Chapter 5: Character Study

In which we examine the instructions that can be used to read, move, compare, and print character data.

A Basic Instruction Set for Character Data Processing

An excellent place to begin programming assembler language is with an application that uses character data. The data is easy to create and read, the instructions are limited to one or two types (SS or SI), and you're unlikely to cause an abend unless you are especially creative at creating errors! Later, when we discuss register operations, we will consider some RR instructions for working with large character fields. For the moment, the following list contains the most significant commands you'll need for processing character data.

- MVC – Move Characters - This is the all-purpose instruction for copying character data between storage locations in memory.
- MVI – Move Immediate - This is the immediate version (SI) of MVC for moves involving one-byte fields
- CLC – Compare Logical Characters - Use this instruction to compare character fields in memory and set the condition code for branching.
- CLI – Compare Logical Immediate - The immediate version (SI) of CLC for comparing single bytes.
- BC – Branch on Condition - An instruction used for conditional logic. We will look at the closely-related extended mnemonics of this instruction.
- BRC – Branch Relative on Condition - An instruction used for conditional logic.
- BRCL – Branch Relative on Condition Long - An instruction used for conditional logic.
- MVN – Move Numerics - A rarely used instruction (these days) that operates on the numeric portion of bytes.
- MVZ – Move Zones - A rarely used instruction (these days) that operates on the zone portion of bytes.

Creating Character Data

Character data is created by specifying a "C" for the data type in a DC declarative. In this format, each character occupies 1 byte of storage. Characters are represented using the EBCDIC encoding sequence, where each character is represented using 8 bits. As result, there are $2^8 = 256$ possible bit patterns or characters which can be formed.

A character field can be created using any of the following formats,

```
name DC dCLn'constant' or
name DS dCLn'constant' or
name DS dCLn
```

- where 'name' is an optional field name
- 'd' is a duplication factor used to create consecutive copies of the field (default = 1 copy)
- 'C' represents the character data type
- 'L' represents an optional length (default = 1 byte)
- 'n' is the number of bytes in the field
- 'constant' is an initial value of the field in character format.

It is important to note that if a name for a field is specified, it will represent the address of the *first* byte of the field. Additionally, the field's name will be associated with a length attribute, which equals the number of bytes in the field. If the "Ln" construction is omitted in a DS, the length will default to 1 byte.

```
X       DS      C         X HAS LENGTH = 1
```

If we omit the "Ln" construction in a DC, the length of the constant determines the field length.

```
Y       DC      C'ABC'          Y HAS LENGTH = 3
```

The length of a field defined using DC is limited to a maximum of 256 bytes, and the length of a field defined using DS is 65,535 bytes.

When specifying a constant, the length of the constant and the length of the field may differ in size. If the length of the constant is shorter than the field length, the assembler will pad the constant on the right with blanks to fill up the field.

```
P       DC      CL5'ABC'          P HAS LENGTH = 5, P = 'ABC  '
```

Programmers often exploit this fact to initialize long fields with blanks. For example,

```
P1      DC      CL80' '         P1 HAS LENGTH = 80, ALL BLANKS
```

When the constant is longer than the field, it will be truncated on the right to fit inside the field. The assembler does not generate a warning message when this occurs.

```
Q       DC      CL3'ABCDE'       Q HAS LENGTH = 3, Q = 'ABC'
```

During assembly, fields that were defined consecutively in the source code with DC's or DS's, are assigned consecutive storage locations in the program according to the value of the location counter, which is maintained by the assembler.

For example, if FIELDA is associated with address x'1000', then FIELDB is located at x'1003,' and FIELDC is located at x'100C'.

LOCATION

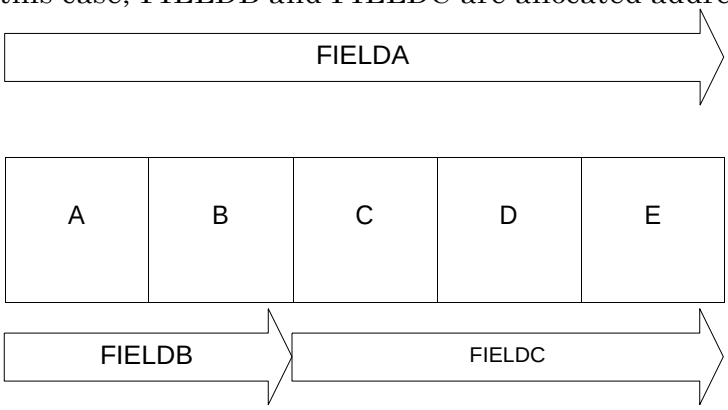
```
1000        FIELDA DS CL3

1003        FIELDB DS CL9

100C        FIELDC DS CL3
```

An exception to the sequential allocation of fields occurs if the duplication factor is specified as 0. In this case, the location counter is not advanced, and the next field redefines the previous field. Consider the following example and note the location counter values.

LOCATION

```
 1000       FIELDA   DS    0CL5
 1000       FIELDB   DC     CL2'AB'
 1002       FIELDC   DC     CL3'CDE'
```

In this case, FIELDB and FIELDC are allocated addresses "inside" FIELDA.

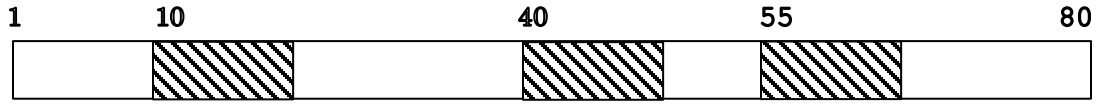


Sequential allocation of storage areas can also be altered by ORG directives. An ORG directive resets the location counter to a named location.

LOCATION

```
 1000       FIELDA   DS     CL5
                     ORG    FIELDA
 1000       FIELDB   DC     CL2'AB'
 1002       FIELDC   DC     CL3'CDE'
```

Let's consider the problem of defining an output buffer that contains 80 characters. Assume that the buffer, RECOUT, contains three 10-byte character fields, FIELDA, FIELDB, and FIELDC that start in columns 10, 40 and 55 respectively.



