First Compiler
  - ✓ RCA on an RCA 501, August 17, 1960
- The First successful cross platform compilation
  - ✓ RCA and UNIVAC, December 6-7, 1960
- The proof is in the success

COBOL immerges as a success

- Portability between machines
- Clear, understandable language
- Easy to teach
- Designed for Business
- Quickly adopted by all manufacturers except IBM
- SHARE and GUIDE take up the challenge to convince IBM
- Eleven Software Companies announce COBOL compilers
- IBM offers compilers for 7070, 7080, 709/90, and the 705, but refuses to specify any dates for delivery

The IBM Response to COBOL

- It's clear the FORTRAN isn't well suited for business
- IBM plans on a product called the Commercial Translator
- It comes up in the early 60s
- GUIDE indicates that they want Commercial Translator to include support for COBOL
- IBM indicates that they are not compatible and Commercial Translator will be released alone
- Still no indication of a date for COBOL

The IBM Long-term Response

- IBM sees Commercial Translator, as an alternative to COBOL
- IBM designs PL/I as their version of a business language
  - ✓ PL/I takes the innovative data area from COBOL and the formula from FORTRAN, to create a mammoth language
  - ✓ It takes too long to develop, not released until 1966
  - ✓ The documentation is so large that it is called “The Vienna Phone Directory”
  - ✓ No standard committee adopts it
  - ✓ No other manufacturer produces a version
- PL/I is still a language used today, but it was never the success IBM intended

CSI INTERNATIONAL

IBM gets the message

- IBM withdraws Commercial Translator from market
  - ✓ It produces poor code
  - ✓ An extremely weak alternative to COBOL
- IBM provides a COBOL Compilers for the 705 and 7090
- They are secretly planning the IBM/360 for 1964-1965
- SHARE and GUIDE
  - ✓ Indicate that if IBM wants users to buy their new IBM/360, it needs:
    - Robust support for COBOL programs
    - Embrace COBOL or the users will move to other vendors
    - Execute the object code produced by COBOL efficiently

The introduction of the 360

- Announced April 7th, 1964
- IBM receives strong opposition from the dwarfs:
  - ✓ Burroughs 5500
  - ✓ CDC 3600, 3300, and 3200
  - ✓ Honeywell 625, 635, and 1800
  - ✓ RCA 3301
  - ✓ Univac 1108
- IBM's 360 has a strong structural support for COBOL, but is slower

The competition of machines in 1965 and 1966

- IBM makes several introductions to build competition for the dwarfs
- s/360 Model 44, August 1965
  - ✓ Specifically for scientific operations. Reduced Instructions
  - ✓ Extremely fast for the 60s
- s/360 Model 67, August 1965
  - ✓ Designed for COBOL
  - ✓ Decimal instructions are highly efficient, and much faster
- Only IBM has built a machine designed to run COBOL
- s/360 Model 91, January 1966
  - ✓ High speed for scientific
  - ✓ Sadly decimal instructions are missing and depends upon program chk

IBM Sets the long-term strategy

- Announces s/360 Model 85
- Much faster than any other machine on the market
- Designed specifically for COBOL operations
- Leaves the dwarfs in the dust
- Fully embraces the 1968 COBOL Standard produced by CODASYL
- IBM realizes that the users want COBOL, and if their machines fully embrace COBOL, they will win

IBM in 1970s through 1990s

- COBOL becomes a stronger and stronger part of the world
- $2 trillion dollars is the total investment in COBOL systems
- 5 billion lines of new COBOL are developed every year
- 90% of global financial transactions are processed in COBOL.
- 75% of all daily business transactions are processed in COBOL
- 70% of mission-critical applications are in COBOL
- The language supports over 30 billion transactions per day.
- There are 1.5-2 million developers, globally, working with COBOL code.
- 15% of all new application functionality will be written in COBOL
- 800 billion lines of COBOL software are in use today

CSI INTERNATIONAL

The COBOL Language

- COBOL is extremely scalable.
- COBOL can support large volumes of transactional data.
- COBOL is extremely reliable for mission-critical applications, particularly when run on mainframes.
- COBOL is an easy-to-learn language that adapts well to business needs.
- COBOL is extremely portable across platforms.

