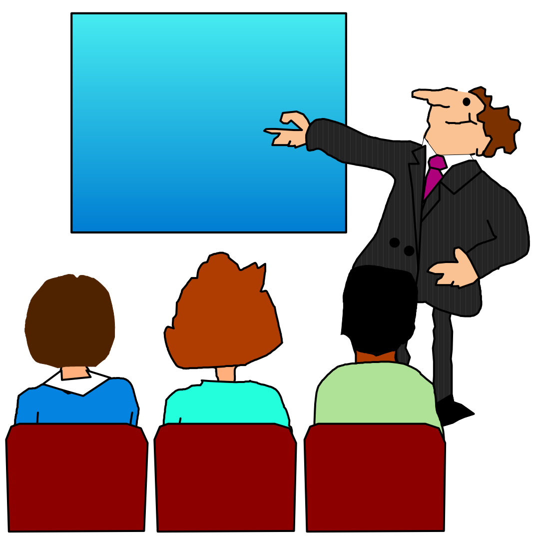
Assembler Language "Boot Camp" Part 5 - Decimal and Logical Instructions

SHARE in San Francisco

August 18 - 23, 2002

Session 8185



Introduction

- Who are we?
  - John Dravnieks, IBM Australia
  - John Ehrman, IBM Silicon Valley Lab
  - Michael Stack, Department of Computer Science, Northern Illinois University

Introduction

- Who are you?
  - An applications programmer who needs to write something in S/390 assembler?
  - An applications programmer who wants to understand S/390 architecture so as to better understand how HLL programs work?
  - A manager who needs to have a general understanding of assembler?
- Our goal is to provide for professionals an introduction to the S/390 assembly language

Introduction

- These sessions are based on notes from a course in assembler language at Northern Illinois University

- The notes are in turn based on the textbook, Assembler Language with ASSIST and ASSIST/I by Ross A Overbeek and W E Singletary, Fourth Edition, published by Macmillan

Introduction

- The original ASSIST (Assembler System for Student Instruction and Systems Teaching) was written by John Mashey at Penn State University

- ASSIST/I, the PC version of ASSIST, was written by Bob Baker, Terry Disz and John McCharen at Northern Illinois University

Introduction

- Both ASSIST and ASSIST/I are in the public domain, and are compatible with the System/370 architecture of about 1975 (fine for beginners)

- Both ASSIST and ASSIST/I are available at http://www.cs.niu.edu/~mstack/assist

Introduction

- Other materials described in these sessions can be found at the same site, at http://www.cs.niu.edu/~mstack/share

- Please keep in mind that ASSIST and ASSIST/I are not supported by Penn State, NIU, or any of us

Introduction

- Other references used in the course at NIU:
  - Principles of Operation
  - System/370 Reference Summary
  - High Level Assembler Language Reference

- Access to PoO and HLASM Ref is normally online at the IBM publications web site

- Students use the S/370 "green card" booklet all the time, including during examinations (SA22-7209)

Our Agenda for the Week

- Session 8181: Numbers and Basic Arithmetic
- Session 8182: Instructions and Addressing
- Session 8183: Assembly and Execution; Branching

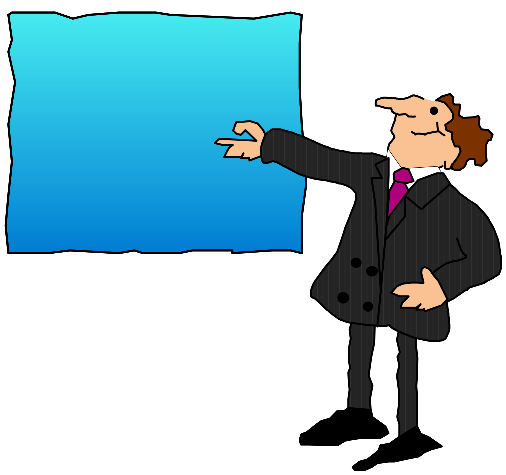
Our Agenda for the Week

- Session 8184: Arithmetic; Program Structures
- <u>Session 8185: Decimal and Logical Instructions</u>
- Session 8186: Assembler Lab Using ASSIST/I