We can define the structure using only DS and DC directives,

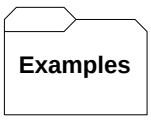
```
RECOUT   DS    0CL80    AN 80-BYTE BUFFER
         DC     CL9' '  UNNAMED 9-BYTE FIELD
FIELDA   DS     CL10
         DC     CL20' ' UNNAMED 20-BYTE FIELD
FIELDB   DS     CL10
         DC     CL5' '  UNNAMED 5-BYTE FIELD
FIELDC   DS     CL10
         DS     CL16' ' UNNAMED 16-BYTE FIELD
```

Using ORG, you might find it easier to create the structure and arrange the fields at specific locations,

```
RECOUT   DC     CL80' ' AN 80-BYTE BUFFER
         ORG    RECOUT+9
FIELDA   DS     CL10
         ORG    RECOUT+39
FIELDB   DS     CL10
         ORG    recout+54
FIELDC   DS     CL10
         ORG    ,
```

We begin by defining an 80-byte buffer initialized with spaces. ORG is used to reset the location counter to specific columns so the three subfields can be defined. The last ORG (with a comma that denotes the operand field was omitted) resets the location counter to the maximum value it had before we began coding ORGs. In other words, the location counter is set to the location of the byte following RECOUT.

A common error for beginning assembler programmers is omitting the last ORG. If that occurs, fields that are defined later may become part of the structure you are defining.



Some Typical DSs and DCs:

Note - L' indicates the length attribute of the following field.

```
A         DS    CL8      AN 8-BYTE CHARACTER FIELD, UNINITIALIZED
B         DS    C        A 1-BYTE CHARACTER FIELD, UNINITIALIZED
C         DS    CL2000   A 2000-BYTE CHARACTER FIELD, UNINITIALIZED
NAME      DS   0CL30     A SUBDIVIDED FIELD, UNINITIALIZED
LNAME     DS    CL15     THE FIRST SUBFIELD
FNAME     DS    CL15     THE SECOND SUBFIELD
BLANK1    DC    CL80' '  80-BYTE FIELD FILLED WITH BLANKS, L'BLANK1=80
BLANK2    DC  80C' '     80 CONSECUTIVE 1-BYTE,BLANK FIELDS,L'BLANK2=1
D         DC    CL2'ABC' CONSTANT TOO LONG, RIGHT TRUNCATED D='AB'
E         DC    CL5'AB'  CONSTANT TOO SHORT, RIGHT PADDED E='AB   '
F         DC   3C'XY'    L'F = 2, THREE FIELDS CREATED
          DC    C'COST'  AN UNNAMED, INITIALIZED FIELD
          ORG   A+8
X         DS    CL4
          ORG   ,
Y         DS    CL4
```

Programming Exercise

Take the first program as a template or skeleton program. Type or copy the list of declarations above into your skeleton. The first question to be answered is *Where do I put them?* The answer is *almost anywhere* – provided you know what you are doing. A better question is *Where can I put them without causing assembly errors?* Try adding the list at the bottom of the program, just after the save area definition and before the LTORG. Now assemble the program again and take a look at the program listing. Notice the location counter values printed on the left side of the listing. Do the values make sense in terms of the fields you are defining?

Here is part of my assembler listing for these statements above,

```
000148                             164 RECIN    DS    CL80       INPUT AREA FOR RECORDS                        00251000
000198                             165 RECOUT   DS    CL80       OUTPUT AREA FOR RECORDS                       00251100
0001E8 0000000000000000            166 SAVEAREA DC    18F'0'     AREA FOR MY CALLEE TO SAVE & RESTORE MY REGS  00252000
000230                             167 A        DS    CL8     AN 8-BYTE CHARACTER FIELD, UNINITIALIZED         00252100
000238                             168 B        DS    C       A 1-BYTE CHARACTER FIELD, UNINITIALIZED          00253000
000239                             169 C        DS    CL2000  A 2000-BYTE CHARACTER FIELD, UNINITIALIZED       00253200
000A09                             170 NAME     DS   0CL30    A SUBDIVIDED FIELD, UNINITIALIZED                00253300
000A09                             171 LNAME    DS    CL15    THE FIRST SUBFIELD                               00253500
000A18                             172 FNAME    DS    CL15    THE SECOND SUBFIELD                              00253801
000A27 4040404040404040            173 BLANK1   DC    CL80' ' 80-BYTE FIELD FILLED WITH BLANKS, L'BLANK1=80    00253902
000A77 4040404040404040            174 BLANK2   DC  80C' '    80 CONSECUTIVE 1-BYTE BLANK FIELDS,L'BLANK2=1    00254002
000AC7 C1C2                        175 D        DC    CL2'ABC' CONSTANT TOO LONG, RIGHT TRUNCATED D='AB'       00254103
000AC9 C1C2404040                  176 E        DC    CL5'AB'  CONSTANT TOO SHORT, RIGHT PADDED E='AB   '      00254203
        SKELETON ASSEMBLER PROGRAM                                                                            Page    5
  Active Usings: CHARPROG+X'6',R12
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                HLASM R6.0  2021/09/01 10.19
000ACE E7E8E7E8E7E8                177 F        DC   3C'XY'   L'F = 2, THREE FIELDS CREATED                    00254303
000AD4 C3D6E2E3                    178          DC    C'COST' AN UNNAMED, INITIALIZED FIELD                    00254506
000AD8                 00AD8 00238 179          ORG   A+8                                                      00254604
000238                             180 X        DS    CL4                                                      00254704
00023C                 0023C 00AD8 181          ORG   ,                                                        00255004
000AD8                             182 Y        DS    CL4                                                      00256004
```

- RECIN starts at location 000148 and is 80 bytes long. A decimal 80 is equivalent to hexadecimal 50, so the address of the next field, RECOUT is x'148' + x'50' = x'000198'.

- Why do fields NAME and LNAME have the same location, x000A09'?

- BLANK1 is defined as an 80-byte field containing blanks, but we only see the first eight bytes of blanks, x'4040404040404040' in the object code on the left because we coded PRINT NODATA. This directive limits the printing of the constants we define to the first eight bytes.

- Field F was defined with a repetition factor of three. The constant we provided was C'XY', so we see a hexadecimal representation of XY as x'E7E8' occurring consecutively three times. F refers to the first byte of the first field at location x'00ACE'

- When we code the final ORG, the location counter is at x'23C'. The ORG resets it to x'AD8'. Why?