COBOL Standards

- CODASYL Report on COBOL, April 1960
- CODASYL Extended COBOL Specifications, November 1962
- CODASYL COBOL, July 1969, Adopted by ANSI and ISO
- ANSI X3.23-1974, American National Standard COBOL, May 1974
- ANSI X3.23-1985, American National Standard COBOL, Sept. 1985
- ISO/IEC 1989, International Standard COBOL, June 2014
- ISO/IEC 1989, International Standard COBOL, January 2023
- Over 65 years of stable design

IBM Mainframe meets Physics

- As IBM enters the twenty first century
- The machines can't get much smaller to include more circuits
- IBM meets the challenge with creative and innovative design
  - ✓ Parallel activity
  - ✓ Extreme Cashing
  - ✓ Path prediction
  - ✓ Moving more and more of the operation onto the chip
- Highly memory oriented programs suffer
- IBM once again sees COBOL as an important Partner

Twenty Years of IBM Mainframe Development

- z900/z800 – December 2000 (Architecture 5)
- z990/z890 – June 2003 (Architecture 6)
- z9ec/z9bc – September 2005 (Architecture 7)
- z10ec/z10bc – February 2008 (Architecture 8)
- z196/z114 – August 2010 (Architecture 9)
- zEC12/zBC12 – September 2012 (Architecture 10)
- z13/z13s – March 2015 (Architecture 11)
- z14/z14R1 – August 2017 (Architecture 12)
- z15-T01/z15-T02 – September 2019 (Architecture 13)
- z16-A01/z16-A02 – May 2022 (Architecture 14)

z/900, z/990
(Arch 5 and 6)

- 64bit Architecture
  - ✓ 16 64bit registers
  - ✓ Full support for 24/31/64 addressability
  - ✓ Below the line
  - ✓ Above the bar (64bit Virtual Memory Objects)
- Relative Displacement Instructions
  - ✓ Branching without base registers
  - ✓ Distance is 2 billion half words forward or backward from location
- Long Displacement Instructions
  - ✓ Normal displacement is 4096 bytes off of a base register
  - ✓ Long displacement is 1 megabyte off of a base register

z/9, z/10
(Arch 7 and 8)

- Relative Displacement Instructions
  - ✓ Allows for completely relocatable code
  - ✓ Necessary for successful Linux implementation
    - Linux unable to utilize virtual storage
    - Only one large address space for the entire Linux
  - ✓ Baseless operation
- Immediate Literal Instructions
  - ✓ One, two, and four byte literals are part of the instruction
  - ✓ Moves operations onto the chip and avoids accessing memory
  - ✓ Loading 256 bytes of instructions into in Line Buffer

z196, zEC12, z13
(Arch 9, 10, and 11)

- Decimal Floating Point Instructions
  - ✓ 16 128 bit Floating Point Registers
  - ✓ 31 BCD Digits plus a sign
- Full set of Decimal Instructions
- IBM's first attempt to provide instructions for Cobol
- Reduces dependency on memory to memory operation
- Implementation of VECTOR facility
  - ✓ 32 128 bit Instructions
  - ✓ Integer
  - ✓ Floating
  - ✓ String

z14, z15, z16
(Arch 12, 13, and 14)

- Vector Facility Upgrades
- 32 128bit Registers and four areas of instruction:
  - ✓ Integer
  - ✓ Floating Point
  - ✓ String
  - ✓ Binary Coded Decimal – Implemented in the z14
- Extremely high speed operation
  - ✓ Up to 1, 2, 4, 8, or 16 elements operating simultaneously
  - ✓ Operation completely in processor unit
- The Vector Binary Coded Decimal Facility
  - ✓ Completely designed for the use with Cobol
  - ✓ 31 Digits of BCD per vector register

z16 adds New Value

- 7nm Telum Processor Unit
- 256KB L1 per PU Core
- 32MB L2 Semi-private per PU Core
- Completely new design in improved performance
- Speed improvement is dependent:
  - ✓ Operation close to the Core
  - ✓ Registers, and inline operation processing within Core
- Performance Issues since z13:
  - ✓ SIIS Storage into instruction stream
  - ✓ Heavy Memory Operations, such as BCD operations in Cobol
- Solution is the Vector Facility

