logo-200x40-noshadow

**IBM S/390 Assembly Language**

TPF Software Inc. Training Material

TPF Software India Private Limited

Unit 1, Block C, II Floor, Tek Meadows Campus,

51, Rajiv Gandhi Salai, Sholinganallur,

Chennai, TN, India. Pin Code : 600 119

Table of Contents

[Introduction 5](#_Toc368391609)

[IBM S/390 Architecture 6](#_Toc368391610)

[Introduction 6](#_Toc368391611)

[Main Storage 6](#_Toc368391612)

[CPU 6](#_Toc368391613)

[Main Components of the CPU 7](#_Toc368391614)

[Channel Subsystem 8](#_Toc368391615)

[S/390 Assembly Language 9](#_Toc368391616)

[Instruction 9](#_Toc368391617)

[Instruction Formats 9](#_Toc368391618)

[Storage Addressing 11](#_Toc368391619)

[Storage Operands 12](#_Toc368391620)

[Instructions 17](#_Toc368391621)

[Load and Store 14](#_Toc368391622)

[LA – Load Address 15](#_Toc368391623)

[LR – Load Register 16](#_Toc368391624)

[L – Load 17](#_Toc368391625)

[ST – Store 18](#_Toc368391626)

[LH – Load Halfword 19](#_Toc368391627)

[STH – Store Halfword 20](#_Toc368391628)

[LHI – Load Halfword Immediate 21](#_Toc368391629)

[IC – Insert Character 22](#_Toc368391630)

[STC – Store Character 23](#_Toc368391631)

[ICM – Insert Character under Mask 24](#_Toc368391632)

[STCM – Store Character under Mask 25](#_Toc368391633)

[LM – Load Multiple 26](#_Toc368391634)

[STM – Store Multiple 27](#_Toc368391635)

[LTR – Load and Test Register 28](#_Toc368391636)

[Arithmetic 33](#_Toc368391637)

[AR – Add Register 34](#_Toc368391638)

[A – Add 35](#_Toc368391639)

[AH – Add Halfword 36](#_Toc368391640)

[AHI – Add Halfword Immediate 37](#_Toc368391641)

[SR – Subtract Register 38](#_Toc368391642)

[S – Subtract 39](#_Toc368391643)

[SH – Subtract Halfword 40](#_Toc368391644)

[MR – Multiply Register 41](#_Toc368391645)

[M – Multiply 42](#_Toc368391646)

[MH – Multiply Halfword 43](#_Toc368391647)

[MHI – Multiply Halfword Immediate 44](#_Toc368391648)

[DR – Divide Register 45](#_Toc368391649)

[D – Divide 46](#_Toc368391650)

[Compare 51](#_Toc368391651)

[CR – Compare Register 52](#_Toc368391652)

[C – Compare 53](#_Toc368391653)

[CH – Compare Halfword 54](#_Toc368391654)

[CHI – Compare Halfword Immediate 55](#_Toc368391655)

[Branch 59](#_Toc368391656)

[BCR – Branch on Condition Register 60](#_Toc368391657)

[BC – Branch on Condition 61](#_Toc368391658)

[Other Branch on Condition/Register Instructions 62](#_Toc368391659)

[BCTR – Branch on Count Register 63](#_Toc368391660)

[BCT – Branch on Count 64](#_Toc368391661)

[BASR – Branch and Save Register 65](#_Toc368391662)

[BAS – Branch and Save 66](#_Toc368391663)

[Character: Move, Compare 71](#_Toc368391664)

[MVI – Move Immediate 72](#_Toc368391665)

[MVC – Move Character 73](#_Toc368391666)

[MVCL – Move Character Long 74](#_Toc368391667)

[CLI – Compare Logical Immediate 75](#_Toc368391668)

[CLC – Compare Logical Character 76](#_Toc368391669)

[CLCL – Compare Logical Character Long 77](#_Toc368391670)

[Decimal - Packed 82](#_Toc368391671)

[Packed Decimal Number 83](#_Toc368391672)

[CVB – Convert to Binary 84](#_Toc368391673)

[CVD – Convert to Decimal 85](#_Toc368391674)

[AP – Add Packed 86](#_Toc368391675)

[SP – Subtract Packed 87](#_Toc368391676)

[MP – Multiply Decimal 88](#_Toc368391677)

[DP – Divide Decimal 89](#_Toc368391678)

[CP – Compare Packed 90](#_Toc368391679)

[ZAP – Zero and Add Packed 91](#_Toc368391680)

[SRP – Shift and Round Packed 92](#_Toc368391681)

[Decimal - Zoned 93](#_Toc368391682)

[Zoned Decimal Number 94](#_Toc368391683)

[PACK– Pack 95](#_Toc368391684)

[UNPK– Unpack 96](#_Toc368391685)

[Shift 101](#_Toc368391686)

[SLL – Shift Left Logical 102](#_Toc368391687)

[SRL – Shift Right Logical 103](#_Toc368391688)

[SLA – Shift Left Arithmetic 104](#_Toc368391689)

[SRA – Shift Right Arithmetic 105](#_Toc368391690)

[Logical 110](#_Toc368391691)

[Logical Operations 111](#_Toc368391692)

[NR – AND Register 112](#_Toc368391693)

[N – AND 113](#_Toc368391694)

[NI – AND Immediate 114](#_Toc368391695)

[NC – AND Character 115](#_Toc368391696)

[OR – OR Register 116](#_Toc368391697)

[O – OR 117](#_Toc368391698)

[OI – OR Immediate 118](#_Toc368391699)

[OC – OR Character 119](#_Toc368391700)

[XR – XOR Register 120](#_Toc368391701)

[X – XOR 121](#_Toc368391702)

[XI – XOR Immediate 122](#_Toc368391703)

[XC – XC Character 123](#_Toc368391704)

[Others 128](#_Toc368391705)

[TM – Test Under Mask 129](#_Toc368391706)

[EX – Execute 130](#_Toc368391707)

[Assembler Directives 135](#_Toc368391708)

[DC - Define Constant 136](#_Toc368391709)

[DS - Define Storage 137](#_Toc368391710)

[EQU – Equate 138](#_Toc368391711)

[DSECT – Define Dummy Section 139](#_Toc368391712)

[ORG – Origin 140](#_Toc368391713)

[USING – Define Base Register 140](#_Toc368391714)

[DROP – Drop Base Register 140](#_Toc368391715)

[CSECT – Define Control Section 140](#_Toc368391716)

# 

# Introduction

## IBM S/390 Architecture

**CPU**

**Main Storage**

**Channel Subsystem**

**CPU**

**Serial Channel Path**

**Parallel Channel Path**

***IBM S/390 Architecture***

### Introduction

* The S/390 is a 32-bit architecture introduced by IBM in late 1990
* It supports 32-bit arithmetic computations whereas only 31-bit addressing
* It allows [backward compatibility](http://en.wikipedia.org/wiki/Backward_compatibility) with its predecessors S/360, S/370 etc. that supports 24-bit addressing

### Main Storage

* The main storage provides high speed access to data by CPU and Channel Subsystem
* Data and Programs that are currently being processed are stored in the main storage
* In a S/390 architecture, the storage is available in multiple blocks of size 4KB
* The main memory may also include a faster access buffer storage, called a cache

### CPU

* The **C**entral **P**rocessing **U**nit, often referred to as ‘brain’ of the computer, controls all the activities of the computer system
* It is responsible for instruction execution, interrupt action, timing function and other machine related functions
* S/390 Architecture supports more than one CPU

### Main Components of the CPU

**General Registers**

* The General registers are used to store addresses and data
* There are 16 registers, identified by numbers 0-15
* Each register is of 32 bits (4 bytes) in size
* For certain operations, two adjacent general registers are coupled, providing an 64-bit format (even-odd pair)

**PSW**

* The **P**rogram **S**tatus **W**ord contains information required for execution of the currently active program
* It is 64 bits in size and includes the instruction address, condition code, and other control fields

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0  8 | **PSW Key**  12 | 15 | **P** | 18 | **CC**  20 | 31 |

|  |  |
| --- | --- |
| **A**  32  33 | **Instruction Address**  63 |

***PSW Format***

* **PSW Key**
  + Bits 8-11 form the access key for storage references by the CPU
  + The PSW key is matched with a storage key when information is stored or fetched from a location that is protected
* **Problem State (P)**
  + Bit 15 states whether the CPU is in the problem state or supervisor state

Bit 15 = 0 🡪 Supervisor State

Bit 15 = 1 🡪 Problem State

* **Condition Code (CC)**
  + Bits 18-19 form the condition code
  + The condition code is set to 0, 1, 2, or 3, depending on the result obtained in executing certain instructions
* **Addressing Mode (A)**
  + Bit 32 specifies the addressing mode that is being used

Bit 32 = 0 🡪 24-bit addressing

Bit 32 = 1 🡪 31-bit addressing

* **Instruction Address**
  + Bits 33-63 form the instruction address
  + This address designates the location of the next instruction to be executed

### Channel Subsystem

* Input/output (I/O) operations involve the transfer of information between main storage and an I/O device
* The channel subsystem directs the flow of information between I/O devices and main storage, relieving the CPU of the task of communicating directly

## S/390 Assembly Language

### Instruction

* An instruction is a binary bit pattern which performs a specific task
* The instructions are 1, 2 or 3 half words in size and must be located in storage on a half word boundary
* The various type of instructions are General, Decimal, Floating Point and Control Instructions
* Each instruction consists of two major parts
  + Operation code that specifies the operation to be performed
  + Operands, on which the operation is performed. The Operands can be a Register(R), Base displacement address(BDDD), Indexed base displacement address(XDDD) or Immediate value(I).