Becoming Familiar with Character Data

It's easy to learn the EBCDIC representation of the alphabet since it falls into three blocks of consecutive values:

```
Characters A-I are represented as hex characters C1 … C9

Characters J-R are represented as hex characters D1 … D9
(Junior (JR) is a useful mnemonic for the middle block)

Characters S-Z are represented as hex characters E2 … E9
```

Digits are easy, too:

```
Digits 0-9 are represented as hex characters F0 … F9
```

Self-Defining Terms

A **self-defining term** is an assembler expression that represents a number. There are four types: Character, Decimal, Binary, and Hexadecimal. I mention the idea here because it is customary to use a self-defining term whenever you code an immediate instruction like MVI and CLI - instructions which are covered later in this chapter. So, while a self-defining term is not technically character data, we can represent a self-defining term using a character format.

Each *character* self-defining term begins with the letter "C" and is followed by an apostrophe (') and up to four (usually one) characters excluding apostrophes (') and ampersands (&), and a terminating apostrophe (').

Here are some examples and their numeric values:

```
C'A'  =  193 in decimal  = x'C1'
C' '  =   64 in decimal  = x'40'
```

```
C'1'  =  241 in decimal  = x'C1'
```

In most cases, we aren't interested in the numeric value. Instead, we just need to represent a single character easily. Here's an example of an instruction that uses a self-defining term:

```
CLI    CUSTID,C'A'
```

In this example, the numeric value of the character "A" is stored inside the immediate instruction – hence the name immediate – the data for operand 2 immediately follows the op-code in the object code for the instruction.

Because ampersands and apostrophes have special uses, we must code them twice whenever we use them in a self-defining term:

```
C''''     A single apostrophe
C'&&'     A single ampersand
```

Each *decimal* self-defining term is just an unsigned string of digits with a value from 0 to 2147483647 (the maximum integer represented with 31 binary digits). Some examples are 0, 2, and 2087.

Each *hexadecimal* self-defining term begins with the letter "X" and is followed by an apostrophe (') and up to eight hexadecimal digits, and a terminating apostrophe ('). If fewer than eight digits are used, the missing high-order digits are assumed to be 0's. Here are some examples and their numeric values:

```
X'C1'     193
X'FF'     255
X'F'       15
```

Each *binary* self-defining term begins with the letter "B" and is followed by an apostrophe (') and a string of 1's and 0's, and a terminating apostrophe ('). At most 32 digits are significant. If fewer than 32 digits are used, the missing high-order digits are assumed to be 0's. Here are some examples and their numeric values:

```
B'11000001'     193
B'11111111'     255
B'1111'          15
```

As you can see from the examples, a given number can have many different self-defining terms that represent it. The choice of representation is usually determined by context within a program.

Symbols and Attributes

The symbols we create to name things in our programs (like fields, files, or locations) are called **internal symbols**. As in any language, there are a few rules governing the creation of these symbols:

1) Symbols must begin with a single letter, which is optionally followed by a sequence of letters or digits.

2) Letters include the characters A-Z, a-z, and $ _ # or @.

3) Symbols may contain a maximum of 63 characters.

4) HLASM doesn't distinguish between lower and uppercase letters. "DOG", "dog", and "Dog" represent the same symbols.

Whenever a symbol is created, the assembler assigns the symbol a collection of attributes, including value, relocation, length, type, scale, and integer. The only two attributes we consider for the moment are **value** and **length**.

The *value* of a symbol is the contents of the location counter when the symbol is processed.

When assembling a program, the assembler starts the location counter at 0 by default. As it defines instructions or fields, the location counter typically advances. In all cases, the location counter is used to determine the location of the components in your program. Essentially, the value of a symbol is its location relative to the beginning of your program. We have already discussed the idea of base/displacement addressing, so you should understand how the value of a variable is important for creating addresses.

The *length* of a symbol is generally the number of bytes in the item associated with the symbol.

We can refer to the length attribute of a symbol by coding L'in front of the symbol. Here is an example:

```
OUT      DS    CL80
IN       DS    CL80
NAME     DS    CL5'ABCDE'
         ...
         MVC   OUT(L'NAME),IN
```

The length attribute allows us to explicitly override `OUT`'s implicit length (80) with the length attribute value of `NAME` (5).

We will see that length attributes are important, and there are many clever ways to use them to build programs that can self-adjust when field sizes are changed. In the example above, if the length of `NAME` is changed to ten, ten bytes will be moved by the `MVC` as a result of reassembling the code.

A Brief Introduction to Branching and Jumping

Programs have to make decisions and take actions. In assembler, when processing character data, this is a two-step process:

1) Compare two character fields. The comparison is accomplished with either CLC or CLI instructions. The comparison sets the condition code to indicate how operand 1 compares to operand 2.

2) Test the condition code value and jump (branch) to another part of the program (or not) based on the condition code.

Here is an example. Assume the following fields have been defined:

```
FIELDA      DS    CL3'CAT'
FIELDB      DS    CL3'DOG'
```

We execute the following code:

```
            CLC   FIELDA,FIELDB   SET THE CC
            JL    ALOW            JUMP IF OP1 < OP2
            ...                   OTHERWISE, OP1 >= OP2 ... FALL THROUGH

ALOW        DS   0H
```

The `CLC` instruction compares `FIELDA` and `FIELDB`. Since the first bytes of these fields differ, the condition code is set immediately to "low" after comparing the first bytes, since "C" is lower than "D" in the EBCDIC encoding sequence. (The condition code is always set to indicate how operand 1 compares to operand 2. Next, the JL (Jump Low) mnemonic instruction (in machine code this is supported by a `BRC` instruction) interrogates the condition code, and jumps on a "low" condition to the target address (`ALOW`). If the condition code had been set to "equal" or "high", the `JL` would not have been taken. Instead, control would have continued with the next instruction beneath the Jump.

CLC and CLI can only set the condition code to "equal", "low", or "high". Because of this we need only seven branch instructions when working with character data:

```
JE   -  Jump Equal
JL   -  Jump Low
JH   -  Jump High
JNE  -  Jump Not Equal
JNL  -  Jump Not Low
JNH  -  Jump Not High
J    -  Jump Unconditionally
```

Instead of using Jump instructions we could use Branch instructions instead. The difference in these instructions is in how the jump/branch is accomplished. Branch instructions use base/displacement addresses for the targets, while Jumps use relative addresses. Jumping is a newer and more flexible technique, so I recommend using Jumps instead of branches.

```
BE   -  Jump Equal
BL   -  Jump Low
BH   -  Jump High
BNE  -  Jump Not Equal
BNL  -  Jump Not Low
BNH  -  Jump Not High
B    -  Jump Unconditionally
```

When specifying a target address, there are two techniques:

1) Use an Equate name,

```
HERE     EQU    *
```

2) Use a DS label,

```
HERE     DS    0H
```

Being an old dog, I tend to favor the first technique. Many programmers argue that the second technique, which employs a halfword (H), guarantees that the target address will never be odd. This is true. In most cases you won't notice a difference – I never have. I did run into an interactive debugger that could not handle Equates as targets of branches. If you are just starting out, use the second technique. Some of my older code still employs the first technique without apologies.