The State of Cobol VSE/ESA

- Hardware Support
  - ✓ ESA instructions, limited in their ability to use current facilities
  - ✓ Heavily designed towards memory to memory operation
- Software Support
  - ✓ Old design for code base
    - Built upon 1968 code structure
    - Limited ability to react to hardware changes
    - Designed for a stable architecture
    - Discarded and rewritten for Enterprise Cobol
  - ✓ Use of runtime and assembler objects
    - Internal routines are used from the 1968 version of Cobol
    - LE used for runtime, limits performance in exchange for support

z/Cobol™ for VSE

- z/Cobol™ for VSE is designed for all modern IBM Mainframes
  - ✓ 100% Compatible for IBM Cobol VSE/ESA 1.1, 21csw 1.2
  - ✓ Meets the National Institute of Standards and Technology
    - 100% Certified against ANSI 1985 Cobol Standard
    - 340,000 Lines of Cobol, Testing Code Suite with 500+ programs tests
  - ✓ Maximum Performance on all IBM z Series mainframe platforms
- Completely New
- Generates High Performance Code
- Built for z/VSE and VSE^n

CSI INTERNATIONAL

Compiler Operation

- Cobol Source code can be structured to match:
  - ✓ ANSI 1968, ANSI 1974, ANSI 1985, or ISO/IEC 1989:2014(E)
  - ✓ Enhancements specifically designed for VSE
    - TCP/IP for ANSI Communication Description Entries
    - Report Description Entries, Compliant with ANSI 1974 and ANSI 1985
    - Addition VSE based file methods, and all IBM extensions
- Generated Object code:
  - ✓ Linked with provided library of non LE routines.
  - ✓ No LE environment required.
  - ✓ Object code can be executed anywhere in VSE
- Supports fully functional export of assembler code as output

Code Generated

- 100% 64 bit.
  - ✓ Dynamically adjusts to the calling environment, and uses IBM z/OS save area structures
- Reentrant, no SIIS issues
- Optimized for non memory to memory operation
- Takes full advantage of Long Displacements and Relative Branching
  - ✓ Each storage section can be addressed 1 Megabyte at a time
  - ✓ Branching works with full word relative movements
- All supporting routines
  - ✓ Provided in linkable 64bit object decks
  - ✓ Designed to work with 24/31/64 objects where necessary
- Runtime error recovery, including vector displays

Intellectual Property z/Cobol™ for VSE

- U.S. Patents:
  - ✓ 10,901,739 Issued: January 26th, 2021
  - ✓ 11,429,390 Issued: August 20th, 2022
- COBOL Standards
  - ✓ CODASYL COBOL – Issued: July, 1968
  - ✓ X3.23-1974 ANSI COBOL, Issued: May 10th, 1974
  - ✓ X3.23-1985 ANSI COBOL, Issued: September 10th, 1985
  - ✓ ISO/IEC 1989/Amendment 1, Intrinsic function module

Embracing the Power of the Hardware

- Levels of Support
  - ✓ Basic z/Series support, Architecture level 5
    - 64 instructions, relative branching, and long displacements
  - ✓ Immediate Values, Architecture level 7
    - Reduces the use of literal pools, and moves literals into instructions
  - ✓ Decimal Floating Point, Architecture level 9
    - Moves binary coded decimal away from memory to memory operations
  - ✓ Vector Facility, Architecture 12
    - Uses all 32 Vector Registers
    - Completely eliminates binary coded decimal memory instructions
    - All math operations occur on the chip, storing only when necessary

Vector Code Generated

- As math operations occur when z/Cobol™ elements are loaded into Vectors
- All 32 Vectors are continually used
- Vectors are saved prior to call operations
  - ✓ Allows z/Cobol™ code to use vectors while in CICS
  - ✓ Interfaces with standard callable routines
- Perform Verb operations
  - ✓ Utilizes Vectors for comparisons, and increments
  - ✓ Operates with as much data loaded into Vectors
- High performance operation
- Maintaining maximum instructions and data on chip and not memory

Performance Improvements

- Machine Support
  - ✓ z Series - Support for 64bit register usage
  - ✓ z10 - Decimal Floating Point
  - ✓ EC12 and BC12 - Decimal Floating Point Zone Extension
  - ✓ z13 and z13s - 32 Vector Registers and Instructions
  - ✓ z14 and z14 ZR1 - Vector Packed Decimal Facility
  - ✓ z15 and z15 T02 - Enhanced Vector Facility
  - ✓ z16 - Advanced Vector Facility Packed Decimal
- Structural Support
  - ✓ Highly optimized code generation
  - ✓ Use of registers, and vector registers, for intermediate results
  - ✓ Persistent and compressed code generation