Today's Agenda

- The SI and SS Instruction Formats
- Decimal Arithmetic
- Instructions for Logical Operations
- Wrap Up

The SI and SS Instruction Formats



SI Instructions

- This format encodes the second operand as an "immediate" data byte within the instruction

- The symbolic instruction format is
  - **`label    mnemonic   address,byte`**

- The encoded form of an SI instruction is
  - $\mathbf{h_{OP}h_{OP}h_{I2}h_{I2}\ h_{B1}h_{D1}h_{D1}h_{D1}}$

SI Instructions

- MOVE IMMEDIATE is our first SI instruction
  - `label MVI` $D_1(B_1),I_2$
  - Stores a copy of the immediate byte, I_2, at the memory location given by $D_1(B_1)$

SI Instructions

- The second operand can be specified as a decimal number or as any one-byte value valid in DC
  - Decimal: **91**
  - Hexadecimal: **X'5B'**
  - Binary: **B'01011011'**
  - Character: **C'$'**

- For example, to place a single blank at PLINE
  - ```
             MVI  PLINE,C' '
    ```
    ```

SI Instructions

- The COMPARE LOGICAL IMMEDIATE instruction compares the byte in memory to the immediate data byte as unsigned binary numbers
  - **label CLI $D_1(B_1),I_2$**

- CLI sets the condition code in the same way as other compare instructions

SI Instructions

- The following code sample scans an 80-byte data area labelled CARD and replaces blanks with zeros

```
         ...
         LA    4,CARD      Start scan here
         LA    3,80         and scan 80 bytes
SCAN     CLI   0(4),C' '   Look for blank
         BNE   BUMP        Branch if not blank
         MVI   0(4),C'0'   Else change to 0
BUMP     LA    4,1(,4)     Move to next byte
         BCT   3,SCAN      Continue for 80
         ...
```

SS Instructions

- In this format, which occupies 3 halfwords, both operands reference memory locations, and there is either one 256-byte-max length or two 16-byte-max lengths

- The symbolic instruction format is either
  - label mnemonic addr**1**(len),addr**2** or
  - label mnemonic addr**1**(len**1**),addr**2**(len**2**)

SS Instructions

- Each SS instruction is defined to have one of the length formats; we will see only the first for now

- The encoded form of an SS instruction is
  - $\mathbf{h_{OP}h_{OP}h_Lh_L \quad h_{B1}h_{D1}h_{D1}h_{D1} \quad h_{B2}h_{D2}h_{D2}h_{D2}}$ or
  - $\mathbf{h_{OP}h_{OP}h_{L1}h_{L2} \quad h_{B1}h_{D1}h_{D1}h_{D1} \quad h_{B2}h_{D2}h_{D2}h_{D2}}$

SS Instructions

- **Very Important**: the encoded length is one less than the symbolic length (as well as the effective length) and is referred to as the "length code"

- Thus, in the first format, 1 to 256 bytes may be specified where 0 to 255 is encoded

- An explicit length of 0 results in an encoded length of 0, so the effective length is 1

SS Instructions

- MOVE CHARACTERS is our first SS instruction

  - `label   MVC   D1(L,B1),D2(B2)` ($D_1(L,B_1),D_2(B_2)$)

  - Copies from 1 to 256 bytes from the second operand location to the first

SS Instructions

- For example, to copy 8 bytes from the location addressed by register **`1`** to **`14`** bytes beyond the location addressed by register **`12`**
  - Symbolic: **`MVC 14(8,12),0(1)`**
  - Encoded: **`D207 C00E 1000`**
  - Note the encoded length!

SS Instructions

- Implicit addresses may be used, of course, and with or without an explicit length
  - `MVC    FIELD1(15),FIELD2`
  - `MVC    FIELD1,FIELD2`

- Both generate the same object code if **`FIELD1`** (the first operand) has a "length attribute" of **`15`**, as in
  - `FIELD1    DS    CL15`

SS Instructions

- Any explicit length will take precedence over the implicit length derived from the length attribute

- So, in the previous example the following instruction will move only 8 bytes, even though **`FIELD1`** has a length of 15
  - **`MVC    FIELD1(8),FIELD2`**

- Implicit lengths change automatically when data lengths changes

SS Instructions

- The effect of MVC is to replace L bytes beginning at the first operand location with a copy of the L bytes beginning at the second operand location

- The target is altered, one byte at a time, starting on the "left" (the beginning, or low, address)

SS Instructions

- This means that the fields can overlap with predictable results, and here is an historically important example

- There is often a "print buffer" in which output lines are constructed, and after printing a line, the buffer should be cleared to blanks (this assumes the PLINE has a length attribute of 133, as it would if `PLINE   DS   CL133` is used)

SS Instructions

- So, we would normally clear the buffer by copying a string of blanks to it
  - ```
             MVC   PLINE,=CL133' '
    ```