As an illustration of the ideas above, lets solve the following problem,

Given the three field definitions below, write the code that will compare `FIELDA` with `FIELDB` and copy the "smaller" value to `FIELDC`. If the `FIELDA` and `FIELDB` contain the same value, copy either value to `FIELDC`. The code should work for any values you use to initialize `FIELDA` and `FIELDB`.

```
FIELDA   DC    CL3'AAA'
FIELDB   DC    CL3'ABC'
FIELDC   DC    CL3' '
```

Here is an initial solution.

```
         CLC  FIELDA,FIELDB   SET CC
         JL   ALOW            JUMP IF FIELDA IS LOWER
         MVC  FIELDC,FIELDB   FIELDB IS LOWER OR EQUAL
         J    DONE            DON'T FALL INTO NEXT CASE
ALOW     DS  0H
         MVC  FIELDC,FIELDA
DONE     DS  0H
```

A tighter solution assumes that FIELDA is lower and then tests to see if that was correct.

```
                MVC  FIELDC,FIELDA   ASSUME FIELDA IS LOWEST
                CLC  FIELDC,FIELDB   SET CC
                JNH  DONE            FIELDC IS LOWER OR EQUAL SO DONE
                MVC  FIELDC,FIELDB   FIELDC WAS HIGHER SO CHANGE
DONE            DS  0H
```

Programming Exercise

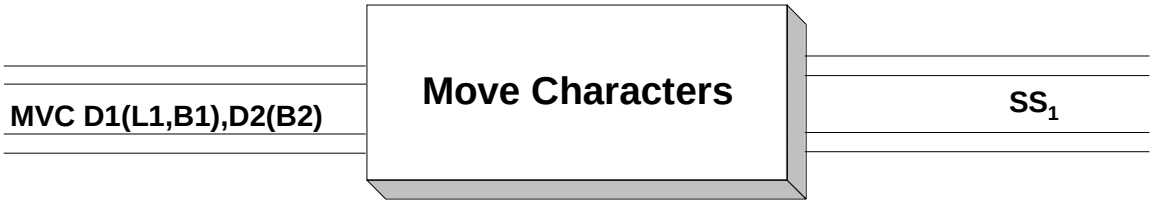
Build the three fields above into a program along with the first code that compares FIELDA and FIELDB and moves the "smaller" value to FIELDC. Print all three fields on a single line. Print out a second line by using the second comparison technique. Try changing the values in `FIELDA` and `FIELDB` to verify that both techniques work. How would you solve the problem for comparing three fields instead of two (moving the smallest value to a fourth field)?

In the rest of this chapter, we will look at a collection of individual instructions in some detail. The first step is to learn how each instruction works on its own. Character data processing is fairly straightforward and simple to work with. In future chapters, we will also examine how instructions are used in concert to accomplish certain goals.

Pay attention to the instruction types. The two most important types for this chapter are SS1 and SI. Instruction types can provide general guidelines for how individual instructions work. For example, SS1 instructions can process a maximum of 256 bytes and the number of bytes that are processed is determined by the length of operand 1. In this chapter, those ideas apply to both `MVC` and `CLC` instructions. SI instructions work with a single byte and contain a single byte of data. Those ideas apply to both MVI and CLI instructions.

The branching instructions include BC, BRC, and BRCL, although at first, we won't code these directly. Instead we will use branch or jump mnemonics that use these three instructions underneath.

The final instructions we consider are MVZ and MVN. These two "legacy" instructions are included for completeness. We are unlikely to code these in this course.



Op Code	LL_1	B_1D_1	D_1D_1	B_2D_2	D_2D_2

MVC is an instruction designed to copy a collection of consecutive bytes from one storage location to another. As you can see from the instruction format above, the instruction contains the number of bytes to be copied and the beginning addresses of the source and target fields. Notice that the instruction does not specify the ending addresses of either field - the instruction is no respecter of fields. **MVC** copies LL1 + 1 consecutive bytes from the storage location designated by B2D2D2D2 to the storage location designated by B1D1D1D1. The copying occurs one byte at a time from Operand 2 to Operand 1, and within each operand, from lower to higher numbered addresses. Fields may overlap, complicating the move operation.

The length byte (LL1) determines the number of bytes that will be copied and is usually determined implicitly from the length of operand 1, but the implicit length can be overridden by coding an explicit length. Consider the two example **MVC'** s below,

Object code Assembler code

```
                FIELDA      DS  CL8
                FIELDB      DS  CL5
                            ...
D207C008C010                MVC FIELDA,FIELDB       Implicit length
D202C008C010                MVC FIELDA(3),FIELDB    Explicit length
```

In the first example, the length implicitly defaults to 8, the length of FIELDA. In the second example, the length is explicitly 3. Notice that the assembled length (LL1) is one less than the implicit or explicit length. This can be seen in the object code above, where the assembled lengths are x'07' and x'02'.

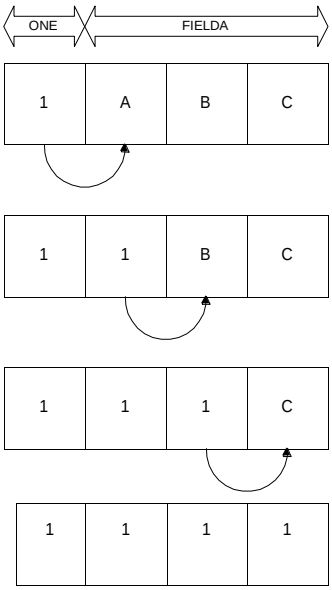
The copying operation is usually straightforward, but can be complicated by overlapping the source and target fields. Keep in mind that the copy is made one byte at a time.