Designing for COBOL

- After 60 Years, IBM keeps COBOL in the design of every new machine
- This can be clearly illustrated with the new z series machines
- Let's go through an example of:
  - ✓ The DIVIDE verb
  - ✓ Completely Supported by Vector Decimal Instructions

DIVIDE

FUNCTION: To divide one number into another and store the result in the last named field or the specified one.

```
DIVIDE  {literal-1   }  INTO  {literal-2   }  [GIVING field-name-3]
        {field-name-1}        {field-name-2}

        [UNROUNDED] [; ON SIZE ERROR any imperative statement]
```

Notes:

1. All notes specified under the ADD verb apply to the DIVIDE verb.

2. In addition to the above, division by zero constitutes a special type of "size error". Regardless of whether or not the "size error" option has been specified, an attempted division by zero will leave the result field unaltered.

CSI INTERNATIONAL

FUNCTION: To divide one numerical data-item into others and set the values of items equal to the results.

Option 1:

DIVIDE { data-name-1 | literal-1 } INTO data-name-2 [ROUNDED] [,data-name-3 [ROUNDED] ...]

[; ON SIZE ERROR any imperative statement]

Option 2:

DIVIDE { data-name-1 | literal-1 } INTO { data-name-2 | literal-2 } GIVING data-name-3 [ROUNDED]

[, data-name-4 [ROUNDED] ...] [; ON SIZE ERROR any imperative statement]

Notes:

1. The data-names used must refer to numeric elementary items whose descriptions appear in the DATA DIVISION of the program.

2. All rules specified under the ADD verb regarding the size of operands, presence of editing symbols in operands, the ON SIZE ERROR option, the ROUNDED option, the GIVING option, truncation, and the editing of results, apply to the DIVIDE verb.

   The implementor will insure that all size errors on intermediate results will be detected. The number of significant digits of the calculated result is determined by the composite of receiving data items (that is, the data-item resulting from the superimposition of all receiving data items lined up on their decimal point) which must not contain more digits than the maximum allowable operand (18 decimal digits).

3. When Option 1 is used, the initial value of data-name-1 or literal-1 will be divided into the initial values of each data-name-2, data-name-3, ... The values of each of these dividends will be replaced by the value of the quotient resulting from operation on that data-name. Note that literals must not be used as dividends.

4. Division by zero constitutes a special type of "size error". Program control may be provided through the use of a test for zero prior to attempting division. If the "zero test" type of program control is not provided, the rules specified under the ADD verb with respect to the ON SIZE ERROR option apply.

Digitized by Google

CSI INTERNATIONAL

7.16 THE DIVIDE STATEMENT

7.16.1 FUNCTION

The DIVIDE statement divides one numeric data item into others and sets the values of data items equal to the quotient and remainder.

7.16.2 GENERAL FORMAT

Format 1

```
DIVIDE { identifier-1 } INTO identifier-2 [ROUNDED] [, identifier-3 [ROUNDED]] ...
       { literal-1    }

     [; ON SIZE ERROR imperative-statement]
```

Format 2

```
DIVIDE { identifier-1 } INTO { identifier-2 } GIVING identifier-3 [ROUNDED]
       { literal-1    }      { literal-2    }

     [, identifier-4 [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 3

```
DIVIDE { identifier-1 } BY { identifier-2 } GIVING identifier-3 [ROUNDED]
       { literal-1    }    { literal-2    }

     [, identifier-4 [ROUNDED]] ... [; ON SIZE ERROR imperative-statement]
```

Format 4

```
DIVIDE { identifier-1 } INTO { identifier-2 } GIVING identifier-3 [ROUNDED]
       { literal-1    }      { literal-2    }

       REMAINDER identifier-4    [; ON SIZE ERROR imperative-statement]
```

Format 5

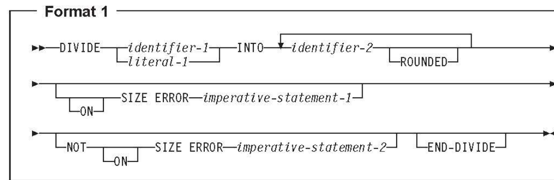
```
DIVIDE { identifier-1 } BY { identifier-2 } GIVING identifier-3 [ROUNDED]
       { literal-1    }    { literal-2    }

