Index

Index

CHAPTER 0000 INTRODUCTION

- **Overview**

  This handbook describes about S/390 Assembly language instructions and helps programmer to understand & code the machine instructions. The system can understand and interpret only machine language. Machine language is in binary form and thus, very difficult to write.

  The assembler language is a symbolic programming language that you can use to code instructions instead of coding in machine language. The assembler must translate the symbolic language into machine language before the system can run the program.

- **Advantages**

  Interact with Operating System Services
  Load Module is compact; Faster Execution
  Programmer can understanding how the system works internally

- **Disadvantages**

  Hard to Understand, unlike High-Level Languages
  Hard to find Resources ??? !!!
  Machine Dependent which makes porting impossible

Notes

CHAPTER 0001 S/390 ARCHITECTURE

Evolution

S/360 24 bit addressing / no virtual memory concepts. Named after availability of the system around the clock(360 Degree)

S/370 31 bit addressing / virtual memory concept introduced.

S/390 Extended System Architecture(ESA) with more OS functionality like Base Sysplex .

Z/OS 64 bit addressing

Uni-Processor

- Single Processor & Memory space
- Availability - Contains single points of failure; disruptive change
- Systems Management - Easy to Manage Work

Tightly Coupled Multiprocessor

- Multiple Processors, but sharing single Memory space.
- Capacity - Increased over that of a uni-processor but limited by the maximum number of CP(s) in the Central Processing Complex(CPC)
- Systems Management - Easy to Manage Work

Loosely Coupled Multiprocessor

- Multiple CPC's each having separate Memory space.
- Each CPC's communicated via Channel-to-Channel Subsystem.
- Requires additional systems management, separate MVS images communicate to share data sets, printers, and consoles

TPF is LCMP system, capable of handling 6 such TCMPs with 8 processors in each CPC.

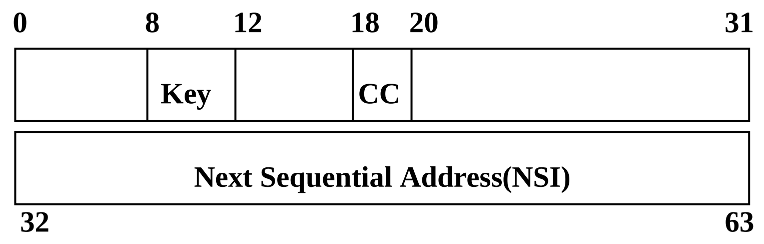
Notes

Registers

- General Purpose Registers(4 bytes) 0-15
- Floating Point Registers (8 bytes) 0,2,4,6
- Control Registers (4 bytes) 0-15
- Access Register (4 bytes) 0-15

Program Status Word (PSW)

- The program-status word (PSW) includes the instruction address (4 bytes), condition code(2 bits), and other information used to control instruction sequencing and to determine the state of the CPU. The active or controlling PSW is called the current PSW. It governs the program currently being executed.



- Storage Key(4 bits) are used to specify protection of page where the instruction addresses in the memory.

Notes

Symbol Table

- The Symbol Table is built & used by the assembler. After the program has been assembled, it is forgotten.
- The Assembler has two passes thru the program
  - Locates any labels and builds up the symbol table to say where these labels are in the program
  - Uses the symbol table to turn any references to labels which it finds in the coding into base and displacement format

Addressing

- Absolute Address: Physical Main Memory
- Real Address: View of Main Memory by each processor using its Prefix register.
- Virtual Address: View of Memory in Secondary Storage devices
  like DASD, TAPE or any external cache.
- Prefixing : Translation of Real Address into Absolute Address using Prefix register.
- Dynamic Address Translation(DAT) : Translation of Virtual Address into Absolute using Page table & Segment table
- Programs and data are relocatable i.e., they may be in different places in store each time they are used. Therefore base & displacement is used instead of absolute addressing

Notes

Assembler Directive

USING

Directive declares a base register, which is to be used by the assembler. The assembler will use the register to address the locations, which fall under the designated area (name).

```
USING          name, register     or
USING          name, registers
```

If more than one register is specified, the first 4K ok locations are addressed by the 1st register and the next 4K by the 2nd and son on.

DROP

Directs the assembler to stop using the register(s) specified as base register.

```
DROP      register     or
DROP      registers
```

TITLE

Causes a skip to the next page in the print.

```
TITLE     string
```

SPACE

Directive will result in a blank line being inserted in the assembly listing. Parameter [n] is optional which indicates the number of blank lines.(Def: 1)

```
SPACE          [n]
```

PRINT

Directive to control the amount of details in the assembly listing.

```
PRINT     [on/off] [,gen/nogen] [,nodata/data]
```

ON/OFF – print the assembler statements in the listing or not
GEN/NOGEN – print the statements generated by the macros or not.

EJECT

Directive causes a skip to the new page in the print out.

```
EJECT
```

END

Directive marks the last source statement in the module. Any further statements will be ignored.

```
END
```

CSECT
Signifies the start of Code Section in the program. Execution of instructions starts from here.

```
CSECT     name
```

DSECT
Signifies the start of Data Section in the program. Also know as Dummy section, since it doesn't occupy any memory in the program.

```
DSECT     name
```

LTORG
Positions the literal pool, which was created since the beginning of the program or since the last LTORG at the next double word boundary in the program.

```
LTORG
```

EQU
Informs the assembler to assign to the label name the value of the expression. The expression may be either relocatable or absolute. Assembler will substitute the value of the expression wherever the name is used.

```
NAME      EQU       expression
```

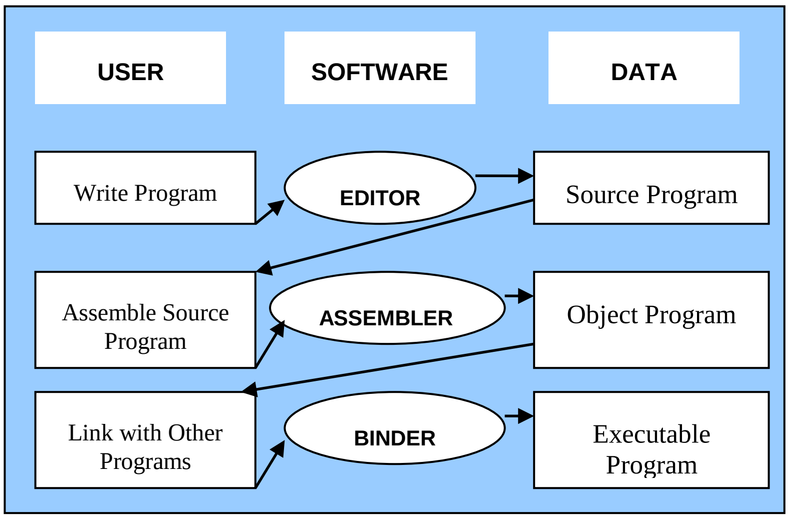
ORG
This statement changes the current value of the location counter. Re-defines the contents or layout of the storage.

```
ORG       relocatable expression
```

Omitting the operand in the ORG directive will result in the location counter being set to the next available value in the current CSECT.

Notes

Assembler Process



Assembler Coding Format

- Columns 1 to 72 is used for coding
- Column 1 for coding labels or ‘*’ to indicate comment
- Column 10 for instructions
- Column 16 for operands & continuation
- Column 36 for comments
- Column 72 for continuation marker

Notes

Assembler Statements

The assembler language is made up of statements that represent either instructions or comments. The instruction statements are the working part of the language and are divided into the following three groups:

Machine instructions	eg: LA, AR
Assembler instructions	eg: CSECT, ORG
Macro instructions	eg: OPEN, GET

Program Addressability

- To establish addressability of a control section you must:
  - Specify a base address from which the assembler can compute displacements to the labels within the CSECT
  - Assign the base registers to contain this base address
  - Write the instructions that loads the base registers with the base address
- The following example shows the base address at TEST, that is assigned to register 12

```
TEST      CSECT              The base address
          USING TEST,12      Assign base register
          LR 12,15           Load the base address
```

During execution, the EPA(Effective Program Address) is loaded into base register R15.

Notes

- **Types**

  Decimal- Numbers ranges from 0 to 9 with the base of 10

  Binary - Numbers ranges from 0 to 1 with the base of 2

  +123 = ‘01111011’ -123=‘10000101’

  Hexadecimal– Numbers ranges from 0 to F with the base of 16

  +123 = ‘7B’ -123 = ’85’

  Packed Decimal-Numbers which are defined for the ease of calculation.

  +123 = ‘123C’ -123 = ‘123D’

  Zoned Decimal-Numbers which are defined for the ease of display.

  +123 = ‘F1F2C3’ -123 = ‘F1F2D3’

- **Conversion**

  Binary to Hexadecimal B’10101001’ => B’**1010 1001**’ => X’**A9**’

  Binary to Decimal B’10101001’ =>

  => 1*2^7+0*2^6+1*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0

  => 128+0+32+0+8+0+0+1 = 169

  Decimal to Binary Divide the decimal number by 2 and group the reminder and quotient and form a binary number.

  Hexadecimal to Binary Convert each nibbles in the hexadecimal number into its equivalent binary.

  Decimal to Hexadecimal X’**A9**’ => 12*16^1+9*16^0 => **169**

  Hexadecimal to Decimal Divide the decimal number by 16 and group the reminder and quotient and form a Hexadecimal number.