Consider the following examples,

```
Object code    Assembler code

               ONE       DC     C'1'
               FIELDA    DC     CL3'ABC'
               FIELDB    DC     CL3'DEF'
               FIELDC    DC     CL4'1234'

D202C008C00B             MVC  FIELDA,FIELDB  After FIELDA = 'DEF'
D201C00EC008             MVC  FIELDC,FIELDA  After FIELDC = 'ABCD'
D201C008C007             MVC  FIELDA,ONE     After FIELDA = '111'
```

In the first **MVC** above, three consecutive bytes in FIELDB are simply copied to FIELDA. In the second example, four consecutive bytes are copied into FIELDC (implicit length = 4) from FIELDA. Since FIELDA was only three bytes long, the fourth byte was copied from the first byte of the next field - FIELDB. The third **MVC** is complicated by the fact that the source and target fields overlap. We will examine the third move in some detail.

MVC FIELDA,ONE THIS IS A 3-BYTE MOVE



First byte of source copied to first byte of target.

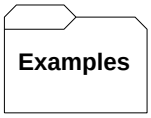
Second byte of source copied to second byte of target.

Third byte of source copied to third byte of target.

The example above depends heavily on the fact that the source and target fields overlap and that bytes are copied one at a time. In fact, it is common to use this technique to clear fields. Assume you have a buffer you would like to fill with spaces. By defining a single

blank directly in front of the field you want to clear and moving the blank field to the buffer, the blank can be propagated throughout the buffer:

```
          MVC   BUFFER,BLANK   BLANK IS PROPAGATED
          ...
BLANK     DC    C' '
BUFFER    DS    CL80
```



Some Unrelated MVC's:

```
A         DC    C'123'
B         DC    C'ABCD'
C         DC    C'PQ'

          ...          Result:
          MVC   A,B          A = 'ABC' B = 'ABCD'
          MVC   A+1,B        A = '1AB' B = 'CBCD'
          MVC   A+1(2),B     A = '1AB' B = 'ABCD'
          MVC   B,=C'XY'     B = 'XY??'  Two bytes copied from
                                         the literal pool, two
                                         unknown bytes are copied
          MVC   B,B+1        B = 'BCDP'  Left shift
          MVC   B+1,B        B = 'AAAA'  First byte is propagated
                             C = 'AQ'    Four bytes moved
          MVC   C,A          C = '12' A = '123' Two bytes are copied
          MVC   A(L'C),C     A = 'PQ3' Explicit Length attribute
          MVC   A(1000),B    Assembly Error - max length is 256 bytes
          MVC   A,B(20)      Assembly Error - Op-1 determines length
```

Tips

1. Pay attention to the lengths of the fields involved in any MVC statement. If the target field is longer than the source field, bytes following the source may be transferred. If the target field is shorter than the source field, bytes from the source may be truncated.

2. Be careful when using literals in an MVC since stray bytes in the literal pool will be moved if the specified length of operand 1 is longer than the operand 2 literal. Lengths can also be specified in a literal (=CL133' '). The following typical error, MVC BUFFER,=C' ', can be repaired as MVC BUFFER,=CL80' '. This assumes the length of BUFFER is 80.

Trying It Out in VisibleZ:

1. Load the program **mvc.obj** from the ＼Codes directory and single-step through each instruction until you are about to execute the first MVC instruction.

2. What are the contents of register 12?

3. In object code, what is the base/displacement address of the target operand? What is the effective address of the target? Is this address highlighted in red?

4. In object code, what is the base/displacement address of the source operand? What is the effective address of the source? Is this address highlighted in green?

5. In object code, what is the length that is associated with this MVC? Exactly how many bytes will the instruction move?

6. Continue to step through the program and answer questions 3 and 4 for each MVC.

Op Code	II_2	B_1D_1	D_1D_1

MVI is used to move (copy) a one-byte immediate constant to a field in storage. Operand 1 denotes the field in main storage, while the second operand is coded as a **self-defining term** that gets assembled as a one-byte immediate constant (II_2) in the second byte of the object code. Only the first byte of Operand 1 is affected by the move.
As an example, consider the following code,

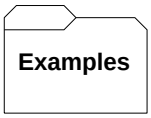
```
            MVI   FIELDA,X'C1'
            ...
FIELDA      DC    X'123456'
```

After execution, FIELDA contains X'C13456'. The immediate instruction alters only the first byte of the field.

The following example illustrates how the assembler might process an **MVI** instruction.

```
LOC          OBJECT CODE

000F12       92F4C044                MVI   CUSTCODE,C'4'
                                     ...
001028                    CUSTCODE   DS    CL1
```

In the example above, the op-code for **MVI** is x'92'. The self-defining term C'4' is assembled as the one-byte hexadecimal constant x'F4', and CUSTCODE is translated into the base/displacement address C044.



Some Unrelated MVI's:

```
J           DC   C'ABC'
            MVI  J,C'X'       J = C'XBC'
            MVI  J,C'B'       J = C'BBC'
            MVI  J,C'5'       J = C'5BC'
            MVI  J,X'F5'      J = C'5BC'
            MVI  J,197        J = C'EBC'
            MVI  J,=C'5'      ERROR-OPERAND 2 NOT A SELF-DEFINING TERM
            MVI  J(1),X'C5'   ASSEMBLY ERROR - LENGTH SPECIFICATION NOT
                              ALLOWED IN OPERAND 1
```

Trying It Out in VisibleZ:

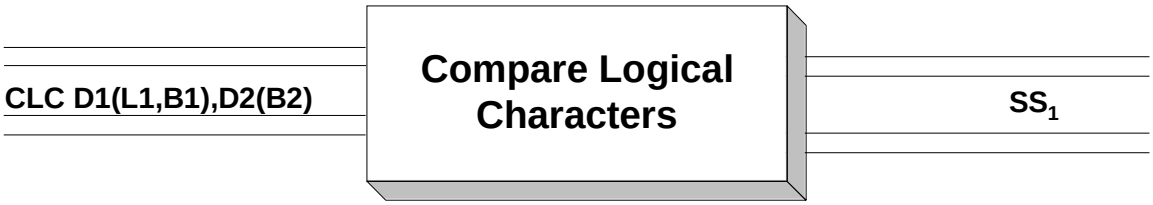
1) Load the program **mvi.obj** from the \Codes directory and single-step through each instruction until you are about to execute the first `MVI` instruction.

2) What are the contents of register 12?

3) VisibleZ highlights the current instruction in yellow, the first source byte in green, and the first target byte in red. Why in this `MVI` does a green byte appear inside the current instruction?

4) In object code, what is the base/displacement address of the target operand? What is the effective address of the target? Is this address highlighted in red?