- But by using the overlap, we can "save" 129 bytes
  - ```
             MVI   PLINE,C' '
    ```
  - ```
             MVC   PLINE+1(132),PLINE
    ```
    ```

SS Instructions

- Suppose `FIELD DC C'123456'` What is FIELD after `MVC FIELD+2(4),FIELD` ?
- `C'121212'`

SS Instructions

- Another SS instruction which uses the first length format is COMPARE LOGICAL CHARACTERS
  - **label CLC $D_1(L,B_1),D_2(B_2)$**
- As with all compares, this just sets the condition code
- The operation stops when the first unequal bytes are compared

Decimal Arithmetic

In Which We Switch to Counting
on Our Fingers or Toes
Instead of Our Hands

Decimal Data

- Thus far, the computations we've done have been with binary data
- This is not always satisfactory, especially when financial calculations are required
- For example, decimal percentages are inaccurate in binary (try long division on $\mathbf{1/10_{10}} = \mathbf{1/1010_2 = .000110011...}$)
- This (infinite repetition) annoys auditors

Decimal Data

- The solution is to use computers with decimal data types and instructions

- There are two decimal data formats
  - Zoned Decimal - associated with I/O operations
  - Packed Decimal - used for decimal arithmetic

Decimal Data

- A <u>zoned decimal</u> number is a sequence of bytes in which each byte has
  - a decimal digit 0-9 in the right digit and
  - a zone digit (hex F) in the left digit, except that the rightmost zone is the sign

Decimal Data

- That is, a zoned decimal number has the format
  - **`ZdZdZd...sd`** where
    - **`Z`** is the hex digit **`F`**
    - **`d`** is a decimal digit 0-9
    - **`s`** is the sign
      - **`C, A, F,`** or **`E`** means **+** (**`C`** is preferred)
      - **`D`** or **`B`** means **-** (**`D`** is preferred)

- An example is **`F1F2C3,`** for +123

Decimal Data

- A zoned number is very close to the EBCDIC representation of its value, except that the rightmost byte has a sign, so won't print as a number
  - So our zoned +123 prints as **`12C`**

Decimal Data

- A packed decimal number has the zones removed and the sign switched with its digit; that is,
  - **dddddd...ds**

- Note that there are always an odd number of digit positions in a packed decimal number

- The assembler can generate data of types Z (zoned) and P (packed)

Decimal Data

- label DC mZLn'z'
  - DC Z'+123' = F1F2C3
  - DC ZL3'-1.2' = F0F1D2
- label DC mPLn'p'
  - DC P'+123' = 123C
  - DC P'-1.2' = 012D
  - DC PL2'1234' = 234C

The PACK and UNPK Instructions

- Both of these are SS instructions of the second type - that is, <u>each operand has a length field</u> which will accommodate a length code of **`0-15`** (so the effective length is **`1-16`** bytes)

The PACK and UNPK Instructions

- Use the PACK instruction to convert a number from zoned decimal to packed decimal