Notes

Data Types

Format	Code	Explanation
Binary	**B**	8 binary digits in 1 byte. e.g. B'10101010'
Character	**C**	1 EBCDIC-character in 1 byte. e.g 'A', '9', '/'
Hexa	**X**	2 hex-digits in 1 byte. e.g. X'0DF1'
Half word	**H**	Signed decimal number in a 2-byte field (on boundary). Values can range from –32,768 to +32767. e.g. H'123' , H'-123'
Full word	**F**	Signed decimal number in a 4-byte field (on boundary). Values can vary from –2,147,483,648 to 2,147,483,647. e.g. F'123' , F'-123'
Double	**D**	Signed decimal number in an 8-byte field (on boundary). Total Values it can take is 2^{64}
Packed	**P**	2 decimal digits in 1 byte (signed) Max. number of digits = 31 (16 bytes). e.g. P'123', P'-123'
Zoned	**Z**	1 decimal digits in 1 byte (signed). Similar to Character, but used to store numbers '0' to '9'. e.g. Z'123' , Z'-123'.
Float	**E**	Takes 4 or 8 bytes in Storage based on Single Precision or Double Precision. e.g. E'+123.5' , or E'1E-20'
Address	**A**	Address-value in a 4-byte field (on boundary). e.g A(VARA)
Length	**Y**	Length-value in a 2-byte field (on boundary). e.g Y(VARB-VARA)

Notes

- **Data Definition**

Data Constants DC

The **DC** statement is used to define the constants in a program.

```
Label    DC     [D] T [L]'data'
NAME     DC       2CL3'EDS'
```

Don't store any values inside the Data constants field.

For examples see Appendix : C3

Data Storage   DS

The **DS** statement reserves storage without defining a value for that storage area.

```
Label    DS     [D] T [L]
NAME     DS     10XL30
```

For example see Appendix: C3

D – Duplication Factor
T - Data Type
L - Length Modifier

Notes

- **Operands**

  Explicit
  Operand is coded explicitly in the program.
  ```
  MOVE   GRADE, =C'ONE'
  ```

  Implicit
  Operand is coded implicitly in the program, which is defined in the Data Section.
  ```
  MOVE  GRADE,FIRST
  FIRST   DC   C'ONE'
  ```

- **Terms**

  Self-defining

  Constants that designate instruction components like registers, displacement and masks within the operand entry.
  ```
   MOVE  GRADE,X'A'
  ```

  Symbols

  Sequence of characters limited to a max of 8 and should begin with an alphabet.
  ```
   LEN256   EQU   256
  ```

Notes

- **Literals**

  Literals are explicit constants. Assembler will define the storage needed.
  Literal pool (LTORG) is the place where storage is allocated for all the literals within a CSECT.

  ```
  Example:     ADD   R2,=F'5'    <= Literal

               LTORG
               =F'5'
  ```

- **Expressions**

  Absolute
  Expression whose value is independent of its location in the memory.
  X – Y
  4 * X – 8

  Relocatable
  Expression whose value is dependent on where the program is loaded in the memory.
  X + 4
  X + Y – 4

  Expression who is in the form ***rel-rel*** or ***abs#abs*** where # can be +, - , * or / is always absolute.
  Expression of form ***rel-abs*** or ***rel+abs*** will be relocatable.

Notes

CHAPTER 0110 INSTRUCTION FORMATS

RR Format **-** *Register & Register Operation*

Op Code	**R1**	**R2**	
0	**8**	**12**	**15**

RX Format **–** *Register & Indexed Storage Operation*

Op Code	**R1**	**X2**	**B2**	**D2**	
0	**8**	**12**	**16**	**20**	**31**

RS Format **–** *Register, Mask or Register & Storage*

Op Code	**R1**	**R2**	**B3**	**D3**	
0	**8**	**12**	**16**	**20**	**31**

SI Format **–** *Storage & Immediate Value Operation*

Op Code	**I2**	**B2**	**D1**	
0	**8**	**16**	**20**	**31**

SS Format **–** *Storage & Storage Operation*

Op Code	**L1**	**B1**	**D1**	**B2**	**D2**	
0	**8**	**16**	**20**	**32**	**36**	**47**

Notes

CHAPTER 0111 LOAD & STORE INSTRUCTIONS

LOAD INSTRUCTIONS

Load Address

Mnemonic	Operands
LA	**R1, D2 (X2, B2)**

41	R1	X	B	D2
0	8	12	16	20 31

Instruction Format: RX

To place the address of a storage-location in a register, the LA (Load Address) instruction is performed. The effective address of the 2nd-operand field is loaded in the 1st-operand register. The address is placed right adjusted in the register and if necessary left padded with zeros. Note that the effective address of the 2nd-operand field is calculated adding up the values of the index-register, the base-register and the displacement. The instruction may also be used to increment the contents of a register. The contents of register 5 are incremented by 8 with the coding: LA R5,8(0,R5)

Condition codes

Remains unchanged.

Notes

Load Fullword

Mnemonic	Operands
L	**R1, D2 (X2, B2)**

58	**R1**	**X2**	**B2**	**D2**	
0	8	12	16	20	31

Instruction Format : RX

To move four bytes with binary data from a storage location into a register, the L (Load fullword) instruction is performed. The data of the source-field (2nd operand), a fullword (boundary!) in storage, is loaded into the target-register (1st operand). The source-field remains unchanged.

Condition codes

Condition code remains unchanged!

Notes

Load Halfword

Mnemonic	Operands
LH	**R1, D2 (X2, B2)**

48	**R1**	**X2**	**B2**	**D2**
0	8	12	16	20 31

Instruction Format : RX

To move two bytes with binary data from a storage location into a register, the LH (Load Halfword) instruction is performed. It will do this in two steps. First step: the data of the source-field (2nd operand), a halfword (boundary!) in storage, is extended to a 32-bit signed binary integer. The extension bits are the same as the high-order bit of the source-field. Second step: the 32 bits are loaded into the target-register (1st operand). The source-field remains unchanged.

Condition codes

Condition code remains unchanged!

Notes

Load Multiple

Mnemonic	Operands
LM	**R1, R3, D2 (B2)**

98	**R1**	**R3**	**B2**	**D2**
0	8	12	16	20 31

Instruction Format : RS

To move a sequence of four bytes of binary data from a storage location into adjacent registers, the LM (Load Multiple) instruction is performed. The data of the source-field (2nd operand), a sequence of fullwords (boundary!) in storage, is loaded into the adjacent target-registers, which are defined by the starting-register (1st operand) and the ending-register (3rd operand). The target-registers are loaded in the ascending order of their register numbers (with register R0 follows register R15). The number of target-registers defines the length of the source-field. The source-field remains unchanged.

Condition codes

Condition code remains unchanged!

Notes

Load and Test Register

Mnemonic	Operands
LTR	**R1, R2**

12	**R1**	**R2**
0	8	12 15

Instruction Format : RR

Data transfers between two registers may also performed by the LTR (Load and Test Register) instruction. The contents of the source-register (2nd operand) are loaded into the target-register (1st operand). Additionally the contents of the source-register, treated as a 32-bit signed binary integer, are tested for sign and magnitude. The result of the test is indicated in the condition code. The source-register remains unchanged. If the two operands are the same register as in LTR R3,R3 , no data is transferred, but the condition code is set!

Condition codes

0	1st operand register is zero
1	1st operand register is negative
2	1st operand register is positive
3	Not used

Notes

Load Positive Register

Mnemonic	Operands
LPR	**R1, R2**

10	**R1**	**R2**
0	8	12 15

Instruction Format : RR

To transfer the absolute value between two registers, the LPR (Load Positive Register) instruction is performed. The binary data of the source-register (2nd operand) are changed into positive and the result is loaded into the target-register (1st operand). If the binary data of the source-register is already positive or zero, the data is transferred unchanged; otherwise the two's complement is loaded into the target-register. Afterwards the target-register is tested for magnitude. The result of the test is indicated in the condition code. The source-register and the result in the target-register are treated as 32-bit signed binary integers. The source-register remains unchanged.

Condition codes

0	1st operand register is zero
1	Not used
2	1st operand register is positive
3	Not used

Notes

Load Negative Register

Mnemonic	Operands
LNR	**R1, R2**

11	R1	R2
0	8	12 15

Instruction Format : RR

To transfer the two's complement of the absolute value between two registers, the LNR (Load Negative Register) instruction is performed. The binary data of the source-register (2nd operand) are changed into negative and the result is loaded into the target-register (1st operand). If the binary data of the source-register is already negative or zero, the data is transferred unchanged; otherwise the two's complement is loaded into the target-register. Afterwards the target-register is tested for magnitude. The result of the test is indicated in the condition code. The source-register and the result in the target-register are treated as 32-bit signed binary integers. The source-register remains unchanged.

Condition codes

0	1st operand register is zero
1	1st operand register is negative
2	Not used
3	Not used

Notes

Insert Character

Mnemonic	Operands
IC	**R1, D2 (X2, B2)**

43	R1	X2	B2	D2
0	8	12	16	20 31

Instruction Format : RX

To move one byte with binary data from a storage location into a register, the IC (Insert Character) instruction is performed. The data of the source-field (2nd operand), a single byte in storage, is inserted into the low-order byte of the target-register (1st operand). The source-field and the three high-order bytes of the target-register remain unchanged.

Condition codes

Remains unchanged!

Notes

Insert Character under Mask

Mnemonic	Operands
ICM	**R1, M3, D2 (B2)**

BF	**R1**	**M3**	**B2**	**D2**
0	8	12	16	20 31

Instruction Format : RS

The ICM (Insert Characters under Mask) instruction is an additional instruction to place one or more bytes of data in storage at wanted positions in a register. The data of the source-field (2nd operand), consecutive bytes in storage are inserted into the target-register (1st operand) in the positions indicated by the 4-bit mask (3rd operand). The number of indicated bytes gives the length of the source-field. The source-field and the non-affected bytes of the target-register remain unchanged. No boundary is required.