       REMAINDER identifier-4    [; ON SIZE ERROR imperative-statement]
```

CSI INTERNATIONAL

DIVIDE Statement

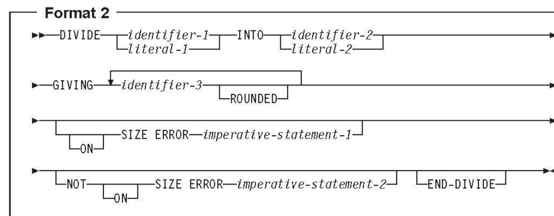
The DIVIDE statement divides one numeric data item into or by other(s) and sets the values of data items equal to the quotient and remainder.

Format 1



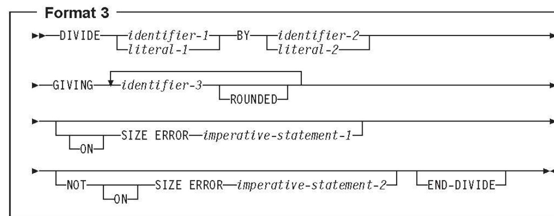
In Format 1, the value of identifier-1 or literal-1 is divided into the value of identifier-2, and the quotient is then stored in identifier-2. For each successive occurrence of identifier-2, the division takes place in the left-to-right order in which identifier-2 is specified.

Format 2



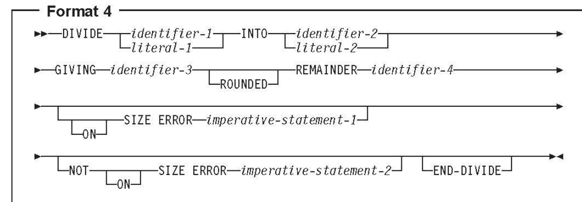
In Format 2, the value of identifier-1 or literal-1 is divided into the value of identifier-2 or literal-2. The value of the quotient is stored in each data item referenced by identifier-3.

Format 3



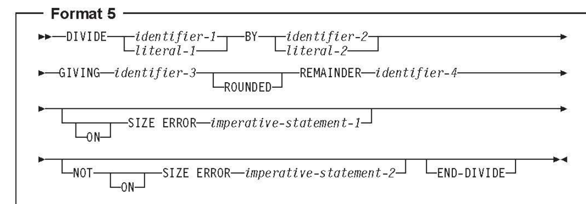
In Format 3, the value of identifier-1 or literal-1 is divided by the value of identifier-2 or literal-2. The value of the quotient is stored in each data item referenced by identifier-3.

Format 4



In Format 4, the value of identifier-1 or literal-1 is divided into identifier-2 or literal-2. The value of the quotient is stored in identifier-3, and the value of the remainder is stored in identifier-4.

Format 5



In Format 5, the value of identifier-1 or literal-1 is divided by identifier-2 or literal-2. The value of the quotient is stored in identifier-3, and the value of the remainder is stored in identifier-4.

For all Formats:

identifier-1, identifier-2
Must name an elementary numeric item. Identifier-1 and identifier-2 cannot be date fields.

identifier-3, identifier-4
Must name an elementary numeric or numeric-edited item.

If identifier-3 or identifier-4 is a date field, then see "Storing Arithmetic Results That Involve Date Fields" on page 178 for details on how the quotient or remainder is stored in identifier-3.

literal-1, literal-2
Must be a numeric literal.

In Formats 1, 2, and 3, floating-point data items and literals can be used anywhere that a numeric data item or literal can be specified.

In Formats 4 and 5, floating-point data items or literals cannot be used.

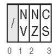
CSI INTERNATIONAL

The number of leftmost zero digits in the second operand, which is in the signed-packed-decimal format, are counted and placed in byte seven of the first operand; all other bytes of the first operand are set to zero. A second operand with all zero digits and sign, results in a count of 31 digits which is the number of decimal digits in signed-packed-decimal format.

When the no-validation (NV) control is zero or the condition-code-set (CS) control is one, all digit and sign codes of the second operand are checked for validity. When the validity check fails, if the NV control is zero a general-operand data exception is recognized; otherwise, the condition code is set to three.

The interpretation of the sign of the second operand is a function of the negative zero (NZ) control and this affects the condition code.

The M_3 field has the following format:



The bits of the M_3 field are:

- ***Reserved:*** Bit 0 is ignored but should contain zero; otherwise, the program may not operate compatibly in the future.
- ***No Validation (NV):*** When bit 1 is zero, the second operand digits and sign are checked for validity. If the validity check fails, a general-operand data exception is recognized. When bit 1 is one, no general-operand data exception is reported, though validity checking may still be performed for condition code setting.
- ***Negative Zero (NZ):*** Bit 2 only applies when bit 3 is one. When bit 2 is zero, both positive and negative zero values result in condition code zero being set. When bit 2 is one, a positive zero results in condition code zero being set and a negative zero results in condition code one being set.
- ***Condition Code Set (CS):*** When bit 3 is zero, the condition code is not set and remains unchanged. When bit 3 is one, the condition code is set as specified in the resulting condition code section below.

Resulting Condition Code:

When the CS bit is one, the condition code is set follows:

0 All digits and sign are valid and the source op and is equal to zero, except when NZ=1 and sign is negative.
1 All digits and sign are valid and the source op and is negative, or when NZ=1 and the sign negative and source operand is equal to zero.
2 All digits and sign are valid, the source opera is not equal to zero, and the sign is positive.