### Instruction Formats

* The instructions used in application programming fall into one of the following categories:

**RR Format** 🡪 Register and Register operation

|  |  |  |
| --- | --- | --- |
| **Op Code** | **R1** | **R2** |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 8 | 12 | 15 |

Where,

R1 = Register 1

R2 = Register 2

**RX Format** 🡪 Register and Indexed Storage operation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Op Code** | **R1** | **X2** | **B2** | **D2** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 8 | 12 | 16 | 20 | 31 |

Where,

R1 = Register 1 – First operand

X2 = Index Register for second operand

B2 = Base Register for second operand

D2 = Displacement value for second operand

**RS Format** 🡪 Register and Storage operation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Op Code** | **R1** | **R3** | **B2** | **D2** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 8 | 12 | 16 | 20 | 31 |

Where,

R1 = Register 1 – First operand

R3 = Register / Mask Value

B2 = Base Register for second operand

D2 = Displacement value for second operand

**SI Format** 🡪 Storage and Immediate operation

|  |  |  |  |
| --- | --- | --- | --- |
| **Op Code** | **I1** | **B2** | **D2** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 8 | 16 | 20 | 31 |

Where,

I1 = Immediate Value

B2 = Base Register for second operand

D2 = Displacement value for second operand

**SS Format** 🡪 Storage and Storage operation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Op Code** | **L** | **B1** | **D1** | **B2** | **D2** |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 8 | 16 | 20 | 32 | 36 | 47 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Op Code** | **L1** | **L2** | **B1** | **D1** | **B2** | **D2** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 8 | 12 | 16 | 20 | 32 | 36 | 47 |

Where,

L, L1 = Length of first operand

L2 = Length of second operand

B1 = Base Register for first operand

D1 = Displacement value for first operand

B2 = Base Register for second operand

D2 = Displacement value for second operand

**R &I Format** 🡪 Register Immediate and Extended Op code

|  |  |  |  |
| --- | --- | --- | --- |
| **Op Code** | **R1** | **Ext** | **I2** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 8 | 12 | 16 |  | 31 |

Where,

R1 = Register – First Operand

Ext = Op-code Extension

I2 = Immediate value

### Storage Addressing

* In a S/390 Architecture, storage locations are addressed through the base and displacement method
* The Base is a general purpose register that contains the address of a storage location
* The Displacement is an offset from the Base that contains the required data
* The page size used is 4KB. Therefore the displacement should be in the range 0-4095
* The effective address is calculated by adding the displacement to the contents of base register
* Certain instructions allow optional usage of an Index register. The index register, if present, is included in the effective address calculation
* Register R0 cannot be used as a Base or Index register. If used, its contents are not included in the effective address calculation.

**Example**

**L R1, 10(R3, R2)**

Base Register R2 = 0C DB 10 00

Index Register R3 = 00 00 00 02

Displacement = 00 00 00 0A

Effective Address = 0C DB 10 0C

**R2**

**0C DB 10 0C**

**0C DB 1F FF**

**0C DB 10 00**

**+**

**Displacement**

**R3**

***Effective Address Calculation***

### Storage Operands

Two ways of representing storage operands are:

* + **Direct** - The storage location is explicitly represented in an instruction with a base, index register along with displacement
  + **Indirect –** The storage location is implicitly represented by means of a label which at assembly time is resolved into the **Direct** form

**Example:**

**ARRAY**

**0C DB 10 0C**

**0C DB 10 00**

**R2**

* + - Direct Addressing: **LA R1, 12(, R2)**
    - Indirect Addressing: **LA R1, ARRAY**

# Instructions

## Load and Store

### LA – Load Address

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **LA R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **41** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The effective address computed by the  **X2**,  **B2**, and  **D2** fields is loaded into register  **R1** | |
| **Condition Code Change:** No | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * Usage of R0 as base or index register will result in value zero being used in effective address computation instead of the contents of R0 * This instruction is handy to: * Load a constant within the range 0 – 4095 into a register * Increment a register by a constant * Add two registers and a constant | |
| **Instance:**  R1 = Irrelevant R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 0F 1A 20 AB 10 .. ..  ↓  **LA R1 , 10(R5,R2)**  ↓  R1 = 0C DB 10 0C R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 0F 1A 20 AB 10 .. .. | |

### LR – Load Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **LR R1 , R2** | |  |  |  | | --- | --- | --- | | **18** | **R1** | **R2** | |
| **Description:**  The four byte contents of register  **R2** is loaded into register  **R1** | |
| **Condition Code Change**: No | |
| **Exception:** None | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R4 = Irrelevant R5 = 0C DB 10 00  ↓  **LR R4 , R5**  ↓  R4 = 0C DB 10 00 R5 = 0C DB 10 00 | |

### L – Load

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **L R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **58** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The **four byte contents** stored in the effective address specified by the  **X2**,  **B2**, and  **D2** fields is loaded into register  **R1** | |
| **Condition Code Change**: No | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:**   * The second operand needs to be full word boundary aligned | |
| **Instance:**  R1 = Irrelevant R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 0F 1A 20 AB 10 .. ..  ↓  **L R1 , 10(R5,R2)**  ↓  R1 = 00 0F 1A 20 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 0F 1A 20 AB 10 .. .. | |

### ST – Store

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ST R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **50** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The **four byte contents** of register  **R1** is stored in the storage location specified by the  **X2**,  **B2**, and  **D2** fields | |
| **Condition Code Change**: No | |
| **Exception:** Access – Store 2nd Operand | |
| **Programming Considerations & Tips:**   * The second operand needs to be full word boundary aligned | |
| **Instance:**  R1 = 00 0F 1A 20 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 7B C3 82 E1 AB 10 .. ..  ↓  **ST R1 , 10(R5,R2)**  ↓  R1 = 00 0F 1A 20 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 0F 1A 20 AB 10 .. .. | |

### LH – Load Halfword

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **LH R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **48** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The **two bytes contents** stored in the effective address specified by the  **X2**,  **B2**, and  **D2** fields is **sign extended** and loaded into the register  **R1** as a four byte value | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:**   * The second operand needs to be half word boundary aligned | |
| **Instance:**  R1 = 0B 01 F1 05 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 0F 1A 20 AB 10 .. ..  ↓  **LH R1 , 10(R5,R2)**  ↓  R1 = 00 00 00 0F R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 0F 1A 20 AB 10 .. ..    R1 = 0B 01 F1 05 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 D0 0F 1A 20 AB 10 .. ..  ↓  **LH R1 , 10(R5,R2)**  ↓  R1 = FF FF D0 0F R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 D0 0F 1A 20 AB 10 .. ..    **Note:** Sign extended bits are underlined | |

### STH – Store Halfword

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **STH R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **40** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The **rightmost** **two bytes** of register  **R1**  is stored in the storage location specified by the  **X2**,  **B2**, and  **D2** fields | |
| **Condition Code Change:** No | |
| **Exception:** Access – Store 2nd Operand | |
| **Programming Considerations & Tips:**   * The second operand needs to be half word boundary aligned | |
| **Instance:**  R1 = 0B 01 F1 05 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 10 0F 1A 20 AB 10 .. ..  ↓  **STH R1 , 10(R5,R2)**  ↓  R1 = 0B 01 F1 05 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 F1 05 1A 20 AB 10 .. .. | |

### LHI – Load Halfword Immediate

**R & I**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **LHI R1 , I2** | |  |  |  |  | | --- | --- | --- | --- | | **A7** | **R1** | **8** | **I2** | |
| **Description:**  The Halfword immediate operand provided in  **I2** is loaded into the register **R1** as a four byte value | |
| **Condition Code Change:** No | |
| **Exception:** None | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R1 = 0B 01 F1 05  ↓  **LHI R1 , -10**  ↓  R1 = FF FF FF F6 | |

### IC – Insert Character

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **IC R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **43** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The **one byte content** stored in the effective address specified by the  **X2**,  **B2**, and  **D2** fields is **inserted into rightmost byte** of register **R1 ,** leaving the **foremost three bytes unaltered** | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R1 = 0B 01 F1 05 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 D4 0F 1A 20 AB 10 .. ..  ↓  **IC R1 , 10(R5,R2)**  ↓  R1 = 0B 01 F1 D4 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 D4 0F 1A 20 AB 10 .. .. | |

### STC – Store Character

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **STC R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **42** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The **rightmost byte** of register **R1** is stored in the storage location specified by the  **X2**,  **B2**, and  **D2** fields | |
| Condition Code Change:  **No** | |
| **Exception:** Access – Store 2nd Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R1 = 0B 01 F1 05 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 D4 0F 1A 20 AB 10 .. ..  ↓  **STC R1 , 10(R5,R2)**  ↓  R1 = 0B 01 F1 05 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 05 0F 1A 20 AB 10 .. .. | |

### ICM – Insert Character under Mask

**RS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ICM R1, M3, D2 (B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **BF** | **R1** | **M3** | **B2** | **D2** | |
| **Description:**  The contents stored in the effective address specified by **B2** and  **D2** fields are inserted into register  **R1**, based on the four bit mask stated in **M3**  Bits from left to right of mask **M3** corresponds to individual bytes of register **R1**. The byte positions in register **R1** corresponding to the ones in mask **M3** are filled with successive bytes from storage location specified by  **B2** and  **D2** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | All inserted bytes are zeroes (or) M3 = 0 | | 1 | If the leftmost bit of the storage operand is 1 | | 2 | If the leftmost bit of the storage operand is 0 | | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:**   * This instruction is handy to load a four byte content into a register from a non-full word aligned storage location unlike Load instruction | |
| **Instance:**  R1 = 0B 01 F1 05 R2 = 0C DB 10 00  CC =Irrelevant [0C DB 10 0C] 🡪 D4 0F 1A 20 AB 10 .. ..  ↓  **ICM R1 ,B’1011’, 12(R2)**  ↓  R1 = D4 01 0F 1A R2 = 0C DB 10 00  CC = 1 [0C DB 10 0C] 🡪 D4 0F 1A 20 AB 10 .. ..  **Note:** The above instruction can also be coded as **ICM R1 ,X’B’, 12(R2)** | |