Condition codes

0	All inserted bits are zero or mask is zero
1	Leftmost inserted bit is one
2	Leftmost inserted bit is zero, but not all bits are zero
3	Not used

Notes

STORE INSTRUCTIONS

Store Fullword

Mnemonic	Operands
ST	**R1, D2 (X2, B2)**

50	**R1**	**X2**	**B2**	**D2**
0	8	12	16	20 31

Instruction Format : RX

To move the four bytes of binary data from a register to a storage location, the ST (STore fullword) instruction is performed. The contents of the source-register (1st operand) are stored in the target-field (2nd operand). The target-field has to be a Fullword (boundary!) in storage. The source-register remains unchanged.

Condition codes

Remains unchanged!

Notes

Store Halfword

Mnemonic	Operands
STH	**R1, D2 (X2, B2)**

40	**R1**	**X2**	**B2**	**D2**	
0	8	12	16	20	31

Instruction Format : **RX**

To move the two low-order bytes of binary data from a register to a storage location, the STH (STore Halfword) instruction is performed. The contents of the two low-order bytes of the source-register (1st operand) are stored in the target-field (2nd operand). The two high-order bytes of the source-register are ignored. The target-field has to be a halfword (boundary) in storage. The source-register remains unchanged.

Condition codes

Remains unchanged!

Notes

Store Character

Mnemonic	Operands
STC	**R1, D2 (X2, B2)**

42	**R1**	**X2**	**B2**	**D2**	
0	8	12	16	20	31

Instruction Format : RX

To move the low-order byte of binary data from a register to a storage location, the STC (STore Character) instruction is performed. The contents of the low-order byte of the source-register (1st operand) are stored in the target-field (2nd operand). The three high-order bytes of the source-register are ignored. The source-register remains unchanged.

Condition codes

Condition code remains unchanged!

Notes

Store Character under Mask

Mnemonic	Operands
STCM	**R1, M3, D2 (B2)**

BE	**R1**	**M3**	**B2**	**D2**	
0	8	12	16	20	31

Instruction Format : **RS**

The STCM (STore Characters under Mask) instruction is an additional instruction to store one or more bytes of data from wanted positions in a register in a storage field. The data of the source-register (1st operand), bytes indicated by the 4-bit mask (3rd operand), are stored at the target-location (2nd operand). The number of indicated bytes gives the length of the target-field. The source-register remains unchanged.No boundary is required.

Condition codes

Remains unchanged!

Notes

Store Multiple

Mnemonic	Operands
STM	**R1, R3, D2 (B2)**

0	8	12	16	20 31
90	**R1**	**R3**	**B2**	**D2**

Instruction Format : **RS**

To move the binary data from adjacent registers to a storage location, the STM (STore Multiple) instruction is performed. The contents of the source-registers, which are defined by the starting-register (1st operand) and the ending-register (3rd operand), are stored in the target-field (2nd operand). The source-registers are stored in the ascending order of their register numbers (with register R0 follows register R15). The target-field has to be a sequence of fullwords (boundary!) in storage. The number of source-registers defines the length of the target-field. The source-registers remain unchanged.

Condition codes

Condition code remains unchanged!

Notes

Add Register

Mnemonic	Operands
AR	**R1, R2**

1A	**R1**	**R2**
0	8	12 15

Instruction Format : RR

To add the contents of two registers, the AR (Add Register) instruction is performed. The contents of the 2nd-operand register are added to the contents of the 1st-operand register and the resulting sum is placed in the 1st-operand register. The operands and the result are treated as 32-bit signed binary integers. The 2nd-operand register remains unchanged. The calculation follows algebraic rules. The result is tested and a condition code is set. If the result does not fit into the 1st-operand register an overflow occurs and additionally the condition code is set to 3.rmed. The contents of the 2nd-operand remain unchanged.

Condition codes

0	Result is zero
1	Result is negative
2	Result is positive
3	Result doesn’t fit into the 1st operand register

Notes

Add Fullword

Mnemonic	Operands
A	**R1, D2 (X2, B2)**

5A	**R1**	**X2**	**B2**	**D2**
0	8	12	16	20 … 31

Instruction Format : RX

A four byte field in storage containing a binary value has to be added to the binary value of a register. The A (Add fullword) instruction handles this problem. The data of the 2nd-operand field, a fullword (->> boundary!) in storage, is added to the contents of the 1st-operand register and the resulting sum is placed in the 1st-operand register. The operands and the result are treated as 32-bit signed binary integers. The 2nd-operand field remains unchanged. The calculation follows algebraic rules. The result is tested and a condition code is set. If the result does not fit into the 1st-operand register an overflow occurs and additionally the condition code is set to 3. The 2nd field, a fullword (boundary!) in storage, is left unchanged.

Condition codes

0	Result is zero
1	Result is negative
2	Result is positive
3	Result doesn’t fit into the 1st operand register

Notes

Add Halfword

Mnemonic	Operands
A	R1, D2 (X2, B2)

4A	R1	X2	B2	D2
0	8	12	16	20 31

Instruction Format : **RX**

A two byte field in storage containing a binary value has to be added to the binary value of a register. The problem can be solved by using the AH (Add Halfword) instruction. The data of the 2nd-operand field, a halfword (boundary!) in storage, is added to the contents of the 1st-operand register and the resulting sum is placed in the 1st-operand register. The 2nd-operand field is treated as a 16-bit signed binary integer. The 1st-operand register and the result are treated as 32-bit signed binary integers. The 2nd-operand field remains unchanged. The calculation follows algebraic rules. The result is tested and a condition code is set. If the result does not fit into the 1st-operand register an overflow occurs and additionally the condition code is set to 3.

Condition codes

0	Result is zero
1	Result is negative
2	Result is positive
3	Result doesn't fit into the 1st operand register

Notes

Subtract Register

Mnemonic	Operands
SR	**R1, R2**

1B	**R1**	**R2**
0	8	12 15

Instruction Format : RR

To subtract the contents of one register from another register, the SR (Subtract Register) instruction is performed. The contents of the 2nd-operand register are subtracted from the contents of the 1st-operand register and the result is placed in the 1st-operand register. The operands and the result are treated as 32-bit signed binary integers. The 2nd-operand register remains unchanged. The calculation follows algebraic rules. The result is tested and a condition code is set. If the result does not fit into the 1st-operand register an overflow occurs and additionally the condition code is set to 3.

Condition codes

0	Result is zero
1	Result is negative
2	Result is positive
3	Result doesn't fit into the 1st operand register

Notes

Subtract Fullword

Mnemonic	Operands
S	**R1, D2 (X2, B2)**

5B	R1	X2	B2	D2
0	8	12	16	20 … 31

Instruction Format : RX

A four byte field in storage containing a binary value has to be subtracted from the binary value of a register. The S (Subtract fullword) instruction handles this problem. The data of the 2nd-operand field, a fullword (boundary!) in storage, is subtracted from the contents of the 1st-operand register and the result is placed in the 1st-operand register. The operands and the result are treated as 32-bit signed binary integers. The 2nd-operand field remains unchanged. The calculation follows algebraic rules. The result is tested and a condition code is set. If the result does not fit into the 1st-operand register an overflow occurs and additionally the condition code is set to 3.

Condition codes

0	Result is zero
1	Result is negative
2	Result is positive
3	Result doesn’t fit into the 1st operand register

Notes

Subtract Halfword

Mnemonic	Operands
S	R1, D2 (X2, B2)

4B	R1	X2	B2	D2
0	8	12	16	20 31

Instruction Format : **RX**

A two byte field in storage containing a binary value has to be subtracted from the binary value of a register. The problem can be solved by using the SH (Subtract Halfword) instruction. The data of the 2nd-operand field, a halfword (boundary!) in storage, is subtracted from the contents of the 1st-operand register and the result is placed in the 1st-operand register. The 2nd-operand field is treated as a 16-bit signed binary integer. The 1st-operand register and the result are treated as 32-bit signed binary integers. The 2nd-operand field remains unchanged. The calculation follows algebraic rules. The result is tested and a condition code is set. If the result does not fit into the 1st-operand register an overflow occurs and additionally the condition code is set to 3.

Condition codes

0	Result is zero
1	Result is negative
2	Result is positive
3	Result doesn’t fit into the 1st operand register

Notes

Multiply Register

Mnemonic	Operands
MR	**R1, R2**

1C	**R1**	**R2**
0	8	12 15

Instruction Format : **RR**

To multiply the contents of one register by another register, the MR (Multiply Register) instruction is performed. The considerations about the size of the product and the 1st-operand register-pair are described in the M (Multiply fullword) instruction. The contents of the 1st-operand odd-register (the contents of the even-register are ignored) are multiplied by the contents of the 2nd-operand register and the resulting product is placed in the 1st-operand register-pair. The operands are treated as 32-bit signed binary integers and the result is treated as a 64-bit signed binary integer. The 2nd-operand register remains unchanged.

Condition codes

Remains unchanged!

Notes

Multiply Fullword

Mnemonic	Operands
M	**R1, D2 (X2, B2)**

5C	R1	X2	B2	D2
0	8	12	16	20 … 31

Instruction Format : RX

The binary value of a register has to be multiplied by a four byte field in storage containing also a binary value. The M (Multiply fullword) instruction handles this problem. Let's consider the multiply instruction first (specially the size of the product): when two 32-bit operands are multiplied, the product requires 64-bits. Certainly the product cannot fit into the 1st-operand register! So it is not placed in a single register, but in a register-pair! The assembler requires that the register-pair has to be an even-odd pair! The contents of the 1st-operand odd-register (the contents of the even-register are ignored) are multiplied by the data of the 2nd-operand field, a fullword (boundary!) in storage and the resulting product is placed in the 1st-operand register-pair. The operands are treated as 32-bit signed binary integers and the result is treated as a 64-bit signed binary integer. The 2nd-operand field remains unchanged.