3 A digit or sign is invalid

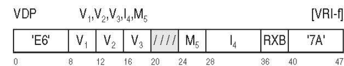
When the CS bit is zero, the condition code rema unchanged.

The operation is suppressed when a data except is recognized.

Program Exceptions:

- Data with DXC FE, vector instruction
- Data with DXC 00, general operand
- Operation (if the vector-packed-decim enhancement facility 2 is not installed)
- Transaction constraint

VECTOR DIVIDE DECIMAL



The second operand (the dividend) is divided by third operand (the divisor). The sign and specif number of rightmost digits of the quotient are plac in the first-operand location with other digits set zero. The operands and result are in the signe packed-decimal format.

The sign codes of the second and third operar may be modified for use in the operation by the fo operand two positive (P2) and force operand thı positive (P3) controls respectively.

All digit codes are checked for validity. The s codes are checked for validity unless overridden the force operand two positive (P2) or force opera three positive (P3) controls.

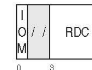
When the result, after the result digits count (RD control is applied, is nonzero and the force opera one positive (P1) control is zero, rules of algebra determine the sign of the result and a preferred sign code is used. When the result, after the RDC control is applied, is zero and the P1 control is zero, the sign of the result is made positive with preferred sign code 1100. When the P1 control is one, the sign of the result is made positive with sign code 1111.

If the divisor is zero and the divisor sign code used is valid, a decimal-divide exception is recognized. This includes the case of division of zero by zero. The divisor sign code used is the third operand sign code when the force operand 3 positive (P3) bit is zero, and is a positive sign code when the force operand 3 positive (P3) bit is one.

If the RDC control does not specify enough digits to contain all leftmost nonzero digits of the quotient, decimal overflow occurs. The operation is completed. The result is obtained by ignoring the overflow digits, and if the condition code set (CS) control is one, condition code 3 is set. If the decimal-overflow mask in the PSW is one and the instruction-overflow mask (IOM) is zero, a program interruption for decimal overflow occurs.

If the RDC control specifies less than thirty one digits, zeros are placed in the remaining leftmost digits of the first operand.

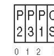
The I_4 field has the following format:



The bits of the I_4 field are defined as follows:

- ***Instruction-Overflow Mask (IOM):*** When the vector-packed-decimal-enhancement facility 1 is not installed, bit 0 is reserved and must contain zero; otherwise, a specification exception is recognized. When the vector-packed-decimal-enhancement facility 1 is installed, bit 0 is the instruction-overflow mask. When the instruction-overflow mask is one, recognition of a decimal overflow program interrupt is suppressed.
- ***Reserved:*** Bits 1-2 are reserved and must contain zeros. Otherwise, a specification exception is recognized.
- ***Result Digits Count (RDC):*** Bits 3-7 contain an unsigned binary number specifying the number of rightmost digits of the quotient to be placed in the first operand. If the magnitude of the quotient is larger than the largest decimal number that can be represented with the specified number of digits, decimal overflow occurs, and if the decimal-overflow mask is one, a program interruption for decimal overflow occurs. If the RDC field is zero, a specification exception is recognized.

The M_5 field has the following format:



The bits of the M_5 field are:

- ***Force Operand 2 Positive (P2):*** When bit 0 is one, the second operand sign is treated as a positive sign and is not checked for validity. When bit 0 is zero, the second operand sign is used in the operation and is checked for validity.
- ***Force Operand 3 Positive (P3):*** When bit 1 is one, the third operand sign is treated as a positive sign and is not checked for validity. When bit 1 is zero, the third operand sign is used in the operation and is checked for validity.
- ***Force Operand 1 Positive (P1):*** When bit 2 is one, the sign of the result placed in the first operand is forced to positive and a sign code of 1111 is used. When bit 2 is zero, the sign of the result placed in the first operand is the preferred sign code for the sign of the quotient.