- Use the UNPK instruction to convert a number from packed decimal to zoned decimal

The PACK and UNPK Instructions

- **`label PACK` $D_1(L_1,B_1),D_2(L_2,B_2)$**
  - The rightmost byte of the second operand is placed in the rightmost byte of the first operand, with zone (sign) and numeric digits reversed
  - The remaining numeric digits from operand 2 are moved to operand 1, right to left, filling with zeros or ignoring extra digits

The PACK and UNPK Instructions

- `|`D_5D_4`|`D_3D_2`|`D_1S`|<-|`ZD_5`|`ZD_4`|`ZD_3`|`ZD_2`|`SD_1`|`
  - where each 'Z' is a zone F

- **`PACK B(1),B(1)`** exchanges a byte's digits

The PACK and UNPK Instructions

- **PACK P(3),Z(4)**
  - `P(3) <---- Z(4)`
  - Before: `?? ?? ??      F5 F4 F3 D2`
  - After: `05 43 2D      F5 F4 F3 D2`

- **PACK P(2),Z(4)**
  - `P(2) <---- Z(4)`
  - Before: `?? ??      F5 F4 F3 C2`
  - After: `43 2C      F5 F4 F3 C2`

The PACK and UNPK Instructions

- **`label  UNPK  `**$D_1(L_1,B_1),D_2(L_2,B_2)$

  - The rightmost byte of the second operand is placed in the rightmost byte of the first operand, with zone (sign) and numeric digits reversed

  - The remaining numeric digits from operand 2 are placed in the numeric digits of operand 1, and the zone digits of all but the rightmost byte of operand 1 are set to F, filling with **`X'F0'`** or ignoring extra digits

The PACK and UNPK Instructions

- **|ZD_5|ZD_4|ZD_3|ZD_2|SD_1|<- |D_5D_4|D_3D_2|D_1S|**
  - where each 'Z' is a zone F

- **UNPK B(1),B(1)** exchanges a byte's digits

The PACK and UNPK Instructions

- **`UNPK Z(5),P(3)`**
  - `Z(5) <------- P(3)`
  - Before: `?? ?? ?? ?? ??      12 34 5C`
  - After: `F1 F2 F3 F4 C5      12 34 5C`

- **`UNPK Z(4),P(2)`**
  - `Z(4) <---- P(2)`
  - Before: `?? ?? ?? ??     12 3F`
  - After: `F0 F1 F2 F3     12 3F`

The CVB and CVD Instructions

- These two RX instructions provide conversions between packed decimal and binary formats

- With PACK and UNPK, we can now convert between zoned and binary formats

The CVB and CVD Instructions

- **`label    CVB    `**$R_1,D_2(X_2,B_2)$
  - Causes the contents of $\mathbf{R_1}$ to be replaced by the binary representation of the <u>packed</u> decimal number in the doubleword (on a doubleword boundary) addressed by operand 2

- A data exception (0007) occurs if operand 2 is not a valid packed decimal number

- A fixed-point divide exception (0009) occurs if the result is too large to fit in a 32-bit word

The CVB and CVD Instructions

For example:

```
            CVB     3,Z
            ...
Z           DS      0D
            DC      PL8'-2'
```

will convert 000000000000002D at location Z
(data type D has doubleword alignment)
to FFFFFFFE in register 3

The CVB and CVD Instructions

- **`label    CVD    R1,D2(X2,B2)`** ($R_1,D_2(X_2,B_2)$)
  - Causes the contents of the doubleword (on a doubleword boundary) addressed by operand 2 to be replaced by the packed decimal representation of the binary number in $\mathbf{R_1}$

- Note that the "data movement" is left to right (like ST)

- The exceptions which apply to CVB (0007 and 0009) do not apply to CVD

Numeric Data Conversion Summary

```
     Data in     PACK    Data in           CVB     Data in
---> character --------> packed decimal --------> binary
     format              format                    format
                            |                         |
                            V                         V
                         Perform                   Perform
                         packed                    binary
                         arithmetic                arithmetic
                            |                         |
                            V                         V
     Results in  UNPK    Results in         CVD    Results in
<--- zoned      <-------- packed decimal <-------- binary
     format      EDIT    format                    format
```