### STCM – Store Character under Mask

**RS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **STCM R1, M3, D2 (B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **BE** | **R1** | **M3** | **B2** | **D2** | |
| **Description:**  The contents of register  **R1** is stored in the storage location specified by **B2** and  **D2** fields, based on the four bit mask stated in **M3** | |
| **Condition Code Change:** No | |
| **Exception:** Access – Store 2nd Operand | |
| **Programming Considerations & Tips:**   * This instruction is handy to store a four byte content of a register into a non-full word aligned storage location unlike Store instruction | |
| **Instance:**  R1 = D4 01 0F 1A R2 = 0C DB 10 00  [0C DB 10 0C] 🡪 AD 8E C2 61 AB 10 .. ..  ↓  **STCM R1 ,B’1010’, 12(R2)**  ↓  R1 = D4 01 0F 1A R2 = 0C DB 10 00  [0C DB 10 0C] 🡪 D4 0F C2 61 AB 10 .. ..  **Note:** The above instruction can also be coded as **STCM R1 ,X’A’, 12(R2)** | |

### LM – Load Multiple

**RS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **LM R1, R3, D2 (B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **98** | **R1** | **R3** | **B2** | **D2** | |
| **Description:**  The **consecutive four byte contents** stored in the effective address specified by the **B2** and  **D2** fields are loaded intoregisters ranging from **R1** up to  **R3** respectively | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:**   * To load registers R14, R15, R0 and R1, instruction LM R14, R1, 10(R2) can be used | |
| **Instance:**  R2 = 0C DB 10 00 R4, R5, R6, R7 = Irrelevant  [0C DB 10 0C] 🡪 D4 0F 1A 20 AB 10 00 0F 00 00 AB DE 84 2C 7D 12 4E 21 ..  ↓  **LM R4 ,R7, 12(R2)**  ↓  R2 = 0C DB 10 00  R4 = D4 0F 1A 20 R5 = AB 10 00 0F  R6 = 00 00 AB DE R7 = 85 2C 7D 12  [0C DB 10 0C] 🡪 D4 0F 1A 20 AB 10 00 0F 00 00 AB DE 84 2C 7D 12 4E 21 .. | |

### STM – Store Multiple

**RS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **STM R1, R3, D2 (B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **90** | **R1** | **R3** | **B2** | **D2** | |
| **Description:**  The contents in registers ranging from **R1** up to  **R3** are consecutively stored inthe effective address specified by the **B2** and  **D2** fields | |
| **Condition Code Change:** No | |
| **Exception:** Access – Store 2nd Operand | |
| **Programming Considerations & Tips:**   * To store registers R14, R15, R0 and R1, instruction STM R14, R1, 10(R2) can be used | |
| **Instance:**  R2 = 0C DB 10 00  R14 = D4 0F 1A 20 R15 = AB 10 00 0F  R0 = 00 00 AB DE R1 = 85 2C 7D 12  [0C DB 10 0C] 🡪 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0F 7D E2 F6 ..  ↓  **STM R14 ,R1, 12(R2)**  ↓  R2 = 0C DB 10 00  R14 = D4 0F 1A 20 R15 = AB 10 00 0F  R0 = 00 00 AB DE R1 = 85 2C 7D 12  [0C DB 10 0C] 🡪 D4 0F 1A 20 AB 10 00 0F 00 00 AB DE 84 2C 7D 12 E2 F6 .. | |

### LTR – Load and Test Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **LTR R1 , R2** | |  |  |  | | --- | --- | --- | | **12** | **R1** | **R2** | |
| **Description:**  This instruction performs the same function as LR instruction, additionally, the copied content is tested for its sign and the condition code is set accordingly | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Zero | | 1 | Negative | | 2 | Positive | | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * This instruction is handy to test if the contents of a register is zero, positive or negative | |
| **Instance:**  R4 = Irrelevant R5 = 0C DB 10 00 CC = Irrelevant  ↓  **LTR R4 , R5**  ↓  R4 = 0C DB 10 00 R5 = 0C DB 10 00 CC = 2 | |

## 

## Arithmetic

### AR – Add Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **AR R1 , R2** | |  |  |  | | --- | --- | --- | | **1A** | **R1** | **R2** | |
| **Description:**  The contents of register **R1** and **R2** are added and the result placed in register **R1** leaving  **R2** unchanged | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** Fixed-point overflow | |
| **Programming Considerations & Tips:**   * The operands and the sum are treated as 32-bit signed numbers | |
| **Instance:**  R4 = 00 A2 30 09 R5 = 00 00 0A B7 CC = Irrelevant  ↓  **AR R4 , R5**  ↓  R4 = 00 A2 3A C0 R5 = 00 00 0A B7 CC = 2 | |

# 

### A – Add

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **A R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **5A** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The content of register  **R1** is added to the four byte contents stored in the effective address specified by the  **X2**,  **B2**, and  **D2** fields and the result is stored in register **R1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** Access – Fetch 2nd Operand, Fixed-point overflow | |
| **Programming Considerations & Tips:**   * The operands and the sum are treated as 32-bit signed numbers | |
| **Instance:**  R7 = 00 00 00 0A R2 = 0C DB 10 00 CC = Irrelevant  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 FF FF FF F3 AB 10 .. ..  ↓  **A R7 , 10(R5,R2)**  ↓  R7 = FF FF FF FD R2 = 0C DB 10 00 CC = 1  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 FF FF FF F3 AB 10 .. ..  **Note:** 2’s complement of -13 is FF FF FF F3 and -3 is FF FF FF FD | |

### AH – Add Halfword

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **AH R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **4A** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The content of register  **R1** is added to the two byte contents stored in the effective address specified by the  **X2**,  **B2**, and  **D2** fields and the result is stored in register **R1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** Access – Fetch 2nd Operand, Fixed-point overflow | |
| **Programming Considerations & Tips:**   * The operands and the sum are treated as signed numbers | |
| **Instance:**  R7 = 00 00 00 0A R2 = 0C DB 10 00 CC = Irrelevant  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 FF F6 01 13 AB 10 .. ..  ↓  **AH R7 , 10(R5,R2)**  ↓  R7 = 00 00 00 00 R2 = 0C DB 10 00 CC = 0  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 FF F6 01 13 AB 10 .. ..  **Note:** 16-bit representation of -10 in2’s complement form is FF F6 | |

### AHI – Add Halfword Immediate

**R & I**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **AHI R1 , I2** | |  |  |  |  | | --- | --- | --- | --- | | **A7** | **R1** | **A** | **I2** | |
| **Description:**  The Halfword immediate operand provided in  **I2** is added to the register **R1** and the result is stored in **R1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** None | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R1 = 0B 01 F1 15 CC = Irrelevant  ↓  **AHI R1 , -16**  ↓  R1 = 0B 01 F1 05 CC = 2 | |

### 

### SR – Subtract Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SR R1 , R2** | |  |  |  | | --- | --- | --- | | **1B** | **R1** | **R2** | |
| **Description:**  The contents of register **R2** is subtracted from **R1** and the result is placed in register **R1** leaving  **R2** unchanged | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** Fixed-point overflow | |
| **Programming Considerations & Tips:**   * The operands and the difference are treated as 32-bit signed numbers | |
| **Instance:**  R4 = 00 A2 30 09 R5 = 00 00 0A B7 CC = Irrelevant  ↓  **SR R4 , R5**  ↓  R4 = 00 A2 25 52 R5 = 00 00 0A B7 CC = 2 | |

# 

### S – Subtract

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **5B** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The four byte contents stored in the effective address specified by the  **X2**,  **B2**, and  **D2** fields is subtracted from register **R1** and the result is stored in **R1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** Access – Fetch 2nd Operand, Fixed-point overflow | |
| **Programming Considerations & Tips:**   * The operands and the difference are treated as 32-bit signed numbers | |
| **Instance:**  R7 = 00 00 00 2A R2 = 0C DB 10 00 CC = Irrelevant  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 00 00 2B AB 10 .. ..  ↓  **S R7 , 10(R5,R2)**  ↓  R7 = FF FF FF FF R2 = 0C DB 10 00 CC = 1  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 00 00 2B AB 10 .. ..  **Note:** 2’s complement of -1 is FF FF FF FF | |

### SH – Subtract Halfword

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SH R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **4B** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The two byte contents stored in the effective address specified by the  **X2**,  **B2** and  **D2** fields is subtracted from register  **R1** and the result is stored in **R1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** Access – Fetch 2nd Operand, Fixed-point overflow | |
| **Programming Considerations & Tips:**   * The operands and the sum are treated as signed numbers | |
| **Instance:**  R7 = FF FF FF FA R2 = 0C DB 10 00 CC = Irrelevant  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 FF FA 01 13 AB 10 .. ..  ↓  **SH R7 , 10(R5,R2)**  ↓  R7 = 00 00 00 00 R2 = 0C DB 10 00 CC = 0  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 FF FA 01 13 AB 10 .. .. | |