Condition codes
Remains unchanged!

Notes

Multiply Halfword

Mnemonic	Operands
MH	**R1, D2 (X2, B2)**

4C	**R1**	**X2**	**B2**	**D2**
0	8	12	16	20 — 31

Instruction Format : RX

The binary value of a register has to be multiplied by a two byte field in storage containing also a binary value. The MH (Multiply Halfword) instruction handles this problem. The contents of the 1st-operand register are multiplied by the data of the 2nd-operand field, a halfword (boundary!) in storage and the resulting product is placed in the 1st-operand register. Now we are multiplying a 32-bit operand by a 16-bit operand. The product contains 48 bits, but only the low-order 32 bits of the product are placed in the 1st-operand register. If the product requires more than 31 bits and the sign bit, the high-order bits are lost and the result is invalid. No overflow occurs! The 2nd-operand field is treated as a 16-bit signed binary integer. The 1st-operand register and the result are treated as 32-bit signed binary integers. The 2nd-operand field remains unchanged. The calculation follows algebraic rules.

Condition codes
Remains unchanged!

Notes

Divide Register

Mnemonic	Operands
DR	**R1, R2**

1D	**R1**	**R2**
0	8	12 15

Instruction Format : RR

To divide the contents of a register-pair by the contents of another register, the DR (Divide Register) instruction is performed. The considerations about the format of the result and the 1st-operand register-pair are described in the D (Divide fullword) instruction. The contents of the 1st-operand register-pair (even-odd) are divided by the contents of the 2nd-operand register. The quotient is placed in the 1st-operand odd-register and the remainder in the even-register. The 2nd-operand register (divisor), the quotient and the remainder are treated as 32-bit signed binary integers. The 1st-operand register-pair (dividend) is treated as a 64-bit signed binary integer. The remainder takes always the sign of the dividend. The 2nd-operand register remains unchanged. The calculation follows algebraic rules.

Condition codes
Condition code remains unchanged!

Notes

Divide Fullword

Mnemonic	Operands
D	**R1, D2 (X2, B2)**

5D	R1	X2	B2	D2
0	8	12	16	20 — 31

Instruction Format : RX

The binary value of a register-pair has to be divided by a four byte field in storage containing also a binary value. The D (Divide fullword) instruction handles this problem. Let's consider the divide instruction first (specially the size of the 1st operand): the problem here is that a division results in a quotient and a remainder. The assembler considers both. The instruction uses an even-odd register-pair to contain the quotient and the remainder. Also the dividend uses this register-pair. The contents of the 1st-operand register-pair (even-odd) are divided by the data of the 2nd-operand field, a fullword (boundary!) in storage. The quotient is placed in the 1st-operand odd-register and the remainder in the even-register. The 2nd-operand field (divisor), the quotient and the remainder are treated as 32-bit signed binary integers. The 1st-operand register-pair (dividend) is treated as a 64-bit signed binary integer. The remainder takes always the sign of the dividend. The 2nd-operand field remains unchanged.

Condition codes
Condition code remains unchanged!

Notes

CHAPTER 1001 MOVE & COMPARE INSTRUCTIONS

MOVE INSTRUCTIONS

Move Character

Mnemonic	Operands
MVC	**D1 (L1 , B1), D2 (B2)**

D2	**L1**	**B1**	**D1**	**B2**	**D2**	
0	8	16	20	32	36	47

Instruction Format : SS

Data transfers between two storage locations are performed by the MVC (MoVe Character) instruction. This is one of the most frequently-used assembler instructions. It moves data from the source-field (2nd operand) into the target-field (1st operand). The maximum length which can be moved is 256 bytes of data. The length is defined by the 1st-operand field, a): explicitly as in MVC FIELD1(14),FIELD2, or b): implicitly using the length defined at field definition. If the specified length is less than the length of the source-field (2nd operand), the low-order bytes of data are lost. The length of the source-field is not of interest. Data movement proceeds from left to right, byte by byte. The source-field remains unchanged.

Condition codes

Condition code remains unchanged!

Notes

Move Immediate

Mnemonic	Operands
MVI	**D1 (B1), I2**

92	**I2**	**B1**	**D1**
0	8	16	20 31

Instruction Format : SI

To move a single byte of data, defined at program coding, into a storage location the MVI (MoVe Immediate) instruction is performed. The source-data (2nd operand) is moved into the target-field (1st operand). The single byte of the 2nd operand is also called the immediate operand. The immediate data may be specified as a): a character (EBCDIC code) operand as in MVI FIELD,C'A', b): a hexadecimal operand as in MVI FIELD,X'C1', or c): a binary operand as in MVI FIELD,B'11000001'. The immediate data to be moved is contained within the instruction itself. The length of the target-field is not of interest. Only the 1st byte of the field will be overwritten.

Condition codes

Remains unchanged!

Notes

Move Character Long

Mnemonic	Operands
MVCL	**R1, R2**

0E	**R1**	**R2**
0	8	12 15

Instruction Format : RR

Data transfers between two large storage locations are performed by the MVCL (MoVe Character Long) instruction. It moves data from the source-field (designated by the 2nd-operand register-pair) into the target-field (designated by the 1st-operand register-pair). The instruction recommends even-odd pairs of registers. Data movement proceeds from left to right, byte by byte, and stops as soon as the end of the target-field is reached. If source-field is shorter than the target-field, the remaining rightmost byte positions of the target-field are filled with padding bytes. If the source-field is larger than the target-field, the remaining bytes of the source-field are ignored. The result is indicated in the condition code which may be checked subsequently. The source-field remains unchanged.

Condition codes

0	operand lengths are equal
1	1st operand length is lower than the 2nd operand length
2	1st operand length is greater than the 2nd operand length
3	Destructive overlap, no movement performed

Notes

Move Numeric

Mnemonic	Operands
MVN	**D1 (L1, B1), D2 (B2)**

D1	**L1**	**B1**	**D1**	**B2**	**D2**
0	8	16	20	32	36 47

Instruction Format : SS

To transfer the four low-order bits (numeric bits, numeric digit) between two storage locations the MVN (MoVe Numerics) instruction is performed. It moves the numeric data from the source-field (2nd operand) to the corresponding positions of the target-field (1st operand). The maximum length which can be handled is 256 bytes of data. The length is defined by the 1st operand, a): explicitly as in MVN FIELDX(24),FIELDZ, or b): implicitly using the length defined at field definition. If the specified length is less than the length of the source-field (2nd operand), the low-order bytes of data are lost. The length of the source-field is not of interest. Numeric digit movement proceeds from left to right, numeric digit by numeric digit. The source-field and the zone digits of the target-field remain unchanged.
See also the MVZ instruction.

Condition codes

Remains unchanged!

Notes

Move with Offset

Mnemonic	Operands
MVO	**D1 (L1, B1), D2 (L2, B2)**

F1	L1	L2	B1	D1	B2	D2
0	8	12	16	20	32	36 47

Instruction Format : SS

The last of the decimal move instructions, the MVO (MoVe with Offset) instruction, is the most complex. The data of the source-field (2nd operand) is moved and adjusted to the left of the low-order digit of the target-field (1st operand). The assembler must know the length of each operand field, which may be up to 16 (!) bytes long. If the length of the 1st-operand (receiving) field is too small, the leftmost positions of the 2nd-operand (sending) field are truncated. If the length of the 1st-operand field is too large, the field is padded with zeros. Data movement proceeds from right to left, digit by digit. The source-field and the low-order digit of the target-field remain unchanged.

Condition codes

Remains unchanged!

Notes

Move Zone

Mnemonic	Operands
MVZ	**D1 (L1, B1), D2 (B2)**

D3	**L1**	**B1**	**D1**	**B2**	**D2**
0	8	16	20	32	36 47

Instruction Format : SS

To transfer the four high-order bits (zone bits, zone digit) between two storage locations the MVZ (MoVe Zones) instruction is performed. It moves the zone data from the source-field (2nd operand) to the corresponding positions of the target-field (1st operand). The maximum length which can be handled is 256 bytes of data. The length is defined by the 1st operand, a): explicitly as in MVZ FIELDA(15),FIELDB, or b): implicitly using the length defined at field definition. If the specified length is less than the length of the source-field (2nd operand), the low-order bytes of data are lost. The length of the source-field is not of interest. Zone digit movement proceeds from left to right, zone digit by zone digit. The source-field and the numeric digits of the target-field remain unchanged.

Condition codes

Remains unchanged!

Notes

COMPARE INSTRUCTIONS

Compare Register

Mnemonic	Operands
CR	**R1, R2**

19	**R1**	**R2**
0	8	12 15

Instruction Format : **RR**

To compare the arithmetic values of two registers, the CR (Compare Register) instruction is performed. The data of the 1st-operand register is compared with the data of the 2nd-operand register. The result of the comparison is indicated in the condition code which may be checked subsequently. The operands are treated as 32-bit signed binary integers and remain unchanged.

Condition codes

0	Operands are equal
1	1st operand register is lower than the 2nd operand register
2	1st operand register is greater than the 2nd operand register
3	Not used

Notes

Compare Fullword

Mnemonic	Operands
C	**R1, D2 (X2, B2)**

59	**R1**	**X2**	**B2**	**D2**
0	8	12	16	20 … 31

Instruction Format : **RX**

To compare the arithmetic value of a register with the value of a four byte field in storage, the C (Compare fullword) instruction is performed. The data of the 1st-operand register is compared with the data of the 2nd-operand field, a fullword (boundary!) in storage. The result of the comparison is indicated in the condition code which may be checked subsequently. The operands are treated as 32-bit signed binary integers and remain unchanged.