Getting results in nice character format, instead of just zoned, requires use of EDIT instruction

Decimal Arithmetic

- The box encloses the only subject on the previous slide which remains to be addressed: decimal arithmetic

- There isn't enough time to cover the decimal arithmetic instructions in detail, but they all have the following characteristics

Decimal Arithmetic

- Two memory operands, each with its own length
- Condition code is set similar to binary equivalents
- In almost all cases (except operand 1 in ZAP), the operands must be valid packed decimal numbers, else an interrupt 0007 occurs (very popular!)

Decimal Arithmetic

- Here are the available instructions
  - AP - ADD DECIMAL
  - CP - COMPARE DECIMAL
  - DP - DIVIDE DECIMAL
  - MP - MULTIPLY DECIMAL
  - SRP - SHIFT AND ROUND DECIMAL
  - SP - SUBTRACT DECIMAL
  - ZAP - ZERO AND ADD DECIMAL
- With the possible exception of SRP, these are easy to understand - see PoO

Instructions for Logical Operations

To Which We Must Say Yes or No

The Logical Operations

- Consider the four possible combinations of 2 bits
  - `a = 0    0    1    1`
  - `b = 0    1    0    1`

- These lead to the following binary relations
  - `a AND b = 0    0    0    1`
  - `a OR b  = 0    1    1    1`
  - `a XOR b = 0    1    1    0`

The Logical Operations

- And these relations lead to the following twelve new instructions:

	RR Format	RX Format	SI Format	SS Format
AND Operation	NR	N	NI	NC
OR Operation	OR	O	OI	OC
XOR Operation	XR	X	XI	XC

The Logical Operations

anything with	itself	zero	one
AND	It remains unchanged	It is changed to zero	It remains unchanged
OR	It remains unchanged	It remains unchanged	It is changed to one
XOR	It is changed to zero	It remains unchanged	It is inverted

The Logical Operations

- All twelve instructions set the condition code:
  - 0 - Result is zero
  - 1 - Result is not zero

- As an example, to change a zoned decimal number to EBCDIC, we have to force the rightmost zone to be F instead of a sign; so, if ZNUM is a three-byte zoned number, the following instruction will work: (why?)
  - ```
            OI      ZNUM+2,X'F0'
    ```
    ```

The Logical Operations

- To zero a register, we normally use SR, but a faster way to zero R5 (for example) is
  - `XR      5,5`

- To set bit 0 of BYTE to **1** while leaving the other bits unchanged
  - `OI      BYTE,B'10000000'`

- To set bit 0 of BYTE to **0** while leaving the other bits unchanged
  - `NI      BYTE,B'01111111'`

The Logical Operations

- To invert bit 0 of BYTE to **1** while leaving the other bits unchanged
  - `XI   BYTE,B'10000000'`
- To round the address in R7 down to the previous fullword boundary
  - `N    7,=X'FFFFFFFC'`
- To round it up to the next fullword boundary
  - `LA   7,3(,7)`
  - `N    7,=X'FFFFFFFC'`

The Logical Operations

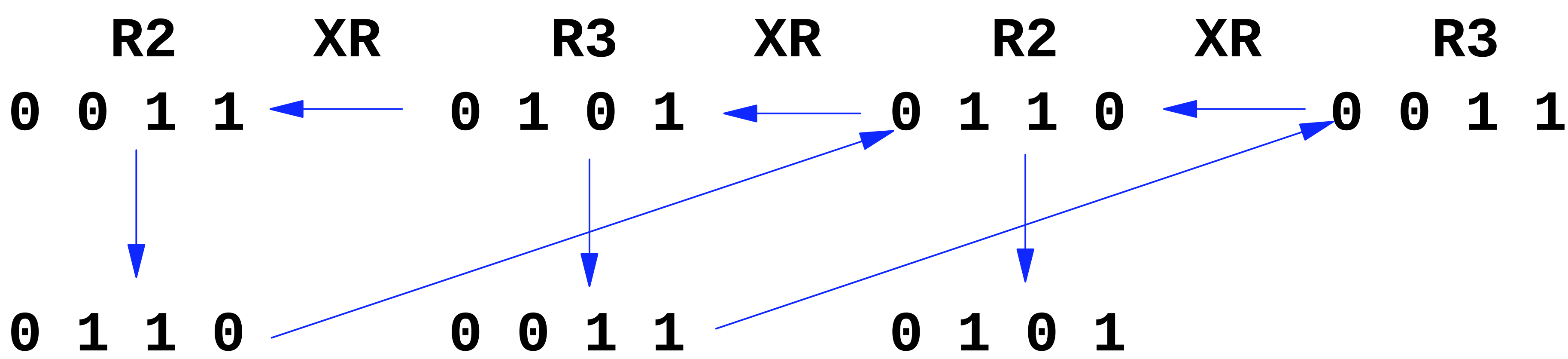
- To exchange the contents of two registers without using any temporary space, use XR three times, alternating registers

- Memory contents can be exchanged similarly by using XC instead of XR