### MR – Multiply Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MR RP1 , R2** | |  |  |  | | --- | --- | --- | | **1C** | **RP1** | **R2** | |
| **Description:**  It multiplies   * **32-bit multiplicand** supplied in the **odd register** of register-pair **RP1** and * **32-bit multiplier** supplied in register  **R2**   andthe **64-bit product** is stored in  **RP1**  **RP1** refers to an even-odd pair register | |
| **Condition Code Change:** No | |
| **Exception:** Specification | |
| **Programming Considerations & Tips:**   * The operands and the product are treated as signed numbers | |
| **Instance:**  R0 = Irrelevant R1 = 00 00 00 0C R5 = 00 00 00 05  ↓  **MR R0 , R5**  ↓  R0 = 00 00 00 00 R1 = 00 00 00 3C R5 = 00 00 00 05 | |

# 

### M – Multiply

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **M RP1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **5C** | **RP1** | **X2** | **B2** | **D2** | |
| **Description:**  It multiplies   * **32-bit multiplicand** supplied in the **odd register** of register-pair **RP1** and * **32-bit multiplier** supplied in storage location specified by the  **X2**,  **B2**, and  **D2** fields   andthe **64-bit product** is stored in  **RP1** | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand, Specification | |
| **Programming Considerations & Tips:**   * The operands and the product are treated as signed numbers | |
| **Instance:**  R14 = Irrelevant R15 = 00 00 00 2A  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 FF FF FF FF AB 10 .. ..  ↓  **M R14 , 12(,R2)**  ↓  R14 = FF FF FF FF R15 = FF FF FF D6  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 FF FF FF FF AB 10 .. .. | |

### MH – Multiply Halfword

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **MH R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **4C** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  It multiplies   * **32-bit multiplicand** supplied in the register **R1** and * **16-bit multiplier** supplied in storage location specified by the  **X2**,  **B2**, and  **D2** fields   andthe **32-bit product** is stored in register **R1** | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:**   * The operands and the product are treated as signed numbers | |
| **Instance:**  R7 = 00 00 8B 2A R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 03 01 13 AB 10 .. ..  ↓  **MH R7 , 10(R5,R2)**  ↓  R7 = 00 01 A1 7E R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 03 01 13 AB 10 .. .. | |

### MHI – Multiply Halfword Immediate

**R & I**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **MHI R1 , I2** | |  |  |  |  | | --- | --- | --- | --- | | **A7** | **R1** | **C** | **I2** | |
| **Description:**  The Halfword immediate operand provided in  **I2** is multiplied to the register **R1** and the result is stored in **R1** | |
| **Condition Code Change:** No | |
| **Exception:** None | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R1 = 00 00 00 15  ↓  **MHI R1 , 10**  ↓  R1 = 00 00 00 96 | |

### DR – Divide Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **DR RP1 , R2** | |  |  |  | | --- | --- | --- | | **1D** | **RP1** | **R2** | |
| **Description:**  It divides   * **64-bit dividend** supplied in the register-pair **RP1** and * **32-bit divisor** supplied in register  **R2**   andstores   * **Remainder** in **even register** of  **RP1** * **Quotient** in **odd register** of  **RP1**   **RP1** refers to an even-odd pair register | |
| **Condition Code Change:** No | |
| **Exception:** Specification, Divide by Zero | |
| **Programming Considerations & Tips:**   * The operands and the results are treated as signed numbers | |
| **Instance:**  R0 = 00 00 00 00 R1 = 00 00 08 2F R5 = 00 00 00 0A  ↓  **DR R0 , R5**  ↓  R0 = 00 00 00 05 R1 = 00 00 00 D1 R5 = 00 00 00 0A | |

# 

### D – Divide

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **D RP1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **5D** | **RP1** | **X2** | **B2** | **D2** | |
| **Description:**  It divides   * **64-bit dividend** supplied in the register-pair **RP1** and * **32-bit divisor** supplied in storage location specified by the  **X2**,  **B2**, and  **D2** fields   andstores   * **Remainder** in **even register** of  **RP1** * **Quotient** in **odd register** of  **RP1** | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand, Specification, Divide by Zero | |
| **Programming Considerations & Tips:**   * The operands and the results are treated as signed numbers | |
| **Instance:**  R14 = 00 00 00 00 R15 = 00 00 00 2A  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 00 00 00 02 AB 10 .. ..  ↓  **D R14 , 12(,R2)**  ↓  R14 = 00 00 00 00 R15 = 00 00 00 15  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 00 00 00 02 AB 10 .. .. | |

## Compare

### CR – Compare Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CR R1 , R2** | |  |  |  | | --- | --- | --- | | **19** | **R1** | **R2** | |
| **Description:**  The contents of register **R1** and **R2** are compared and the condition code is set accordingly leaving the operands unchanged | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Operand1  = Operand2 | | 1 | Operand1  < Operand2 | | 2 | Operand1  > Operand2 | | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * The operands are treated as 32-bit signed numbers | |
| **Instance:**  R4 = 00 A2 30 09 R5 = 00 00 0A B7 CC = Irrelevant  ↓  **CR R4 , R5**  ↓  R4 = 00 A2 30 09 R5 = 00 00 0A B7 CC = 2 | |

### C – Compare

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **C R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **59** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The contents of register **R1** and **four byte contents** stored in the effective address specified by the **X2**, **B2**, and **D2** are compared and the condition code is set accordingly leaving the operands unchanged | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Operand1  = Operand2 | | 1 | Operand1  < Operand2 | | 2 | Operand1  > Operand2 | | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:**   * The operands are treated as 32-bit signed numbers | |
| **Instance:**  R7 = FF FF FF FF R2 = 0C DB 10 00 CC = Irrelevant  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 00 00 00 AB 10 .. ..  ↓  **C R7 , 10(R5,R2)**  ↓  R7 = FF FF FF FF R2 = 0C DB 10 00 CC = 1  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 00 00 00 AB 10 .. .. | |

### CH – Compare Halfword

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CH R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **49** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The contents of register **R1** and **two byte contents** stored in the effective address specified by the **X2**, **B2**, and **D2** are compared and the condition code is set accordingly leaving the operands unchanged | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Operand1  = Operand2 | | 1 | Operand1  < Operand2 | | 2 | Operand1  > Operand2 | | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:**   * The operands are treated as signed numbers irrespective of difference in size | |
| **Instance:**  R7 = FF FF FF FE R2 = 0C DB 10 00 CC = Irrelevant  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 FF FE 00 00 AB 10 .. ..  ↓  **CH R7 , 10(R5,R2)**  ↓  R7 = FF FF FF FE R2 = 0C DB 10 00 CC =0  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 FF FE 00 00 AB 10 .. .. | |

### CHI – Compare Halfword Immediate

**R & I**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **CHI R1 , I2** | |  |  |  |  | | --- | --- | --- | --- | | **A7** | **R1** | **E** | **I2** | |
| **Description:**  The Halfword immediate operand provided in  **I2** is compared against the register **R1** and the condition code is set accordingly | |
| Condition Code Change:  **Yes**   |  |  | | --- | --- | | 0 | Operand1  = Operand2 | | 1 | Operand1  < Operand2 | | 2 | Operand1  > Operand2 | | |
| **Exception:** None | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R1 = FF FF FF FF CC = Irrelevant  ↓  **CHI R1 , 2**  ↓  R1 = FF FF FF FF CC = 1 | |

## 

## 

## Branch

### BCR – Branch on Condition Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BCR M1 , R2** | |  |  |  | | --- | --- | --- | | **07** | **M1** | **R2** | |
| **Description:**  It branches to location specified in **R2** when the condition code satisfies the four bit mask specified in  **M1** and if not, continues with the next sequential instruction  The four possible condition codes 0, 1, 2 and 3 corresponds to four bits (left to right) of mask | |
| **Condition Code Change:** No | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * Using R0 will not branch even when the condition code matches. * Mask ‘1111’ will branch unconditionally irrespective of condition code. | |
| **Instance:**  **BCR B’1011’, R14**  ↓  The branch occurs to address specified in R14 when CC holds a value of 0, 2 or 3 and continues with next instruction when CC is 1 | |

### BC – Branch on Condition

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **BC M1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **47** | **M1** | **X2** | **B2** | **D2** | |
| **Description:**  It branches to the address location specified by **X2**, **B2**, and **D2** fields when the condition code satisfies the four bit mask specified in **M1** or continues with the next sequential instruction if not  The four possible condition codes 0, 1, 2 and 3 corresponds to four bits (left to right) of mask | |
| **Condition Code Change:** No | |
| **Exception:** None | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  **BC B’1001’, NEXTSEG**  ↓  It branches to location NEXTSEG when CC holds a value of 0 or 3 and in case of 1 or 2 it continues with executing the following instruction | |

### Other Branch on Condition/Register Instructions

|  |  |  |  |
| --- | --- | --- | --- |
| **Performs** | **Instruction** | | **Implied Mask** |
| **Location** | **Register** |
| Branch unconditionally | **B** | **BR** | B’1111’, X’F’ |
| Branch on equal | **BE** | **BER** | B’1000’, X’8’ |
| Branch on high | **BH** | **BHR** | B’0010’, X’2’ |
| Branch on low | **BL** | **BLR** | B’0100’, X’4’ |
| Branch on not equal | **BNE** | **BNER** | B’0111’, X’7’ |
| Branch on not high | **BNH** | **BNHR** | B’1101’, X’D’ |
| Branch on not low | **BNL** | **BNLR** | B’1011’, X’B’ |
| Branch on overflow  Branch on one | **BO** | **BOR** | B’0001’, X’1’ |
| Branch on zero | **BZ** | **BZR** | B’1000’, X’8’ |
| Branch on plus | **BP** | **BPR** | B’0010’, X’2’ |
| Branch on minus  Branch on minus | **BM** | **BMR** | B’0100’, X’4’ |
| Branch on not zero | **BNZ** | **BNZR** | B’0111’, X’7’ |
| Branch on not plus | **BNP** | **BNPR** | B’1101’, X’D’ |
| Branch on not minus  Branch on not mixed | **BNM** | **BNMR** | B’1011’, X’B’ |
| Branch on no overflow  Branch on not one | **BNO** | **BNOR** | B’1110’, X’E’ |