5) Continue to step through the program and answer question 4 for each `MVI`.



Op Code	LL_1	B_1D_1	D_1D_1	B_2D_2	D_2D_2

CLC is used to compare two fields that are both in storage. The fields are compared, one byte at a time beginning with the bytes specified in addresses B1D1D1D1 and B2D2D2D2 and moving to higher addresses in the source and target fields. Each byte in the source is compared to a byte in the target according to the ordering specified in the EBCDIC encoding sequence. Executing a compare instruction sets the condition code (a two-bit field in the PSW) to indicate how operand 1 (target field) compares with operand 2 (source field). The condition code is set as follows,

Comparison	Condition Code	Value	Test With
Operand 1 = Operand 2	0	Equal	BE (Branch Equal) BNE (Branch Not Equal)
Operand 1 < Operand 2	1	Low	BL (Branch Low) BNL (Branch Not Low)
Operand 1 > Operand 2	2	High	BH (Branch High) BNH (Branch Not High)

The above table also indicates the appropriate branch instructions for testing the condition code. When comparing two fields, a **CLC** instruction should be followed immediately by one or more branch instructions for testing the contents of the condition code:

```
CLC      FIELDA,FIELDB     SET THE CONDITION CODE
BH       AHIGH             BRANCH IF FIELDA IS HIGH
BL       BHIGH             BRANCH IF FIELDA IS LOW
```

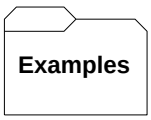
Bytes are compared one at a time until the number of bytes specified (implicitly or explicitly) in operand 1 have been exhausted or until two unequal bytes are found - whichever occurs first. As you can see from the instruction format above, the instruction carries with it the maximum number of bytes to be compared and the beginning addresses of the source and target fields. Notice that the instruction does not specify the ending addresses of either field - the instruction is no respecter of fields. If a longer field is compared to a shorter field, the bytes following the shorter field may be used in the comparison operation.

The length (LL1) determines the maximum number of bytes to compare. The length is usually determined implicitly from the length of operand 1, but the programmer can

provide an explicit length. Consider the two example **CLC**'s below,

```
Object code Assembler code
               FIELDA   DC   CL4'ABCD'
               FIELDB   DC   C'ABE'
               ...
D502C00CC008           CLC  FIELDB,FIELDA     Implicit length = 3,
                                              COND CODE = HIGH SINCE E>C
D501C00CC008           CLC  FIELDB(2),FIELDA  Explicit length = 2,
                                              COND CODE = EQUAL SINCE AB = AB
```

In the first **CLC**, 'A' in FIELDB is compared with 'A' in FIELDA, then 'B' in FIELDB is compared with 'B' in FIELDA, finally, 'E' in FIELDB is compared with 'C' in FIELDA. At this point, the condition code is set to 'HIGH' since 'E' follows 'C' in the EBCDIC encoding sequence. In the second example, 'A' in FIELDB is compared with 'A' in FIELDA, then 'B' in FIELDB is compared with 'B' in FIELDA. The condition code is set to 'EQUAL' since an explicit length of 2 was coded and A=A, B=B.



Some Unrelated CLC's:

```
A        DC    C'PQR'
B        DC    C'ABCD'
C        DC    C'PQ'
D        DC    P'12'          D = X'012C'
```

COMPARISON **RESULT**

```
CLC   A,B         Condition Code = High, one byte compared.
CLC   A(2),C      Condition Code = Equal, two bytes compared.
CLC   C,A         Condition Code = Equal, two bytes compared.
CLC   A,=C' '     Condition Code = High, one byte compared.
                  (Usually not a good idea to compare against Literals.
                   In this example, we may use Unknown bytes in the literal
                   pool because the Fields have different sizes)
CLC   A,=CL3' '   Condition Code = High
                  (This is a better version of the previous comparison. Try to
                   avoid hard-coded lengths. What happens if the length of A
                   is increased?)
CLC   B,=X'C1C2C3C4'  Condition Code = Equal,4 bytes are compared.
CLC   D,=P'12'    This is a dangerous comparison of packed data (Use CP)
CLC   A(500),B    Assembly Error - max length is 256
CLC   A,B(20)     Assembly Error – op 1 determines length
```

Tips

1. As with any storage and storage instruction, you must pay careful attention to the lengths of the two operands. Generally, you should be comparing fields that are the same size.

2. The instruction was designed to compare fields that are in character format. It can be used to compare fields with non-character data, but this takes special consideration to ensure the comparison will produce the desired results. Packed decimal data and binary data are supported by their own special comparison instructions.

3. The condition code can be changed by any other type of comparison instruction and a variety of arithmetic instructions. Don't rely on the condition code to remain set - after you have issued a **CLC**, you should follow it immediately with a branch instruction.

Trying It Out in VisibleZ:

1. Load the program **clc.obj** from the \Codes directory and single-step through each instruction until you are about to execute the first CLC instruction.

2. How many bytes will be compared?

3. What are the hexadecimal contents of the fields that are examined by this CLC?

4. What is the condition code after executing the CLC? Why?

Op Code	I_2	B_1D_1	D_1D_1

CLI is used to compare two fields that are both in storage. Operand 1 is a field in main storage, while the second operand is a **self-defining term** that gets assembled as a one-byte immediate constant (I_2) in the second byte of the object code. Only the first byte of Operand 1 is compared to the immediate constant. The comparison is made based on the ordering of characters in the EBCDIC encoding.

Executing a compare instruction sets the condition code (a two-bit field in the PSW) to indicate how operand 1 (target field) compares with operand 2 (immediate constant). The condition code is set as follows,

Comparison	Condition Code	Value	Test With
Operand 1 = Operand 2	0	Equal	BE (Branch Equal) BNE (Branch Not Equal)
Operand 1 < Operand 2	1	Low	BL (Branch Low) BNL (Branch Not Low)
Operand 1 > Operand 2	2	High	BH (Branch High) BNH (Branch Not High)

The table above also illustrates the appropriate branch instructions for testing the condition code.