Condition codes

0	Operands are equal
1	1st operand register is lower than the 2nd operand register
2	1st operand register is greater than the 2nd operand register
3	Not used

Notes

Compare Halfword

Mnemonic	Operands
CH	**R1, D2 (X2, B2)**

49	**R1**	**X2**	**B2**	**D2**
0	8	12	16	20 31

Instruction Format : RX

To compare the arithmetic value of a register with the value of a two byte field in storage, the CH (Compare Halfword) instruction is performed. It will do this in two steps. First step: the data of the 2nd-operand field, a halfword (boundary!) in storage, is extended to a 32-bit interim field. The extension bits are the same as the high-order bit of the 2nd-operand field. Second step: The data of the 1st-operand register is compared with the data of the interim field. The result of the comparison is indicated in the condition code which may be checked subsequently. The 1st operand is treated as a 32-bit signed binary integer and the 2nd operand is treated as a 16-bit signed binary integer. Both operands remain unchanged.

Condition codes

0	Operands are equal
1	1st operand register is lower than the 2nd operand register
2	1st operand register is greater than the 2nd operand register
3	Not used

Notes

Compare Logical Register

Mnemonic	Operands
CLR	**R1, R2**

15	**R1**	**R2**
0	8	15

Instruction Format : RR

Logical data comparisons of two registers are performed by the CLR (Compare Logical Register) instruction. The data of the 1st-operand register is compared with the data of the 2nd-operand register. Execution proceeds from left to right, bit by bit, and stops as soon as an inequality is recognized or the end of the 1st-operand register is reached. The result is indicated in the condition code which may be checked subsequently. The sign of the operands does not affect the result of the comparison. Both operands remain unchanged.

Condition codes

0	operands are equal
1	1st operand register is lower than the 2nd operand register
2	1st operand register is greater than the 2nd operand register
3	Not used

Notes

Compare Logical Fullword

Mnemonic	Operands
CL	**R1, D2 (X2, B2)**

55	R1	X2	B2	D2
0	8	12	16	20 31

Instruction Format : RX

To compare logically the contents of a register with the contents of a four byte field in storage, the CL (Compare Logical fullword) instruction is performed. The data of the 1st-operand register is compared with the data of the 2nd-operand field, a fullword (boundary!) in storage. Execution proceeds from left to right, bit by bit, and stops as soon as an inequality is recognized or the end of the 1st-operand register is reached. The result is indicated in the condition code which may be checked subsequently. The sign of the operands does not affect the result of the comparison. Both operands remain unchanged.

Condition codes

0	operands are equal
1	1st operand register is lower than the 2nd operand register
2	1st operand register is greater than the 2nd operand register
3	Not used

Notes

Compare Logical Character

Mnemonic	Operands
CLC	**D1 (L1 , B1) , D2 (B2)**

D5	**L1**	**B1**	**D1**	**B2**	**D2**

0 8 16 20 32 36 47

Instruction Format : SS

Logical data comparisons of two storage locations are performed. The data of the 1st-operand field is compared with the data of the 2nd-operand field. Execution proceeds from left to right, bit by bit, and stops as soon as an inequality is recognized or the end of the 1st-operand field is reached. The length is defined by the 1st-operand field. The result is indicated in the condition code which may be checked subsequently. The sign of the operands does not affect the result of the comparison. Both operands remain unchanged.

Condition codes

0	operands are equal
1	1st operand field is lower than the 2nd operand field
2	1st operand field is greater than the 2nd operand field
3	Not used

Notes

Compare Logical Immediate

Mnemonic	Operands
CLI	**D1 (B1) , I2**

95	**I2**	**B1**	**D1**
0	8	16	20 31

Instruction Format : SI

Logical data comparisons of two storage locations are performed. The data of the 1st-operand field is compared with the data of the 2nd-operand field. Execution proceeds from left to right, bit by bit, and stops as soon as an inequality is recognized or the end of the 1st-operand field is reached. The length is defined by the 1st-operand field. The result is indicated in the condition code which may be checked subsequently. The sign of the operands does not affect the result of the comparison. Both operands remain unchanged.

Condition codes

0	operands are equal
1	1st operand field is lower than the 2nd operand field
2	1st operand field is greater than the 2nd operand field
3	Not used

Notes

Compare Logical Character Long

Mnemonic	Operands
CLCL	**R1, R2**

0F	R1	R2
0	8	12 15

Instruction Format : RR

Logical data comparisons of two large storage locations are performed by the CLCL (Compare Logical Characters Long) instruction. The data of the 1st-operand field (designated by the 1st-operand register-pair) is compared with the data of the 2nd-operand field (designated by the 2nd-operand register-pair). The instruction recommends even-odd pairs of registers. The shorter operand field is considered to be extended on the right with padding bytes. Execution proceeds from left to right, bit by bit, and stops as soon as an inequality is recognized or the end of the larger operand field is reached. The result is indicated in the condition code which may be checked subsequently. Both operands remain unchanged.

Condition codes

0	operands are equal
1	1st operand register is lower than the 2nd operand register
2	1st operand register is greater than the 2nd operand register
3	Not used

Notes

Test Under Mask

Mnemonic	Operands
TM	**D1, B1 (I2)**

91	**I2**	**B1**	**D1**
0	8	16	20 31

Instruction Format : SI

To test the bits of a single byte of data, whether they are on or off, the TM (Test under Mask) instruction is performed. The bits of the 1st-operand field are selected and tested by an eight-bit mask, given by the 2nd-operand. A mask-bit of 'one' indicates, that the corresponding storage-bit is to be tested. If the mask-bit is zero, the storage-bit is ignored. The 2nd operand is also called the immediate operand. The immediate data (mask), defined at program coding, is contained within the instruction itself. The length of the 1st-operand field is not of interest. Only the 1st byte of the field is tested. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	All selected bits are zero
1	Selected bits are mixed
2	Not used
3	All selected bits are one.

Notes

CHAPTER 1010 BRANCH INSTRUCTIONS

CONDITIONAL BRANCH

Branch on Condition

Mnemonic	Operands
BC	**M1,D2(X2,B2)**

47	**M1**	**X2**	**B2**	**D2**	
0	8	12	16	20	31

Instruction Format : RX

To react on a condition code, which was previously set in the Program Status Word (PSW), the BC (Branch on Condition) instruction is performed. A 4-bit binary mask (1st operand) is used to test the condition code. If the condition is true, a branch is made to the target-location (2nd operand) given by a label; otherwise the next sequential instruction (NSI) is processed. Instead of coding the 1st-operand mask as in BC 8,LABEL , very often the extended mnemonic instructions are coded as in BZ LABEL .

Condition codes

Remains Unchanged.

Notes

Branch on Condition Register

Mnemonic	Operands
BCR	**R1, R2**

07	R1	R2
0	8	12 15

Instruction Format : RR

To react on a condition code, which was previously set in the Program Status Word (PSW), the BCR (Branch on Condition Register) instruction is performed. A 4-bit binary mask (1st operand) is used to test the condition code. If the condition is true, a branch is made to the target-location (2nd operand) given by the address within a register; otherwise the next sequential instruction (NSI) is processed. Instead of coding the 1st-operand mask as in BCR 11,R7 , very often the extended mnemonic instructions are coded as in BNMR R7 .

Condition codes

Remains Unchanged.

Notes

Branch Instructions (Extended Mnemonic)

Extended Code	Meaning	Machine Instruction	Binary Mask
B or BR	Branch Unconditionally	BC or BCR 15	B’1111’
BZ or BZR	Branch on Zero	BC or BCR 8	B’1000’
BM or BMR	Branch on Negative	BC or BCR 4	B’0100’
BP or BPR	Branch on Positive	BC or BCR 2	B’0010’
BO or BOR	Branch on Overflow	BC or BCR 1	B’0001’
BNZ or BNZR	Branch on not Zero	BC or BCR 7	B’0111’
BNM or BNMR	Branch on not Negative	BC or BCR 11	B’1011’
BNP or BNPR	Branch on not Positive	BC or BCR 13	B’1101’
BNO or BNOR	Branch on not Overflow	BC or BCR 14	B’1110’
BE or BER	Branch on Equal	BC or BCR 8	B’1000’
BL or BLR	Branch or Low	BC or BCR 4	B’0100’
BH or BHR	Branch or High	BC or BCR 2	B’0010’
BNE or BNER	Branch on not Equal	BC or BCR 7	B’1110’
BNL or BNLR	Branch on not Low	BC or BCR 11	B’1011’
BNH or BNHR	Branch on not High	BC or BCR 13	B’1101’

Notes

UNCONDITIONAL BRANCH

Branch And Save

Mnemonic	Operands
BAS	**R1,D2(X2,B2)**

4D	**R1**	**X2**	**B2**	**D2**	
0	8	12	16	20	31

Instruction Format : RX

Unconditional branching within a program and the possibility to return to the branch-point is performed by the BAS (Branch And Save) instruction. The address of the next sequential instruction (NSI) is placed in the 1st-operand register. Subsequently a branch is made to the location given by a 2nd-operand label. The instruction(s) is commonly used to branch to a subroutine with an expected return. Now to return, the help of the unconditional BR (Branch Register) instruction is needed

Condition codes

Remains Unchanged.