### BCTR – Branch on Count Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BCTR R1 , R2** | |  |  |  | | --- | --- | --- | | **06** | **R1** | **R2** | |
| **Description:**  It branches to location specified in **R2** when contents of **R1** decremented by 1 results in a non-zero value | |
| **Condition Code Change:** No | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * Using R0 in place of **R2** will result in decrementing the contents of **R1** by 1 but no branching occurs. * It is commonly used in constructing a loop, but need to be cautious to not enter the loop structure unintentionally with a zero or negative value in **R1** | |
| **Instance:**  R2 = FF FF FF FF  ↓  **BCTR R2, R7**  ↓  R2 = FF FF FF FE  The execution branches to the address specified in R7 after decrementing R2 contents by 1 | |

### BCT – Branch on Count

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **BCT R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **46** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  It branches to the address location specified by **X2**, **B2**, and **D2** fields when contents of **R1** decremented by 1 results in a non-zero value | |
| **Condition Code Change:** No | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * Need to be cautious to not enter the loop structure built using BCT with a zero or negative value in **R1** | |
| **Instance:**  R2 = 00 00 00 0A  ↓  **BCT R2, PAXLOOP**  ↓  R2 = 00 00 00 09  The control branches to the label PAXLOOP after decrementing R2 by 1 | |

### BASR – Branch and Save Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BASR R1 , R2** | |  |  |  | | --- | --- | --- | | **0D** | **R1** | **R2** | |
| **Description:**  It saves the address of the next sequential instruction into Register **R1** and branches to location specified in **R2** | |
| **Condition Code Change:** No | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * Primarily used as to call a subroutine * Using R0 as **R2** will result in no branch | |
| **Instance:**  **BASR R7, R2**  ↓  The branch occurs to address specified in R2 after saving the address of NSI (from PSW) in R7 | |

### BAS – Branch and Save

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **BAS R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **4D** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  It saves the address of the next sequential instruction into Register **R1** and branches to the address location specified by **X2**, **B2**, and **D2** fields | |
| **Condition Code Change:** No | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * Primarily used as to call a subroutine | |
| **Instance:**  **BAS R7, ROUTINE**  ↓  It branches to location **ROUTINE** after saving the address of NSI in R7 | |

## Character: Move, Compare

### MVI – Move Immediate

**SI**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **MVI D1 ( B1), I2** | |  |  |  |  | | --- | --- | --- | --- | | **92** | **I2** | **B1** | **D1** | |
| **Description:**  It moves the immediate value **I2** to the address location specified by **B1** and **D1** | |
| **Condition Code Change:** No | |
| **Exception:** Access – Store 1st Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 24 13 1F 7E C6 .. ↓  **MVI 12(R2),C’A’**  ↓  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 C1 13 1F 7E C6 .. | |

### MVC – Move Character

**SS**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **MVC D1 (L, B1), D2(B2)** | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **FA** | **L** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  It moves **L** number of bytes from address location specified by **B2**, **D2** to **B1**, **D1** | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand | |
| **Programming Considerations & Tips:**   * As **L** is 8 bits in size, the maximum number of bytes that can be moved using this instruction is restricted to 256 * This instruction allows destructive overlapping, i.e. the source and destination addresses can overlap | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00  [0C DC 04 08] 🡪 10 00 00 0F 0A BC ..  [0C DB 10 0C] 🡪 12 30 1F 7E C6 E2 ..  ↓  **MVC 0(5,R7) , 12(R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00  [0C DC 04 08] 🡪 12 30 1F 7E C6 BC ..  [0C DB 10 0C] 🡪 12 30 1F 7E C6 E2 ..    Destructive overlapping:    R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 40 30 1F 7E C6 E2 D4 A7 F5 89 91 .. ↓  **MVC 13(7,R2) , 12(R2)**  ↓  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 40 40 40 40 40 40 40 40 F5 89 91 .. | |

### MVCL – Move Character Long

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MVCL RP1,RP2** | |  |  |  | | --- | --- | --- | | **0E** | **RP1** | **RP2** | |
| **Description:**  MVCL is generally used to overcome the limitation of MVC instruction and can be used to move data up to **16MB**. The description of the operands are as follows:  **RP1** (Even) : Destination Address  **RP1** (Odd): Bits 8-31 contains the length of data to be moved(destination length);  Bits 0-7 are ignored  **RP2** (Even): Source Address  **RP2** (Odd): Bits 8-31 contains the length of the source;  Bits 0-7 contains the padding character which is used when length of source is lesser than the destination length | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Operands are of equal length; data copied | | 1 | Destination length is shorter; data copied but truncated | | 2 | Destination length is longer; data copied and padded | | 3 | Destructive overlap; data not copied | | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand, Specification | |
| **Programming Considerations & Tips:**   * Instruction does not support destructive overlapping | |
| **Instance:**  R2 = 0C DC 04 08 R3 = 00 00 00 08 CC = Irrelevant  R6 = 0C DB 10 0C R7 = 40 00 00 05  [0C DC 04 08] 🡪 99 99 99 99 99 99 99 99 99 99 .. [0C DB 10 0C] 🡪 C1 C7 C9 C1 C6 ..  ↓  **MVCL R2,R6**  ↓  R2, R3, R6, R7 = Contents may have changed CC = 2  [0C DC 04 08] 🡪 C1 C7 C9 C1 C6 40 40 40 99 99 .. [0C DB 10 0C] 🡪 C1 C7 C9 C1 C6 .. | |

### 

### CLI – Compare Logical Immediate

**SI**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **CLI D1 ( B1), I2** | |  |  |  |  | | --- | --- | --- | --- | | **95** | **I2** | **B1** | **D1** | |
| **Description:**  It compares the immediate value **I2** to the one byte data present in the address location specified by **B1** and **D1**  and sets the condition code accordingly | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Operand1  = Operand2 | | 1 | Operand1  < Operand2 | | 2 | Operand1  > Operand2 | | |
| **Exception:** Access – Fetch 1st Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 FF 13 1F .. CC = Irrelevant ↓  **CLI 12(R2),X’0B’**  ↓  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 FF 13 1F .. CC = 2 | |

### CLC – Compare Logical Character

**SS**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **CLC D1 (L, B1), D2(B2)** | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **D5** | **L** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  It compares **L** number of bytes from address location specified by **B2**, **D2** with those bytes specified by **B1**, **D1** and sets the condition code accordingly leaving the operands unchanged | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Operand1  = Operand2 | | 1 | Operand1  < Operand2 | | 2 | Operand1  > Operand2 | | |
| **Exception:** Access – Fetch 1st Operand, Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:**   * As **L** is 8 bits in size, the maximum number of bytes that can be compared using this instruction is restricted to 256 | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 10 00 00 0F 0A BC ..  [0C DB 10 0C] 🡪 12 30 1F 7E C6 E2 ..  ↓  **CLC 0(5,R7) , 12(R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = 1  [0C DC 04 08] 🡪 10 00 00 0F 0A BC ..  [0C DB 10 0C] 🡪 12 30 1F 7E C6 E2 .. | |

### CLCL – Compare Logical Character Long

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLCL RP1,RP2** | |  |  |  | | --- | --- | --- | | **0F** | **RP1** | **RP2** | |
| **Description:**  CLCL is generally used to overcome the limitation of CLC instruction and can be used to compare data up to **16MB**. The description of the operands are as follows:  **RP1** (Even) : Operand**1** Address  **RP1** (Odd): Bits 8-31 contains the length of Operand**1**  Bits 0-7 are ignored  **RP2** (Even): Operand**2** Address  **RP2** (Odd): Bits 8-31 contains the length Operand**2**;  Bits 0-7 contains the padding character to pad the **shorter** operand | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Operand1  = Operand2 | | 1 | Operand1  < Operand2 | | 2 | Operand1  > Operand2 | | |
| **Exception:** Access – Fetch 1st Operand, Access – Fetch 2nd Operand, Specification | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R2 = 0C DC 04 08 R3 = 00 00 00 08 CC = Irrelevant  R6 = 0C DB 10 0C R7 = 40 00 00 05  [0C DC 04 08] 🡪 C1 C7 C9 C1 C6 40 40 40 99 99 .. [0C DB 10 0C] 🡪 C1 C7 C9 C1 C6 98 98 ..  ↓  **CLCL R2,R6**  ↓  R2, R3, R6, R7 = Contents may have changed CC = 0  [0C DC 04 08] 🡪 C1 C7 C9 C1 C6 40 40 40 99 99 .. [0C DB 10 0C] 🡪 C1 C7 C9 C1 C6 98 98 .. | |

## Decimal - Packed

### Packed Decimal Number

A Packed Decimal number can be a maximum of 16-byte long in which each nibble holds a decimal value and the right most nibble holds the sign of the number.

The Digit Code can range from 0-9 and the Sign Code can range from A-F in which B & D are used to represent negative numbers.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Digit Code | Digit Code | … | Digit Code | Digit Code | Sign Code |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 4 | 8 … | 116 | 120 | 124 | 127 |

Packed numbers are byte aligned and operations like pad/truncate are performed from left to right.