When comparing two fields, a **CLI** instruction should be followed immediately by one or more branch instructions for testing the contents of the condition code:

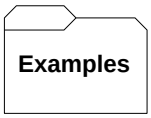
```
FIELDA   DC    C'1234'
         ...
         CLI   FIELDA,C'A'
         BL    ALOW            BRANCH IF FIELDA IS LOW
         BH    AHIGH           BRANCH IF FIELDA IS HIGH
```

In the example above, the first byte of FIELDA, which contains the character "1" and is represented as x'F1', is compared to the self-defining term C'A', which assembles as x'C1'. In EBCDIC, since x'F1' is greater than x'C1', the condition code is set to "high" to indicate that operand 1 is "higher" than operand 2. The BL branch is not taken, but the second branch will be taken to the symbol AHIGH. If the first byte of FIELDA had contained C'A', the condition code would have been set to "equal" by the CLI. Neither branch would have occurred, and execution would have continued with the next statement.

The following example illustrates how the assembler might process a CLI.

```
LOC          OBJECT CODE
000F12       95F4C044                CLI  CUSTCODE,C'4'
                                     ...
001028                       CUSTCODE DS   CL1
```

In the example above, the op-code for CLI is x'95'. The self-defining term C'4' is assembled as the one-byte hexadecimal constant x'F4', and CUSTCODE is translated into the base/displacement address C044.



Some Unrelated CLI's:

```
J        DC   C'ABC'
K        DC   C'DEF'
L        DC   C'GH'
M        DC   C'12345'
                                                 Result:
         CLI  J,C'A'     Condition Code = Equal, one byte compared.
         CLI  J,C'B'     Condition Code = Low.
         CLI  J,C'5'     Condition Code = Low, letters < numbers.
         CLI  K,X'C4'    Condition Code = Equal.
         CLI  L,C'A'     Condition Code = High.
         CLI  L,=C'G'    Assembly error, Literals not allowed.
         CLI  B,X'C1C2'  Assembly error, 1-byte comparisons only.
         CLI  C'A',M     Assembly error, operands out of order.
         CLI  A(20),B    Assembly Error, 1-byte comparisons only.
         CLC  A,B(20)    Operand 2 must be a self-defining constant
```

Tips

1. Use `CLI` instead of `CLC` when comparing 1-byte fields. The resulting code is smaller and slightly more efficient. More importantly, it makes explicit the fact that you are comparing two 1-byte fields.

2. Use an equate name for the second operand in a CLI to enhance readability:

```
CUSTTYPE     DS    CL1
ACUST        EQU   C'A'
BCUST        EQU   C'B'
             ...
             CLI   CUSTTYPE,ACUST
             BE    PROCESSA
             CLI   CUSTTYPE,BCUST
             BE    PROCESSB
             ...
```

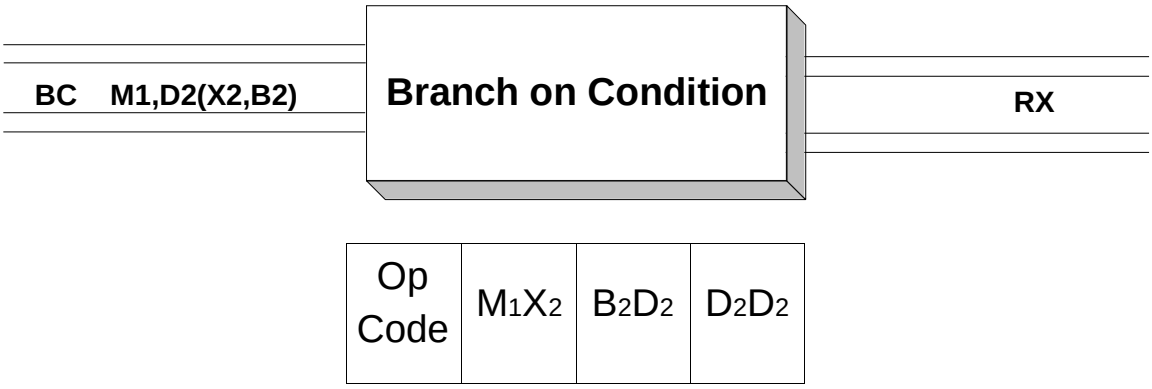
Trying It Out in VisibleZ:

1) Load the program **cli.obj** from the `\Codes` directory and single-step through each instruction until you are about to execute the CLI instruction.

2) How many bytes will be compared?

3) What are the hexadecimal contents of the two fields that are compared?

4) What is the condition code after executing this instruction? Why?



The Branch on Condition instruction(BC) examines the 2-bit condition code in the PSW and branches (or not) based on the value it finds. Operand 1 is a self-defining term representing a 4-bit mask (binary pattern) indicating the conditions under which the branch should occur. Operand 2 is the target address to which the branch will be made if the condition indicated in Operand 1 occurs. If the condition code has one of the values specified in the mask, the instruction address in the PSW is replaced with the instruction's target address. This causes the processor to fetch the instruction located at the target address as the next instruction in the fetch/execute cycle.

There are four possible values for the condition code:

```
Condition Code  Meaning

      00          Zero or Equal

      01          Low or Minus

      10          High or Plus

      11          Overflow
```

When constructing a mask for Operand 1, each bit (moving from the high-order bit to the low-order bit) represents one of the four conditions in the following order: Zero/Equal, Low/Minus, High/Plus, Overflow. Consider the next instruction,

```
        BC  8,THERE
```

The first operand,"8", is a decimal self-defining term representing the binary mask B'1000'. Since the first bit is a 1, the mask indicates that a branch should occur on a zero or equal condition. Since the other bits are all 0, no branch will be taken on the other conditions. The first operand could be designated as any equivalent self-defining term. For example, the following instruction is equivalent to the one above.

```
        BC  B'1000',THERE
```

Extended mnemonics were developed to replace the awkward construction of having to code a mask. The extended mnemonics are easier to code and read. A listing of the extended mnemonics follows below.

```
BE    Branch Equal      BNE Branch Not Equal
BZ    Branch Zero       BNZ Branch Not Zero
BL    Branch Low        BNL Branch Not Low
BM    Branch Minus      BNM Branch Not Minus
BH    Branch High       BNH Branch Not High
BP    Branch Positive   BNP Branch Not Positive
NOP   No Operation      B   Unconditional Branch
```

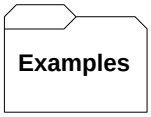
Using extended mnemonics, we could replace the previous Branch On Condition instruction with the following,

```
BZ  THERE
```

The "BZ" means "Branch on Condition Zero". When the assembler processes **BZ**, it generates the mask as B'1000'. The table below indicates the possible mask values and the equivalent extended mnemonics.