Notes

Branch And Save Register

Mnemonic	Operands
BASR	**R1, R2**

0D	**R1**	**R2**
0	8	12 15

Instruction Format : RR

Unconditional branching within a program and the possibility to return to the branch-point is also performed by the BASR (Branch And Save Register) instruction. The address of the next sequential instruction (NSI) is placed in the 1st-operand register. Subsequently a branch is made to the location given by the address within the 2nd-operand register. The instruction(s) is commonly used to branch to a subroutine with an expected return. Now to return, the help of the unconditional BR (Branch Register) instruction is needed.

Condition codes

Remains Unchanged.

Notes

Branch on Count

Mnemonic	Operands
BCT	**R1,D2(X2,B2)**

46	**R1**	**X2**	**B2**	**D2**
0	8	12	16	20 31

Instruction Format : RX

One possibility to control a loop is the do times option. The instruction BCT (Branch on CounT) will handle this in following steps. First step: the contents of the 1st-operand register is subtracted by one. Second step: The result is checked for zero. Third step: If the result is not zero a branch is made to the location given by the 2nd-operand label; if the result is zero the next sequential instruction (NSI) is processed.

Condition codes

Remains Unchanged.

Notes

Branch on Count Register

Mnemonic	Operands
BCTR	**R1, R2**

06	**R1**	**R2**
0	8	12 15

Instruction Format : RR

One possibility to control a loop is the do times option. Also the instruction BCTR (Branch on CounT Register) will handle this in following steps. First step: the contents of the 1st-operand register is subtracted by one. Second step: The result is checked for zero. Third step: If the result is not zero a branch is made to the location given by the address within the 2nd-operand register; if the result is zero the next sequential instruction (NSI) is processed.

Condition codes

Remains Unchanged.

Notes

Branch on Index Low or Equal

Mnemonic	Operands
BXLE	**R1,R2,D3(B3)**

87	**R1**	**R2**	**B3**	**D3**
0	8	12	16	20 31

Instruction Format : RX

One possibility to control a loop is the do until option. The instruction BXLE (Branch on indeX Low or Equal) will handle this in the following steps.
First step : An increment is added top the 1st operand index-register.
Second step : The result is compared with a comparand.
Third step : The result of the comparison determines if a branch occurs or not. If the value of the index-register is low or equal to the value of the comparand a branch is made to the location given by the 2nd operand label; if it is higher the next sequential instruction (NSI) is processed. The increment and the comparand values are designated by the 2nd operand even-odd register pair.

Condition codes

Remains Unchanged.

Notes

OR Register

Mnemonic	Operands
OR	**R1, R2**

16	R1	R2
0	8	12 15

Instruction Format : RR

To set single bits of a register to 'one', the OR (Or Register) instruction is performed. The 'zero'-bits of the 1st-operand register are changed to 'one', if the corresponding bits of the 2nd-operand register are 'one'. In all other cases the bits of the 1st-operand register remain unchanged. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	1st operand register is zero
1	1st operand register is not zero
2	Not used
3	Not used

Notes

OR Full Word

Mnemonic	Operands
O	**R1,D2(X2,B2)**

56	R1	X2	B2	D2
0	8	12	16	20 … 31

Instruction Format : RX

To set single bits of a register to 'one', the O (Or full word) instruction is performed. The 'zero'-bits of the 1st-operand register are changed to 'one', if the corresponding bits of the 2nd-operand field, a full word (boundary!) in storage, are 'one'. In all other cases the bits of the 1st-operand register remain unchanged. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	1st operand register is zero
1	1st operand register is not zero
2	Not used
3	Not used

Notes

OR Character

Mnemonic	Operands
OC	**D1 (L1 , B1) , D2 (B2)**

D6	**L1**	**B1**	**D1**	**B2**	**D2**
0	8	16	20	32	36 47

Instruction Format : SS

To set single bits of a storage field to 'one', the OC (Or Character) instruction is performed. The 'zero'-bits of the 1st-operand field are changed to 'one', if the corresponding bits of the 2nd-operand field are 'one'. In all other cases the bits of the 1st-operand field remain unchanged. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	1st operand field is zero
1	1st operand field is not zero
2	Not used
3	Not used

Notes

OR Immediate

Mnemonic	Operands
OI	**D1 (B1) , I2**

96	I2	B1	D1
0	8	16	20 31

Instruction Format : SI

To set the bits of a single byte of data to 'one', the OI (Or Immediate) instruction is performed. The 'zero'-bits of the 1st-operand field are changed to 'one', if the corresponding bits of the 2nd-operand field are 'one'. In all other cases the bits of the 1st-operand field remain unchanged. The 2nd operand is also called the immediate operand. The immediate data, defined at program coding, is contained within the instruction itself. The length of the 1st-operand field is not of interest. Only the 1st byte of the field is considered. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	1st operand byte is zero
1	1st operand byte is not zero
2	Not used
3	Not used

Notes

AND Register

Mnemonic	Operands
NR	**R1, R2**

14	**R1**	**R2**

0 8 12 15

Instruction Format : **RR**

To set single bits of a register to 'zero', the NR (aNd Register) instruction is performed. The 'one'-bits of the 1st-operand register are changed to 'zero', if the corresponding bits of the 2nd-operand register are 'zero'. In all other cases the bits of the 1st-operand register remain unchanged. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	1st operand register is zero
1	1st operand register is not zero
2	Not used
3	Not used

Notes

AND Full Word

Mnemonic	Operands
N	**R1,D2(X2,B2)**

54	R1	X2	B2	D2
0	8	12	16	20 … 31

Instruction Format : RX

To set single bits of a register to 'zero', the N (aNd full word) instruction is performed. The 'one'-bits of the 1st-operand register are changed to 'zero', if the corresponding bits of the 2nd-operand field, a full word (boundary!) in storage, are 'zero'. In all other cases the bits of the 1st-operand register remain unchanged. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	1st operand register is zero
1	1st operand register is not zero
2	Not used
3	Not used

Notes

AND Character

Mnemonic	Operands
NC	D1 (L1 , B1) , D2 (B2)

D4	L1	B1	D1	B2	D2
0	8	16	20	32	36 47

Instruction Format : SS

To set single bits of a storage field to 'zero', the NC (aNd Character) instruction is performed. The 'one'-bits of the 1st-operand field are changed to 'zero', if the corresponding bits of the 2nd-operand field are 'zero'. In all other cases the bits of the 1st-operand field remain unchanged. The result is indicated in the condition code which may be checked subsequently .

Condition codes

0	1st operand field is zero
1	1st operand field is not zero
2	Not used
3	Not used

Notes

AND Immediate

Mnemonic	Operands
NI	**D1 (B1) , I2**

94	**I2**	**B1**	**D1**
0	8	16 20	31

Instruction Format : **SI**

To set the bits of a single byte of data to 'zero', the NI (aNd Immediate) instruction is performed. The 'one'-bits of the 1st-operand field are changed to 'zero', if the corresponding bits of the 2nd-operand field are 'zero'. In all other cases the bits of the 1st-operand field remain unchanged. The 2nd operand is also called the immediate operand. The immediate data, defined at program coding, is contained within the instruction itself. The length of the 1st-operand field is not of interest. Only the 1st byte of the field is considered. The result is indicated in the condition code which may be checked subsequently. Only the 1st byte of the field is considered.

Condition codes

0	1st operand byte is zero
1	1st operand byte is not zero
2	Not used
3	Not used

Notes

Exclusive OR Register

Mnemonic	Operands
XR	**R1, R2**

0	8	12 15
17	**R1**	**R2**

Instruction Format : RR

To set single bits of a register to 'one' or 'zero' in single step, the XR (eXclusive or Register) instruction is performed. The 'one'-bits of the 1st-operand register are changed to 'zero', respectively the 'zero'-bits are changed to 'one', if the corresponding bits of the 2nd-operand register are 'one'. In all other cases the bits of the 1st-operand register remain unchanged. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	1st operand register is zero
1	1st operand register is not zero
2	Not used
3	Not used

Notes

Exclusive OR Full Word

Mnemonic	Operands
X	**R1,D2(X2,B2)**

57	R1	X2	B2	D2
0	8	12	16	20 31

Instruction Format : RX

To set single bits of a register to 'one' or to 'zero' in one step, the X (eXclusive or fullword) instruction is performed. The 'one'-bits of the 1st-operand register are changed to 'zero', respectively the 'zero'-bits are changed to 'one', if the corresponding bits of the 2nd-operand field, a fullword (boundary!) in storage, are 'one'. In all other cases the bits of the 1st-operand register remain unchanged. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	1st operand register is zero
1	1st operand register is not zero
2	Not used
3	Not used

Notes

Exclusive OR Character

Mnemonic	Operands
XC	**D1 (L1 , B1) , D2 (B2)**

D7	**L1**	**B1**	**D1**	**B2**	**D2**
0	8	16	20	32	36 47

Instruction Format : SS

To set single bits of a storage field to 'one' or 'zero' in one step, the XC (eXclusive or Character) instruction is performed. The 'one'-bits of the 1st-operand field are changed to 'zero', respectively the 'zero'-bits are changed to 'one', if the corresponding bits of the 2nd-operand field are 'one'. In all other cases the bits of the 1st-operand field remain unchanged. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	1st operand field is zero
1	1st operand field is not zero
2	Not used
3	Not used

Notes

Exclusive OR Immediate

Mnemonic	Operands
XI	**D1 (B1) , I2**

97	I2	B1	D1
0	8	16	20 — 31

Instruction Format : **SI**

To set the bits of a single byte of data to 'one' or to 'zero' in one step, the XI (eXclusive or Immediate) instruction is performed. The 'one'-bits of the 1st-operand field are changed to 'zero', respectively the 'zero'-bits are changed to 'one', if the corresponding bits of the 2nd-operand field are 'one'. In all other cases the bits of the 1st-operand field remain unchanged The 2nd operand is also called the immediate operand. The immediate data, defined at program coding, is contained within the instruction itself. The length of the 1st-operand field is not of interest. Only the 1st byte of the field is considered. The result is indicated in the condition code which may be checked subsequently.