- ***Condition Code Set (CS):*** When bit 3 is zero, the condition code is not set and remains unchanged. When bit 3 is one, the condition code is set as specified in the resulting condition code section below.

Resulting Condition Code:

When the CS bit is one, the condition code is set as follows:

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

VECTOR DIVIDE DECIMAL

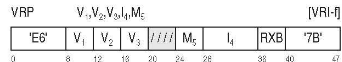
CSI INTERNATIONAL

- Data with DXC FE, vector instruction
- Data with DXC 00, general operand
- Decimal overflow
- Operation (if the vector-packed-decimal facility for z/Architecture is not installed)
- Specification
- Transaction constraint

Programming Note: A zero result after applying the RDC will always result in setting condition code 0 if the CS bit is one, independent of the positive or negative sign of the zero.

VECTOR REMAINDER DECIMAL

VRP V_1,V_2,V_3,I_4,M_5 [VRI-f]



The second operand (the dividend) is divided by the third operand (the divisor). The sign and specified number of rightmost digits of the remainder are placed in the first-operand location with other digits set to zero. The operands and result are in the signed-packed-decimal format.

The sign codes of the second and third operands may be modified for use in the operation by the force operand two positive (P2) and force operand three positive (P3) controls respectively.

All digit codes are checked for validity. The sign codes are checked for validity unless overridden by the force operand two positive (P2) or force operand three positive (P3) controls.

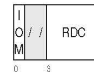
When the result, after the result digits count (RDC) control is applied, is nonzero and the force operand one positive (P1) control is zero, the sign of the dividend after P2 is applied is the sign of the result and a preferred sign code is used. When the result, after the RDC control is applied, is zero and the P1 control is zero, the sign of the result is made positive with preferred sign code 1100. When the P1 control is one, the sign of the result is made positive with sign code 1111.

If the divisor is zero and the divisor sign code used is valid, a decimal-divide exception is recognized. This includes the case of division of zero by zero. The divisor sign code used is the third operand sign code when the force operand 3 positive (P3) bit is zero, and is a positive sign code when the force operand 3 positive (P3) bit is one.

If the RDC control does not specify enough digits to contain all leftmost nonzero digits of the remainder, decimal overflow occurs. The operation is completed. The result is obtained by ignoring the overflow digits, and if the condition code set (CS) flag is one, condition code 3 is set. If the decimal-overflow mask in the PSW is one and the instruction-overflow mask (IOM) is zero, a program interruption for decimal overflow occurs.

If the RDC control specifies less than thirty one digits, zeros are placed in the remaining leftmost digits of the first operand.

The I_4 field has the following format:



The bits of the I_4 field are defined as follows:

- ***Instruction-Overflow Mask (IOM):*** When the vector-packed-decimal-enhancement facility 1 is not installed, bit 0 is reserved and must contain zero; otherwise, a specification exception is recognized. When the vector-packed-decimal-enhancement facility 1 is installed, bit 0 is the instruction-overflow mask. When the instruction-overflow mask is one, recognition of a decimal overflow program interrupt is suppressed.
- ***Reserved:*** Bits 1-2 are reserved and must contain zeros. Otherwise, a specification exception is recognized.
- ***Result Digits Count (RDC):*** Bits 3-7 contain an unsigned binary number specifying the number of rightmost digits of the remainder to be placed in the first operand. If the magnitude of the remainder is larger than the largest decimal number that can be represented with the specified number of digits, decimal overflow occurs, and if the decimal-overflow mask is one, a program interruption for decimal overflow occurs. If the RDC field is zero, a specification exception is recognized.

CSI INTERNATIONAL

IBM Hardware is Designed for COBOL

- The lesson IBM Learned in the 50s and 60s, they follow today
- The user community needs and wants COBOL
- COBOL is designed by the industry and not IBM
- The Hardware is successful when it supports IBM
- Every new release of hardware comes with a new release of Enterprise COBOL from IBM
- No programming language has been designed that is as Common, or as Business Oriented.

CSI INTERNATIONAL

Thank You