```
XR      2,3   Exchange
XR      3,2    contents of
XR      2,3     registers 2 and 3
```

The Logical Operations

- Here's how that works, demonstrated using "mini" registers with all four posible bit combinations



The TEST UNDER MASK Instruction

- `label TM D1(B1),I2` — $D_1(B_1),I_2$

- TM sets the condition code to reflect the value of the tested bits (those corresponding to **1**-bits in the I_2 operand)
  - 0 - Selected bits all zeros, or the I_2 mask was zero
  - 1 - Selected bits mixed zeros and ones
  - 2 - --- (not set)
  - 3 - Selected bits all ones

The TEST UNDER MASK Instruction

- Note that after TM, the extended branch mnemonics are interpreted as
  - BZ - Branch if tested bits are Zeros, or mask is zero
  - BM - Branch if tested bits are Mixed zeros and ones
  - BO - Branch if tested bits are Ones

The TEST UNDER MASK Instruction

- To determine if the first bit of BYTE is one
  - `TM      BYTE,B'10000000'`

- To check if BYTE is zero or blank (X'00' or X'40')
  - `TM      BYTE,B'10111111'`
  - `BZ      BLKZRO`

Wrap Up

In Which We Learn That Only a Small Fraction of the Assembler Language Has Been Covered

Summary

- Five hours is just a start, but a good one

- The one-semester course at NIU has
  - More than 35 hours of lecture
  - A dozen programs (almost one each week)
  - Three exams

- The second course is Data Structures, and all program assignments are in assembler
  - This is good reinforcement

What Wasn't Covered

- Shift instructions, logical and arithmetic
- Frequently used, but difficult instructions
  - Edit (ED) and Edit and Mark (EDMK)
  - Execute (EX)
  - Translate (TR) and Translate and Test (TRT)
- Floating point instructions
  - Hexadecimal (the original)
  - Binary (IEEE standard, recently added)

What Wasn't Covered

- Many general instructions added over the past twenty-five years, such as
  - Relative BRANCH instructions (no base register needed)
  - Instructions which reference a halfword (immediate) operand within the instruction
  - Instructions to save and set the addressing mode (24-bit or 31-bit)
  - And, most recently, the z/Architecture instructions to deal with 64-bit registers and addresses

What Wasn't Covered

- Privileged instructions
- The macro language, including conditional assembly (also available outside macros)
- The USING instruction, extended to allow implicit addresses everywhere
- External subroutines and register save area linkage conventions

Nevertheless...

- You now have a basic understanding of S/390 and z/Architecture
- You have seen what comprises a program written in assembler language
- And you are ready, if you wish, to begin writing programs and go on to the next step
- So, ...

Congratulations!