Condition codes

0	1st operand byte is zero
1	1st operand byte is not zero
2	Not used
3	Not used

Notes

Shift Left Logical

Mnemonic	Operands
SLL	**R1,D2(B2)**

89	**R1**	**///**	**B2**	**D2**	
0	8	12	16	20	31

Instruction Format : **RS**

The data of a register has to be shifted to the left by a number of positions. The SLL (Shift Left single Logical) instruction handles this problem. The number of positions (maximum 63) to be shifted is given in the immediate 2nd-operand, also called as shift value. All 32 bits of the 1st-operand register are included in the left-shift. If the binary value of the 1st-operand register is shifted to the left by one position, the value is doubled. During the left-shift zeros are inserted on the right of the 1st-operand register. Bits shifted out of the leftmost position are lost.

Condition codes

Remains Unchanged.

Notes

Shift Right Logical

Mnemonic	Operands
SRL	**R1,D2(B2)**

88	**R1**	**///**	**B2**	**D2**
0	8	12	16	20 31

Instruction Format : RS

The data of a register has to be shifted to the right by a number of positions. The SRL (Shift Right single Logical) instruction handles this problem. The number of positions (maximum 63) to be shifted is given in the immediate 2nd-operand, also called as shift value. All 32 bits of the 1st-operand register are included in the right-shift. If the binary value of the 1st-operand register is shifted to the right by one position, the value is cut in halve. During the right-shift zeros are inserted on the left of the 1st-operand register. Bits shifted out of the rightmost position are lost.

Condition codes

Remains Unchanged.

Notes

Shift Left Arithmetic

Mnemonic	Operands
SLA	**R1,D2(B2)**

8B	**R1**	**///**	**B2**	**D2**

0 8 12 16 20 31

Instruction Format : **RS**

The data of a register has to be shifted to the left by a number of positions. The SLA (Shift Left single Arithmetic) instruction handles this problem. The number of positions (maximum 63) to be shifted is given in the immediate 2nd-operand, also called as shift value. The 1st-operand register is treated as a 32-bit signed binary integer. If the binary value of the 1st-operand register is shifted to the left by one position, the value is doubled. During the left-shift zeros are inserted on the right of the 1st-operand register. The sign bit is not shifted! The result is tested and a condition code is set. If one or more bits unlike the sign bit are shifted out of the leftmost position, an overflow occurs and additionally the condition code is set to 3.

Condition codes

0	1st operand register is zero
1	1st operand register is negative
2	1st operand register is positive
3	LSB(next to sign bit) is shifted out to the left

Notes

Shift Right Arithmetic

Mnemonic	Operands
SRA	**R1,D2(B2)**

8A	R1	///	B2	D2
0	8	12	16	20 – 31

Instruction Format : RS

The data of a register has to be shifted to the right by a number of positions. The SRA (Shift Right single Arithmetic) instruction handles this problem. The number of positions (maximum 63) to be shifted is given in the immediate 2nd-operand, also called as shift value. The 1st-operand register is treated as a 32-bit signed binary integer. If the binary value of the 1st-operand register is shifted to the right by one position, the value is cut in halve. During the right-shift, bits equal to the sign bit are inserted on the left of the 1st-operand register. Only the 31 rightmost bits are shifted. The sign bit (first bit) remains unchanged! The result is tested and a condition code is set. Bits shifted out of the rightmost position are lost.

Condition codes

0	1st operand register is zero
1	1st operand register is negative
2	1st operand register is positive
3	Not used

Notes

Convert to Binary

Mnemonic	Operands
CVB	**R1,D2(X2,B2)**

4F	R1	X2	B2	D2
0	8	12	16	20 31

Instruction Format : RX

The conversion of packed decimal data to binary data is performed by the CVB (ConVert to Binary) instruction. It converts the packed data of the source-field (2nd operand) to binary data and places the result in the target register (1st operand). The source-field has to be a double word (boundary!) in storage. The result is treated as a 32-bit signed binary integer. The maximum positive number that can be converted is 2,147,483,647(X'7FFFFFFF'), the maximum negative number is -2,147,483,648(X'80000000'). If the source-field does not contain packed decimal data, an error occurs (Data Exception). The source-field remains unchanged.

Condition codes

Remains Unchanged.

Notes

Convert to Decimal

Mnemonic	Operands
CVD	**R1,D2(X2,B2)**

4E	R1	X2	B2	D2
0	8	12	16	20 … 31

Instruction Format : **RX**

The conversion of binary data to packed decimal data is performed by the CVD (ConVert to Decimal) instruction. It converts the binary data of the source-register (1st operand) to packed data and places the result at the target-field (2nd operand). The target-field has to be a double word(boundary!) in storage. The source-data is treated as a 32-bit signed binary integer. The result has the format of packed decimal data (coded positive sign-digit is C ; coded negative sign-digit is D). The source-register remains unchanged.

Condition codes

Remains Unchanged.

Notes

Pack

Mnemonic	Operands
PACK	**D1 (L1 , B1) , D2 (B2)**

F2	L1	L2	B1	D1	B2	D2
0	8	12	16	20	32	36 47

Instruction Format : **SS**

The PACK instruction converts decimal data to packed format. The instruction takes the data from the source-field (2nd operand) and packs it to the target-field (1st operand). The assembler must know the length of each operand field, which may be up to 16 (!) bytes long. The instruction proceeds from right to left. If the length of the 1st-operand (receiving) field is too small, the leftmost positions of the 2nd-operand (sending) field are truncated. If the length of the 1st-operand field is too large, the field is padded with zeros. After execution, the low-order digit of the target-field signals the sign of the packed decimal value (Positive: A, C, E, F ; Negative: B, D). The source-field remains unchanged

Condition codes

Remains Unchanged.

Notes

Unpack

Mnemonic	Operands
UNPK	**D1 (L1 , B1) , D2 (B2)**

F3	L1	L2	B1	D1	B2	D2
0	8	12	16	20	32	36

47

Instruction Format : **SS**

The UNPK (UNPacK) instruction converts packed decimal data to zoned format. The packed decimal data of the source-field (2nd operand) is converted to zoned-format data and placed in the target-field (1st operand). The assembler must know the length of each operand field, which may be up to 16 bytes long. Execution proceeds from right to left. If the length of the 1st-operand (receiving) field is too small, the leftmost positions of the 2nd-operand (sending) field are truncated. If the length of the receiving field is too large, decimal zeros are added. The source-field remains unchanged.

Condition codes

Remains Unchanged.

Notes

Add Pack

Mnemonic	Operands
AP	**D1 (L1 , B1) , D2 (B2)**

FA	**L1**	**L2**	**B1**	**D1**	**B2**	**D2**
0	8	12	16	20	32	36 47

Instruction Format : SS

Two fields containing packed decimal data have to be added. The instruction AP (Add Packed decimal data) will do this in one step. The contents of the 2nd-operand field are added to the contents of the 1st-operand field and the resulting sum is placed in the 1st-operand field. The initial contents of both operand fields have to be in correct packed decimal format, otherwise the instruction results in a system error (Data Exception). The result is also in packed decimal format. The 2nd-operand field remains unchanged. The calculation follows algebraic rules. The result is tested and a condition code is set. In case of an overflow the overflow digits are ignored and additionally the condition code is set to 3.

Condition codes

0	Result is Zero
1	Result is Negative
2	Result is Positive
3	Result is Overflow

Notes

Subtract Pack

Mnemonic	Operands
SP	**D1 (L1 , B1) , D2 (B2)**

FB	**L1**	**L2**	**B1**	**D1**	**B2**	**D2**
0	8	12	16	20	32	36 47

Instruction Format : SS

Two fields containing packed decimal data have to be subtracted from each other. The instruction SP (Subtract Packed decimal data) will do this in one step. The contents of the 2nd-operand field are subtracted from the contents of the 1st-operand field and the result is placed in the 1st-operand field. The initial contents of both operand fields have to be in correct packed decimal format, otherwise the instruction results in a system error (Data Exception). The result is also in packed decimal format. The 2nd-operand field remains unchanged. The calculation follows algebraic rules. The result is tested and a condition code is set. In case of an overflow the overflow digits are ignored and additionally the condition code is set to 3.

Condition codes

0	Result is Zero
1	Result is Negative
2	Result is Positive
3	Result is Overflow

Notes

Zero and Add Pack

Mnemonic	Operands
ZAP	**D1 (L1 , B1) , D2 (B2)**

F8	L1	L2	B1	D1	B2	D2
0	8	12	16	20	32	36 47

Instruction Format : **SS**

To set up packed decimal data in an other field, the ZAP (Zero and Add Packed decimal data) instruction is performed. First the 1st-operand field is set to packed decimal zero. Then the contents of the 2nd-operand field is added to the 1st-operand zero value. The initial contents of the 2nd-operand field has to be in correct packed decimal format, otherwise the instruction results in a system error (Data Exception). The result is also in packed decimal format. The 2nd-operand field remains unchanged. The calculation follows algebraic rules. The result is tested and a condition code is set. In case of an overflow the overflow digits are ignored and additionally the condition code is set to 3

Condition codes

0	Result is Zero
1	Result is Negative
2	Result is Positive
3	Result is Overflow

Notes

Multiply Pack

Mnemonic	Operands
MP	**D1 (L1 , B1) , D2 (B2)**

FC	**L1**	**L2**	**B1**	**D1**	**B2**	**D2**
0	8	12	16	20	32	36 47

Instruction Format : SS

Two fields containing packed decimal data have to be multiplied with each other. The instruction MP (Multiply Packed decimal data) will do this in one step. The contents of the 1st-operand field (multiplicand) is multiplied with the contents of the 2nd-operand field (multiplier) and the result is placed in the 1st-operand field. The initial contents of both operand fields have to be in correct packed decimal format, otherwise the instruction results in a system error (Data Exception). The result is also in packed decimal format. The 2nd-operand field remains unchanged. The multiplicand must have at least as many bytes of leading zeros as the number of bytes in the multiplier. The calculation follows algebraic rules. The result is tested and a condition code is set. In case of an overflow the overflow digits are ignored.