### 

**Examples:**

**Positive Packed Decimal Numbers**

* 05 3C + 53
* 00 04 0A + 40
* 10 03 2F + 10032

**Negative Packed Decimal Numbers**

* 05 3D - 53
* 00 04 0B - 40
* 10 03 2B - 10032

### CVB – Convert to Binary

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CVB R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **4F** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  It converts the 8-byte packed decimal number in address location specified by **X2**, **B2**, and **D2** fields into a 32-bit signed number and stores in Register  **R1** | |
| **Condition Code Change:** No | |
| **Exception:** Data, Specification, Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:**   * The second operand needs to be double word boundary aligned | |
| **Instance:**  R7 = 00 00 00 00 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 00 01 23 45 67 89 0A 12 ..  ↓  **CVB R7 , 10(R5,R2)**  ↓  R7 = 49 96 02 D2 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 00 01 23 45 67 89 0A 12 .. | |

### CVD – Convert to Decimal

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CVD R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **4E** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  It converts the 32-bit signed number in Register **R1** into an 8-byte packed decimal number and stores at address location specified by **X2**, **B2**, and **D2** fields | |
| **Condition Code Change:** No | |
| **Exception:** Specification, Access – Store 2nd Operand | |
| **Programming Considerations & Tips:**   * The second operand needs to be double word boundary aligned | |
| **Instance:**  R7 = 00 00 04 D2 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 00 01 23 45 67 89 0A 12 .. ↓  **CVD R7 , 10(R5,R2)**  ↓  R7 = 00 00 04 D2 R2 = 0C DB 10 00  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 00 00 00 00 00 01 23 4F 12 .. | |

### AP – Add Packed

**SS**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **AP D1 ( L1, B1), D2 ( L2, B2)** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **FA** | **L1** | **L2** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  It adds two packed decimal numbers provided at address locations specified by **X1**, **B1**, **D1** and **X2**, **B2**, **D2** respectively and the result replaces Operand**1**. The length of the operands may vary and is so specified individually as **L1** and **L2** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand, Decimal Overflow, Data | |
| **Programming Considerations & Tips:**   * If L1 is too short to accommodate the result, decimal overflow occurs and the CC 3 is set while the result is truncated | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 09 99 99 9F 0A BC ..  [0C DB 10 0C] 🡪 00 00 1F 7E C6 ..  ↓  **AP 0(4,R7) , 12(3,R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = 2  [0C DC 04 08] 🡪 10 00 00 0F 0A BC ..  [0C DB 10 0C] 🡪 00 00 1F 7E C6 .. | |

### SP – Subtract Packed

**SS**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **SP D1 ( L1, B1), D2 ( L2, B2)** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **FB** | **L1** | **L2** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  It subtracts the packed decimal number provided at address locations specified by **X2**, **B2**, **D2** from that specified at **X1**, **B1**, **D1** and the difference replaces Operand**1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand, Decimal Overflow, Data | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 09 99 99 9F 0A BC ..  [0C DB 10 0C] 🡪 00 00 1F 7E C6 ..  ↓  **SP 0(4,R7) , 12(3,R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = 2  [0C DC 04 08] 🡪 09 99 99 8F 0A BC ..  [0C DB 10 0C] 🡪 00 00 1F 7E C6 .. | |

### MP – Multiply Decimal

**SS**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **MP D1 ( L1, B1), D2 ( L2, B2)** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **FC** | **L1** | **L2** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  It multiplies two packed decimal numbers provided at address locations specified by **X1**, **B1**, **D1** and **X2**, **B2**, **D2** respectively and the result replaces Operand**1**. The length of the operands may vary and is so specified individually as **L1** and **L2** | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand, Decimal Overflow, Data  exception | |
| **Programming Considerations & Tips:**  To ensure that the product will fit into the L1 bytes of the first operand field, the MP instruction restricts the operands as follows:  L2 <= 8 and L2 < L1 (Violation results in specification exception)  L1 must contain at least L2 bytes of leading zeros (Violation results in data exception). | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 00 00 12 3C 0A BC ..  [0C DB 10 0C] 🡪 00 00 08 6C C6 ..  ↓  **MP 0(4,R7) , 12(3,R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 00 10 57 8C 0A BC ..  [0C DB 10 0C] 🡪 00 00 08 6C C6 .. | |

### DP – Divide Decimal

**SS**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DP D1 ( L1, B1), D2 ( L2, B2)** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **FD** | **L1** | **L2** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  It divides the packed decimal number provided at address locations specified by **X2**, **B2**, **D2** from that specified at **X1**, **B1**, **D1** and the quotient and the remainder replaces Operand1. | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand, Decimal Overflow, Data  Exception | |
| **Programming Considerations & Tips:**  The instruction always uses L2 bytes for the remainder and L1 – L2bytes for the quotient.  The sign of the quotient is determined from those of the operands. The sign of the remainder is that of the dividend. The rule applies for zero results.  The DP instruction restricts the operands as follows:  L2 <= 8 and L2 < L1 (Violation results in specification exception)  The leftmost digit of the first operand must be zero. (Violation results in decimal divide exception. The same will occur if the divisor is zero). | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 00 00 00 00 00 01 23 4C 0A BC ..  [0C DB 10 0C] 🡪 08 6C C6 ..  ↓  **DP 0(8,R7) , 12(2,R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 00 00 00 00 01 4C 03 0C 0A BC ..  [0C DB 10 0C] 🡪 08 6C C6 .. | |

### CP – Compare Packed

**SS**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CP D1 ( L1, B1), D2 ( L2, B2)** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **F9** | **L1** | **L2** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  It compare the packed decimal number provided at address locations specified by **X2**, **B2**, **D2** against that specified at **X1**, **B1**, **D1** and sets the condition code accordingly | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Operand1  = Operand2 | | 1 | Operand1  < Operand2 | | 2 | Operand1  > Operand2 | | |
| **Exception:** Access – Fetch 1st Operand, Access – Fetch 2nd Operand, Data | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 09 99 99 9F 0A BC ..  [0C DB 10 0C] 🡪 10 00 00 00 1F 7E C6 ..  ↓  **CP 0(4,R7) , 12(5,R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = 1  [0C DC 04 08] 🡪 09 99 99 9F 0A BC ..  [0C DB 10 0C] 🡪 10 00 00 00 1F 7E C6 .. | |

### ZAP – Zero and Add Packed

**SS**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ZAP D1 ( L1, B1), D2 ( L2, B2)** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **F8** | **L1** | **L2** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  It zeroes the packed decimal number provided at address location specified by **X1**, **B1**, **D1** and adds the packed decimal number at **X2**, **B2**, **D2** by either truncating or padding with zeroes (from left) to match **L1** and the result replaces Operand1 | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand, Decimal Overflow, Data | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 Irrelevant  [0C DB 10 0C] 🡪 56 78 9F 7E C6 ..  ↓  **ZAP 0(5,R7) , 12(3,R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = 2  [0C DC 04 08] 🡪 00 00 56 78 9F ..  [0C DB 10 0C] 🡪 56 78 9F 7E C6 .. | |

### SRP – Shift and Round Packed

**SS**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **SRP D1 ( L1, B1), D2 ( B2), I3** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **F0** | **L1** | **I3** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  It shifts the packed decimal number stored at location specified by **B1** and **D1** fields by the value specified in the **right most 6 bits of** **Operand2** which is in 2’s complement notation  A positive value in Operand**2** indicates a left shift whereas a negative value shifts right. When right shift is specified, the Operand**1** is rounded by decimally adding **I3** to the left most digit shifted out and by propagating the carry, if any, to the left  **Note:** Operand2 is not used as an address instead the left most 10 bits are ignored thus utilizing only the right most 6 bits | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero; no Overflow | | 1 | Result is Negative; no Overflow | | 2 | Result is Positive; no Overflow | | 3 | Overflow | | |
| **Exception:** Access – Fetch 2nd Operand, Decimal Overflow, Data | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R2 = 0C DB 10 00 CC = Irrelevant  [0C DB 10 0C] 🡪 00 24 96 5F 38 23 ..  ↓  **SRP 12(4,R2),B’111110’,5**  ↓  R2 = 0C DB 10 00 CC = 2  [0C DB 10 0C] 🡪 00 00 25 0F 38 23 .. | |

## 

## Decimal - Zoned

### Zoned Decimal Number

A Zoned Decimal number can be a maximum of 16-byte long in which each byte’s left nibble (known as zone bits) hold a zone value usually an hex ’F’ and the right nibble (known as numeric bits) may hold number, alphabet or a special character.

The right most zone bits (i.e. last but one nibble) may also be used to hold the sign of number and this can range from A-F in which B & D are used to represent negative numbers.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Zone Bits | Num Bits | Zone Bits | Num Bits | … | Zone Bits | Num Bits | Zone/Sign | Num Bits |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 4 | 8 | 12 | 16 | 116 | 120 | 124 | 127 |

There are no decimal-arithmetic instructions which operate on decimal numbers in the zoned format; such numbers must first be converted to packed format using PACK/UNPK.

### 

**Examples:**

**Positive Zoned Decimal Numbers**

* F1 C3 + 13
* F0 F5 A6 + 56
* F3 F7 F5 + 375

**Negative Zoned Decimal Numbers**

* F1 D3 - 13
* F0 F5 B6 - 56
* F3 F7 D5 - 375

### PACK– Pack

**SS**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **PACK D1 ( L1, B1) , D2 ( L2, B2)** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **F2** | **L1** | **L2** | **B1** | **D1** | **B2** | **D2** | |
| It converts the zoned decimal number in address location specified by **X2**, **B2**, and **D2** fields of length  **L2** into a packed decimal number of length **L1** and replaces Operand**1** | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand | |
| **Programming Considerations & Tips:**   * When Padding or Truncation is needed on result, it is performed from left to right * Padding is done with binary zeroes | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00  [0C DB 10 0C] 🡪 F9 F7 F6 C6 ..  [0C DC 04 08] 🡪 Irrelevant  ↓  **PACK 0(5,R7) , 12(2,R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00  [0C DB 10 0C] 🡪 F8 F9 F7 7E C6 ..  [0C DC 04 08] 🡪 00 00 00 09 7F.. | |

### UNPK– Unpack

**SS**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **UNPK D1 ( L1, B1) , D2 ( L2, B2)** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **F3** | **L1** | **L2** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  It converts the packed decimal number in address location specified by **X2**, **B2**, and **D2** fields of length  **L2** into a zoned decimal number of length  **L1** and replaces Operand**1** | |
| **Condition Code Change:** No | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand | |
| **Programming Considerations & Tips:**   * When Padding or Truncation is needed on result, it is performed from left to right. * Padding is done with value x’F0’ | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00  [0C DB 10 0C] 🡪 24 6C E2 57 ..  [0C DC 04 08] 🡪 Irrelevant  ↓  **UNPK 0(4,R7) , 12(2,R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00  [0C DB 10 0C] 🡪 24 6C E2 57 ..  [0C DC 04 08] 🡪 F0 F2 F4 C6 .. | |

## 

## Shift

### SLL – Shift Left Logical

**RS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SLL R1,D2 (B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **89** | **R1** |  | **B2** | **D2** | |
| **Description:**  It left shifts bits in register **R1** as per the iteration value specified in the **rightmost six bits of Operand2**  **Note:** Operand2 is not used as an address instead the left most 10 bits are ignored thus utilizing only the right most 6 bits | |
| **Condition Code Change**: No | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * Bits 12-15 of this instruction are insignificant. * It comes handy when the register content is to be multiplied by powers of 2 | |
| **Instance:**  R1 = 7F FF FF FF  ↓  **SLL R1 ,8**  ↓  R1 = FF FF FF 00 | |

### SRL – Shift Right Logical

**RS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SRL R1,D2 (B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **88** | **R1** |  | **B2** | **D2** | |
| **Description:**  It right shifts bits in register **R1** as per the iteration value specified in the **rightmost six bits of Operand2**  **Note:** Operand2 is not used as an address instead the left most 10 bits are ignored thus utilizing only the right most 6 bits | |
| **Condition Code Change:** No | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * Bits 12-15 of this instruction are insignificant. * It comes handy when the register content is to be divided by powers of 2 | |
| **Instance:**  R1 = 7F FF FF FF  ↓  **SRL R1 ,4**  ↓  R1 = 07 FF FF FF | |

### SLA – Shift Left Arithmetic

**RS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SLA R1,D2 (B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **8B** | **R1** |  | **B2** | **D2** | |
| **Description:**  It left shifts bits in register **R1** (excluding the sign bit) as per the iteration value specified in the **rightmost six bits of Operand2**  **Note:** Operand2 is not used as an address instead the left most 10 bits are ignored thus utilizing only the right most 6 bits | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Negative | | 2 | Result is Positive | | 3 | Overflow | | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * Bits 12-15 of this instruction are insignificant. | |
| **Instance:**  R1 = 1F FF FF FF  ↓  **SLA R1 ,2**  ↓  R1 = 7F FF FF FC | |

### SRA – Shift Right Arithmetic

**RS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SRA R1,D2 (B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **8A** | **R1** |  | **B2** | **D2** | |
| **Description:**  It right shifts bits in register **R1** (excluding the sign bit) as per the iteration value specified in the **rightmost six bits of Operand2** and the shifted out bits are replaced with sign bit  **Note:** Operand2 is not used as an address instead the left most 10 bits are ignored thus utilizing only the right most 6 bits | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Negative | | 2 | Result is Positive | | 3 | Overflow | | |
| **Exception:** None | |
| **Programming Considerations & Tips:**   * Bits 12-15 of this instruction are insignificant. | |
| **Instance:**  R1 = CF FF FF 7F  ↓  **SRA R1 ,3**  ↓  R1 = F9 FF FF EF | |

## Logical

### Logical Operations

The three logical operations available are AND, OR and XOR (read as Exclusive OR).

These operations are widely performed on data that needs bit level manipulation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operands | | Operation | | |
| Bit X | Bit Y | **AND** | **OR** | **XOR** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

### NR – AND Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **NR R1 , R2** | |  |  |  | | --- | --- | --- | | **14** | **R1** | **R2** | |
| **Description:**  The AND-ed contents of register **R1** and **R2** are placed in register **R1** and the condition code is set accordingly | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** None | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R4 = 0F E1 2D C3 R5 = F0 1E D2 3C CC = Irrelevant  ↓  **NR R4 , R5**  ↓  R4 = 00 00 00 00 R5 = F0 1E D2 3C CC = 0 | |

### N – AND

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **N R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **54** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The contents of register **R1** and **four byte contents** stored in the effective address specified by the **X2**, **B2**, and **D2** are AND-ed and the condition code is set accordingly leaving the result in register **R1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R7 = FF 00 3E D8 R2 = 0C DB 10 00 CC = Irrelevant  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 12 89 34 27 AB 10 .. ..  ↓  **N R7 , 10(R5,R2)**  ↓  R7 = 12 00 34 00 R2 = 0C DB 10 00 CC = 1  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 12 89 34 27 AB 10 .. .. | |

### NI – AND Immediate

**SI**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **NI D1 (B1), I2** | |  |  |  |  | | --- | --- | --- | --- | | **94** | **I2** | **B1** | **D1** | |
| **Description:**  The immediate operand **I2** and **one byte content** stored in the effective address specified by the **B1** and **D1** are AND-ed and the condition code is set accordingly replacing **Operand1** with the result | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** Access – Store 1st Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R2 = 0C DB 10 00 CC = Irrelevant  [0C DB 10 0C] 🡪 87 00 00 .. ..  ↓  **NI 12(R2), X’FA’**  ↓  R2 = 0C DB 10 00 CC = 1  [0C DB 10 0C] 🡪 82 00 00 .. .. | |

### NC – AND Character

**SS**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **NC D1 (L, B1), D2 (B2)** | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **D4** | **L** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  The data of length **L** fetched from the effective address specified by **B2**, and **D2** fields are AND-ed with contents specified by the **B1**, and **D1** fields and the condition code is set accordingly leaving the result in **Operand1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 31 23 0A BC ..  [0C DB 10 0C] 🡪 05 67 1F 7E C6 ..  ↓  **NC 0(2,R7) , 12(R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = 1  [0C DC 04 08] 🡪 01 23 0A BC ..  [0C DB 10 0C] 🡪 05 67 1F 7E C6 .. | |

### 

### OR – OR Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OR R1 , R2** | |  |  |  | | --- | --- | --- | | **16** | **R1** | **R2** | |
| **Description:**  The OR-ed contents of register **R1** and **R2** are placed in register **R1** and the condition code is set accordingly | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** None | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R4 = 0F E1 2D C3 R5 = F0 1E D2 3C CC = Irrelevant  ↓  **OR R4 , R5**  ↓  R4 = FF FF FF FF R5 = F0 1E D2 3C CC = 1 | |

### O – OR

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **O R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **56** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The contents of register **R1** and **four byte contents** stored in the effective address specified by the **X2**, **B2**, and **D2** are OR-ed and the condition code is set accordingly leaving the result in register **R1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R7 = FF 00 3E D8 R2 = 0C DB 10 00 CC = Irrelevant  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 12 89 34 27 AB 10 .. ..  ↓  **O R7 , 10(R5,R2)**  ↓  R7 = FF 89 3E FF R2 = 0C DB 10 00 CC = 1  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 12 89 34 27 AB 10 .. .. | |

### OI – OR Immediate

**SI**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **OI D1 (B1), I2** | |  |  |  |  | | --- | --- | --- | --- | | **96** | **I2** | **B1** | **D1** | |
| **Description:**  The immediate operand **I2** and **one byte content** stored in the effective address specified by the **B1** and **D1** are OR-ed and the condition code is set accordingly replacing **Operand1** with the result | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** Access – Store 1st Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R2 = 0C DB 10 00 CC = Irrelevant  [0C DB 10 0C] 🡪 87 00 00 .. ..  ↓  **OI 12(R2), X’FA’**  ↓  R2 = 0C DB 10 00 CC = 1  [0C DB 10 0C] 🡪 FF 00 00 .. .. | |

### OC – OR Character

**SS**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **OC D1 (L, B1), D2 (B2)** | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **D6** | **L** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  The data of length **L** fetched from the effective address specified by **B2**, and **D2** fields are OR-ed with contents specified by the **B1**, and **D1** fields and the condition code is set accordingly leaving the result in **Operand1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 31 23 0A BC ..  [0C DB 10 0C] 🡪 05 67 1F 7E C6 ..  ↓  **OC 0(2,R7) , 12(R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = 1  [0C DC 04 08] 🡪 35 67 0A BC ..  [0C DB 10 0C] 🡪 05 67 1F 7E C6 .. | |

### XR – XOR Register

**RR**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **XR R1 , R2** | |  |  |  | | --- | --- | --- | | **17** | **R1** | **R2** | |
| **Description:**  The XOR-ed contents of register **R1** and **R2** are placed in register **R1** and the condition code is set accordingly | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** None | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R4 = CF ED 6F EB R5 = F0 1E D2 3C CC = Irrelevant  ↓  **XOR R4 , R5**  ↓  R4 = 3F F3 BD D7 R5 = F0 1E D2 3C CC = 1 | |

### X – XOR

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **X R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **57** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  The contents of register **R1** and **four byte contents** stored in the effective address specified by the **X2**, **B2**, and **D2** are XOR-ed and the condition code is set accordingly leaving the result in register **R1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** Access – Fetch 2nd Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R7 = FF 00 3E D8 R2 = 0C DB 10 00 CC = Irrelevant  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 12 89 34 27 AB 10 .. ..  ↓  **X R7 , 10(R5,R2)**  ↓  R7 = ED 89 0A FF R2 = 0C DB 10 00 CC = 1  R5 = 00 00 00 02 [0C DB 10 0C] 🡪 12 89 34 27 AB 10 .. .. | |