Condition codes

Remains Unchanged.

Notes

Divide Pack

Mnemonic	Operands
DP	**D1 (L1 , B1) , D2 (B2)**

FD	L1	L2	B1	D1	B2	D2
0	8	12	16	20	32	36 47

Instruction Format : SS

Two fields containing packed decimal data have to be divided by each other. The instruction DP (Divide Packed decimal data) will do this in one step. The contents of the 1st-operand field (dividend) is divided by the contents of the 2nd-operand field (divisor) and the result (quotient and remainder) is placed in the 1st-operand field. The initial contents of both operand fields have to be in correct packed decimal format, otherwise the instruction results in a system error (Data Exception). The results is also in packed decimal format. The 2nd-operand field remains unchanged. The quotient, in the length of the difference between dividend and divisor (L1-L2!), is placed leftmost in the 1st-operand field. The remainder, in the length of the divisor (L2!), is placed rightmost in the 1st-operand field. The calculation follows algebraic rules. Note: The divisor may not be greater than 15 digits plus sign (L2 << 8).

Condition codes

Remains Unchanged.

Notes

Execute

Mnemonic	Operands
EX	**R1,D2(X2,B2)**

44	R1	X2	B2	D2

0 8 12 16 20 31

Instruction Format : RX

To perform SS1 and SS2 instructions with a variable length, the EX (EXecute) instruction is recommended. First the 2nd byte of the instruction at the 2nd-operand address is modified by the rightmost byte of the 1st-operand register (that 2nd byte contains the length attributes!). Afterwards the resulting instruction, called as the target instruction, is executed. Be careful using the EX instruction in connection with other instruction formats! Note: Do NOT use R0 as 1st-operand register ! In this case no OR-connection will be made.

Condition codes

Remains Unchanged.

Notes

Translate

Mnemonic	Operands
TR	**D1 (L1 , B1) , D2 (B2)**

DC	**L1**	**B1**	**D1**	**B2**	**D2**
0	8	16	20	32	36 47

Instruction Format : SS

To translate a hexadecimal value in a field to printable characters, the TR (TRanslate) instruction is performed. Each byte of the 1st-operand field is used as an eight-bit argument to reference a function byte within the 2nd-operand field. First the contents of the argument-byte is added to the initial 2nd-operand address to point at a function-byte. After that the found function-byte replaces the active argument-byte in the 1st-operand field. The instruction is processed from left to right, byte by byte.

Condition codes

Remains Unchanged.

Notes

Translate and Test

Mnemonic	Operands
TRT	**D1 (L1 , B1) , D2 (B2)**

DD	**L1**	**B1**	**D1**	**B2**	**D2**	
0	8	16	20	32	36	47

Instruction Format : **SS**

To check a numeric, an alpha or an alphanumeric field for right characters, the TRT (TRanslate and Test) instruction is performed. Each byte of the 1st-operand field is used as an eight-bit argument to reference a function byte within the 2nd-operand field. The contents of the argument-byte is added to the initial 2nd-operand address to point at a function-byte. The processing stops immediately if the found function-byte is not zero (X'00') or the end of the 1st-operand field is reached. The found non-zero function-byte is inserted in the rightmost byte of the register R2 and additionally the related argument-byte address (within the 1st-operand field) is placed into the register R1. If all found function-bytes are zero, the contents of registers R1 and R2 remain unchanged. The result of the instruction is also indicated in the condition code which may be checked subsequently. The instruction is processed from left to right, byte by byte.

Condition codes

0	All function bytes found are zero
1	A nonzero function byte is found
2	A nonzero function byte is found and it’s the last byte of the 1st operand field
3	Not used

Notes

APPENDIX C1

EBCDIC – Alpha Numeric Characters

	1	2	3	4	5	6	7	8	9
C	A	B	C	D	E	F	G	H	I
D	J	K	L	M	N	O	P	Q	R
E		S	T	U	V	W	X	Y	Z
F	1	2	3	4	5	6	7	8	9

EBCDIC – Special Characters

	A	B	C	D	E	F	0	1	
4		.	<	(	+	|			
5	!	$	*	)	;		&		
6	|	,	%	_	>	?	-	/	
7	:	#	@	‘	=	“			

APPENDIX C2

Special Character Equates			
#CR	Carriage Return	X’15’	
#SP	Blank	X’40’	C’ ‘
#DOT	Dot	X’4B’	C’.’
#EOM	End of message	X’4E’	C’+’
#END	Enditem (!,#)	X’4F’	C’!’
#FC	Asterisk	X’5C’	C’*’
#HYPH	Dash	X’60’	C’-‘
#SLASH	Slash	X’61’	C’/’
#SOM	Start of message	X’6E’	C’>’
#CHA	At-sign	X’7C’	C’@’
#APO	Apostrophe	X’7D’	C’’’

APPENDIX C3

Define Constant

Mnemonic	Operands
DC	**Name; Storage type; Length; Value**

Examples

LABEL1	DC	C'/'	L' = 1	X'61'
LABEL2	DC	X'F9AB'	L' = 2	X'F9AB'
LABEL3	DC	H'100'	L' = 2	X'0064'
LABEL4	DC	F'-2'	L' = 4	X'FFFFFFFE'
LABEL5	DC	2CL2'GAG'	L' = 2	X'C7C1C7C1'
LABEL6	DC	CL5'NO'	L' = 5	X'D5D6404040'
LABEL7	DC	FL3'12'	L' = 3	X'00000C'
LABEL8	DC	3XL2'3A8'	L' = 2	X'03A803A803A8'
LABEL9	DC	3FL2'-3'	L' = 2 (3x)	X'FFFDFFFDFFFD'
LABEL10	DC	A(22)	L' = 4	X'00000016'
LABEL11	DC	P'35'	L' = 2	X'035C'
LABEL12	DC	AL2(-1)	L' = 2	X'FFFF'
LABEL13	DC	2P'-4'	L' = 1 (2x)	X'4D4D'
LABEL14	DC	BL2'01101'	L' = 2	X'000D'
LABEL15	DC	AL1(C'R',X'40',B'101'	L' = 1 (3x)	X'D94005'
LABEL16	DC	CL4'ADE'	L' = 5	X'C1C4C54040'

Define Storage

Mnemonic	Operands
DS	**Name; Storage type; Length;**

Examples

LABEL1	DS	C	Implicit L' = 1	No Boundary
LABEL2	DS	X	Implicit L' = 1	No Boundary
LABEL3	DS	B	Implicit L' = 1	No Boundary
LABEL4	DS	P	Implicit L' = 1	No Boundary
LABEL5	DS	H	Implicit L' = 2	Boundary (2 bytes)
LABEL6	DS	F	Implicit L' = 4	Boundary (4 bytes)
LABEL7	DS	A	Implicit L' = 4	Boundary (4 bytes)
LABEL8	DS	D	Implicit L' = 8	Boundary (8 bytes)
LABEL9	DS	0CL8	Explicit L' = 8	No Boundary
LABEL10	DS	FL3	Explicit L' = 3	Boundary (4 bytes)
LABEL11	DS	3XL6	Explicit L' = 6	No Boundary
LABEL12	DS	2HL3	Explicit L' = 3	Boundary (2 bytes)
LABEL13	DS	5A	Implicit L' = 4	Boundary (4 bytes)

APPENDIX C4

Padding / Truncation

Padding	= fill up / complete
Truncation	= cut off

Data Type	Symbol	Alignment	Fill Character	Boundary
Binary	B	Right	B’0’	
Character	C	Left	X’40’	
Hex	X	Right	X’0’	
Half Word	H	Right	-	2 Bytes
Full Word	F	Right	-	4 Bytes
Double	D	Right	-	8 Bytes
Packed	P	Right	X’0’	
Zoned	Z	Right	X’F0’	

LABEL	DEFINITION	REPRESENTATION	RESULT
FIELDA	DC CL4’ABCDEF’	C1C2C3C4	TRUNCATION
FIELDB	DC CL6’1234’	F1F2F3F44040	PADDING
FIELDC	DC XL2’89ABC’	9ABC	TRUNCATION
FIELDD	DC XL4’567AB’	000567AB	PADDING
FIELDE	DC P’-1234’	01234D	PADDING
FIELDF	DC P’324’	324C	PADDING
FIELDG	DC PL2’1234’	234C	TRUNCATION
FIELDH	DC B’11001’	00011001	PADDING

APPENDIX C5

DSECT with ORG

	ADDRESS	DSECT	
000	ADDR	DS	CL80
050		ORG	ADDR
000	NAME	DS	CL30
01E		ORG	NAME
000	LNAME	DS	CL14
00E		DS	CL2
010	FNAME	DS	CL14
01E	STREET	DS	CL30
03C		ORG	ADDR+30
01E	NUMBER	DS	CL4
022	STR	DS	CL26
03C	CITY	DS	CL20
050		ORG	STR+26
03C	PCODE	DS	CL4
040	CITYN	DS	CL16
		ORG	ADDR+80
	IS	CSECT	