### XI – XOR Immediate

**SI**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **OI D1 (B1), I2** | |  |  |  |  | | --- | --- | --- | --- | | **97** | **I2** | **B1** | **D1** | |
| **Description:**  The immediate operand **I2** and **one byte content** stored in the effective address specified by the **B1** and **D1** are XOR-ed and the condition code is set accordingly replacing **Operand1** with the result | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** Access – Store 1st Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R2 = 0C DB 10 00 CC = Irrelevant  [0C DB 10 0C] 🡪 87 00 00 .. ..  ↓  **XI 12(R2), X’FA’**  ↓  R2 = 0C DB 10 00 CC = 1  [0C DB 10 0C] 🡪 7D 00 00 .. .. | |

### XC – XC Character

**SS**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **XC D1 (L, B1), D2 (B2)** | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **D7** | **L** | **B1** | **D1** | **B2** | **D2** | |
| **Description:**  The data of length **L** fetched from the effective address specified by **B2**, and **D2** fields are XOR-ed with contents specified by the **B1**, and **D1** fields and the condition code is set accordingly leaving the result in **Operand1** | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | Result is Zero | | 1 | Result is Non-Zero | | |
| **Exception:** Access – Fetch 2nd Operand, Access – Store 1st Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = Irrelevant  [0C DC 04 08] 🡪 31 23 0A BC ..  [0C DB 10 0C] 🡪 05 67 1F 7E C6 ..  ↓  **OC 0(2,R7) , 12(R2)**  ↓  R7 = 0C DC 04 08 R2 = 0C DB 10 00 CC = 1  [0C DC 04 08] 🡪 34 44 0A BC ..  [0C DB 10 0C] 🡪 05 67 1F 7E C6 .. | |

## Others

### TM – Test Under Mask

**SI**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TM D1 ( B1), I2** | |  |  |  |  | | --- | --- | --- | --- | | **91** | **I2** | **B1** | **D1** | |
| **Description:**  The one byte content at location specified by Operand**1** is AND-ed with the mask provided in  **I2** and the condition code is set accordingly leaving the operands unchanged | |
| **Condition Code Change:** Yes   |  |  | | --- | --- | | 0 | All tested bits are zero | | 1 | Mixed | | 2 | Mixed | | 3 | All tested bits are one | | |
| **Exception:** Access – Fetch 1st Operand | |
| **Programming Considerations & Tips:** None | |
| **Instance:**  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 BF 13 1F .. CC = Irrelevant ↓  **TM 12(R2),X’F0’**  ↓  R2 = 0C DB 10 00 [0C DB 10 0C] 🡪 BF 13 1F .. CC = 1 | |

### EX – Execute

**RX**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **EX R1 , D2 ( X2, B2)** | |  |  |  |  |  | | --- | --- | --- | --- | --- | | **44** | **R1** | **X2** | **B2** | **D2** | |
| **Description:**  It executes the target instruction available at location specified by Operand**2** after OR-ing its second byte with the right most byte of register **R1.**  After execution of the target instruction the CPU continues with instruction coded after EX | |
| **Condition Code Change:** Depends on the target instruction executed | |
| **Exception:** Access – Fetch 2nd Operand, Execute, Specification | |
| **Programming Considerations & Tips:**   * The OR-ing is done at hardware level and hence the second byte of the target instruction is not permanently changed * The target instruction cannot be an EX instruction | |
| **Instance:**  R1 = 00 00 00 05  ↓  **EX R1 , DYNAMIC**  **.**  **.**  **DYNAMIC CLC 0(0,R7),12(R2) 🡪 D5 00 70 00 20 0C**  The highlighted byte is OR-ed with 05 (from R1) and 6 bytes are compared with the CLC instruction | |

# Assembler Directives

## DC - Define Constant

* The DC directive is used to allocate storage at assembly time and initialize it
* **Format:**

**[label] DC [duplication factor]type[length]’constant’**

Where,

* + Label (optional) – A name used to refer the constant
  + Duplication factor (optional) – A decimal value that specifies the number of times the storage has to be repeated. Default is 1
  + Type (mandatory) - The type of constant
    - C – Character
    - X – Hexadecimal
    - B – Byte
    - F – Full word
    - H – Half word
    - D – Double word
    - P – Packed
    - A – Address
  + Length (optional) – The length of the storage the assembler has to allocate. It has to be prefixed by character **L**
  + Constant (Mandatory) – The initial value to be assigned to the allocated storage area
* Example:

|  |  |  |
| --- | --- | --- |
| **DC** | **Listing** | **Length** |
| DC C’ABCDE’ | C1C2C3C4C5 | 5 |
| DC 3C’A’ | C1C1C1 | 3 |
| DC 3CL2’AB’ | C1C2C1C2C1C2 | 6 |
| DC X’123A’ | 123A | 2 |
| DC XL4’123A’ | 0000123A | 4 |
| DC B’01111010’ | 7A | 1 |
| DC F’-2’ | FFFFFFFE | 4 |
| DC 2F’1’ | 0000000100000001 | 8 |
| DC H’-2’ | FFFE | 2 |
| DC P’1432’ | 01432C | 3 |
| DC PL4’1432’ | 0001432C | 4 |

## DS - Define Storage

* The DS directive is used to allocate storage without initializing it
* **Format:**

**[label] DS [duplication factor]type[length]**

Where,

* + Label (optional) – A name used to refer to the storage
  + Duplication factor (optional) – A decimal value that specifies the number of times the storage has to be repeated. Default is 1
  + Type (mandatory) - The type of constant
  + Length (optional) – The length of the storage the assembler has to allocate. It has to be prefixed by character **L**
* Example:

|  |  |  |
| --- | --- | --- |
| **DS** | **Length** | **Remark** |
| DS CL5 | 5 |  |
| DS 4CL5 | 20 |  |
| DS XL4 | 4 |  |
| DS 0F | 0 | Forces full word boundary alignment |
| DC PL3 | 3 |  |

## EQU – Equate

* The EQU directive is used to define a symbol or a label
* Unlike DC and DS directive, it does not use any storage
* It is used to represent a constant or an address
* Format:

**label EQU expression**

Where,

* + Label – A name used to refer the value of the **expression**
  + Expression – An absolute or re-locatable expression

Note:

The value of absolute expressions is independent of its location in the memory where as the value of a re-locatable expression depends on where the program is loaded

* Example:

|  |  |
| --- | --- |
| **EQU** | **Remark** |
| #FLTCNT EQU 5 | Absolute |
| #TABLEN EQU 5\*L’TABITM | Absolute |
| ERRMSG DC C‘SYSTEM ERROR’  ERRSIZ EQU \*-ERRMSG | Re-locatable |
| BAS R7,ROUTINE  :  ROUTINE EQU \* | Re-locatable |

## DSECT – Define Dummy Section

* The DSECT directive is used to define layout of data in a storage area
* It does not allocate any storage
* The layout is defined using only the DS directive
* The symbols defined in the DSECT can be referenced by including the USING directive that associates the layout to the storage location through a base register
* Format: **Label DSECT**
* Example:

TRAINEE DSECT TRAINEE RECORD LAYOUT

\*

FNAME DS CL20 FIRST NAME

LNAME DS CL20 LAST NAME

\*

GENDER DS CL1 GENDER

#MALE EQU C’M’

#FEMALE EQU C’F’

\*

DOB DS 0CL9 DATE OF BIRTH

DD DS CL2 DAY IN DD FORMAT

MMM DS CL3 MONTH IN MMM FORMAT

YYYY DS CL4 YEAR IN YYYY FORMAT

\*

DS 0H

AGE DS H AGE

\*

LABSCR DS XL2 LAB 1 SCORE

DS XL(4\*L’LABSCR) SPACE FOR 4 MORE LAB SCORES

\*

GRADE DS X GRADE

\* BIT0 = 1: EXCEPTIONAL

\* BIT1 = 1: VERY GOOD

\* BIT2 = 1: GOOD

\* BIT3 = 1: AVERAGE

\* BIT4-7: NOT USED

\*

TRALEN EQU \*- TRAINEE LENGTH OF THE RECORD

In the mainline:

USING TRAINEE, R2

.

.

L R2,HEAPADDR

MVC FNAME(5),=C’DAVID’

.

.

The USING statement in the assembler program uses R2 as the base register whenever referring to any of the symbols in the TRAINEE DSECT.

## ORG – Origin

* The ORG directive instructs the assembler to change the value of the current location counter to that of **Label**
* Typically it is used to redefine storage layouts within a DSECT
* Format: **ORG Label**
* Example:

|  |  |
| --- | --- |
| **Location Counter (hex)** | **DSECT definition** |
|  | EMP DSECT |
| 0000 | NAME DS CL20 NAME |
| 0014 | GEN DS C GENDER |
| 0015 | DOB DS CL9 DATE OF BIRTH |
|  | ORG DOB |
| 0015 | DD DS CL2 DAY IN DD FORMAT |
| 0017 | MMM DS CL3 MONTH IN MMM FORMAT |
| 001A | YYYY DS CL4 YEAR IN YYYY FORMAT |
| 001E | AGE DS X AGE |

## USING – Define Base Register

* The USING directive is used to associate a DSECT to a storage location through a base register
* Format**: USING Label,R1**

## DROP – Drop Base Register

* This assembler directive cancels previous USING on one or more registers
* Format: **DROP** **R1**, [**R2**, **R3**, …]

## CSECT – Define Control Section

* The CSECT directive indicates the beginning of the program
* If preceded by a DSECT, it marks the end of the DSECT and the beginning of the program