Eq/Zero	Low/Min	High/Plus	Overflow	Decimal Condition	Extended Mnemonic
0	0	0	0	0	NOP
0	0	0	1	1	BO
0	0	1	0	2	BH, BP
0	0	1	1	3	NO MNEMONIC
0	1	0	0	4	BL,BM
0	1	0	1	5	NO MNEMONIC
0	1	1	0	6	NO MNEMONIC
0	1	1	1	7	BNE,BNZ
1	0	0	0	8	BE,BZ
1	0	0	1	9	NO MNEMONIC
1	0	1	0	10	NO MNEMONIC
1	0	1	1	11	BNL,BNM
1	1	0	0	12	NO MNEMONIC

1	1	0	1	13	BNH,BNP
1	1	1	0	14	NO MNEMONIC
1	1	1	1	15	B



Some Unrelated Branch on Conditions

```
         CLC   X,Y       SET THE CONDITION CODE
         BP    THERE     BRANCH IF CONDITION CODE IS POSITIVE
         ...             OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
THERE    EQU   *

         CLC   X,Y       SET THE CONDITION CODE
         BE    THERE     BRANCH IF X = Y
         ...             OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
THERE    EQU   *

         CLC   X,Y       SET THE CONDITION CODE
         BH    THERE     BRANCH IF X > Y
         ...             OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
THERE    EQU   *
```

Trying It Out in VisibleZ:

1) Load the program **bc.obj** from the \Codes directory and single-step through each instruction until you are about to execute the first CLC instruction.

2) Assume both source and target fields for the CLC are four bytes in length. How do the fields compare?

3) Step through the CLC. What is the condition code after executing the comparison?

4) The next instruction is a Branch on Condition. What is the condition under which the instruction will branch? What happens when you step through the BC?

5) Why does executing the third BC instruction return you to the first BC instruction?

Op Code	M_1 4	I_2I_2	I_2I_2

The Branch Relative on Condition instruction (BRC) examines the 2-bit condition code in the PSW and branches (or not) based on the value it finds. Operand 1 is a self-defining term representing a 4- bit mask M_1 (binary pattern) indicating the conditions under which the branch should occur.

Operand 2 is the target address to which the branch will be made if the condition indicated in Operand 1 occurs. Rather than representing the target as a base/displacement address, the number of halfwords from the current instruction to the target address is computed as a two's complement integer and stored in the instruction as I_2.

BRC works exactly like a BC except for the mechanism of specifying the target address –this is the address that replaces the current PSW instruction address field. In the case of BRC, the target address is computed by doubling the two's complement integer represented in I_2 and adding the result to the address of the current instruction (not the base register). If the condition code has one of the values specified in the mask, the PSW's instruction address is replaced with this "relative" address. Since I_2 is a 16-bit two's complement integer representing a number of halfwords, this instruction can be used to branch forward $(2^{15} - 1) \times 2 = 65534$ bytes and backward $2^{15} \times 2 = 65536$ bytes from the current instruction.

There are four possible values for the condition code:

```
Condition Code    Meaning

00                Zero or Equal
01                Low or Minus
10                High or Plus
11                Overflow
```

When constructing a mask for Operand 1, each bit (moving from the high-order bit to the low- order bit) represents one of the four conditions in the following order: Zero/Equal, Low/Minus, High/Plus, Overflow. Consider the next instruction,

```
BRC  2,THERE
```

The first operand,"2", is a decimal self-defining term representing the binary mask B'0010'. Since the third bit is a 1, the mask indicates that a branch should occur on a high

or plus condition. Since the other bits are all 0, no branch will be taken on the other conditions. The first operand could be designated as any equivalent self-defining term. For example, the following instruction is equivalent to the one above.

```
BRC  B'0010',THERE
```

When relative branching was introduced, new sets of extended mnemonics were also developed to replace the awkward construction of having to code a mask. The extended mnemonics are converted to BRC's and are easier to code and read than using masks. Here is a list of the branch relative extended mnemonics and the equivalent jump extended mnemonics.

Equal/Zero	Low/Minus	High/Plus	Overflow	Decimal Condition	Extended Mnemonic
0	0	0	0	0	JNOP
0	0	0	1	1	BRO,JO
0	0	1	0	2	BRH,JH
0	0	1	1	3	NO MNEMONIC
0	1	0	0	4	BRL,JL BRM,JM
0	1	0	1	5	NO MNEMONIC
0	1	1	0	6	NO MNEMONIC
0	1	1	1	7	BRNE,JNE BNZ,JNZ
1	0	0	0	8	BRE,JE BRZ,JZ
1	0	0	1	9	NO MNEMONIC
1	0	1	0	10	NO MNEMONIC
1	0	1	1	11	BRNL,JNL BRNM,JNM
1	1	0	0	12	NO MNEMONIC
1	1	0	1	13	BRNH,JNH BRNP,JNP
1	1	1	0	14	NO MNEMONIC
1	1	1	1	15	BRU,J

Notice that branch relative mnemonics have equivalent jump mnemonics. Simply replacing "BR" with "J" produces the equivalent mnemonic in most cases. NOPs and unconditional branches are the exceptions.

Using extended mnemonics, we could replace the previous Branch Relative On Condition instruction (`BRC B'0010',THERE`) with any of the following instructions,

```
BRH   THERE
JH    THERE
BRP   THERE
JP    THERE
```

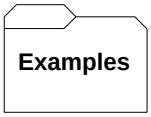
When the assembler processes **BRH, JH**, **BRP**, or **JP,** it generates the mask as B'0010'. The table below indicates the possible mask values and the equivalent extended mnemonics.

Tips

1) BRC and the extended mnemonics that generate BRC's represent a distinct improvement over BC's. Branch instructions specify their target addresses using base/displacement format, so for lengthy programs, several base registers may be needed to cover all the target addresses. With relative branches, the target address is specified as a number of halfwords from the current instruction. No base registers are needed for target addresses.

2) Use jump (J) mnemonics instead of branch relative (BR) mnemonics. Jumping is an accurate description of how these instructions operate.

3) Abandon BC's for any new code you develop – choose jumps. You can also easily replace many older branch instructions with jumps to reclaim the use of registers that were given over to base/displacement addressing. Registers are always at a premium, and saving a register by converting your code to use jumps is relatively (bad pun) easy.



Some Unrelated Branch Relative on Conditions

```
         CLC   X,Y     SET THE CONDITION CODE
         JP    HERE    JUMP IF CONDITION CODE IS POSITIVE
         ...           OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
HERE     EQU   *
         CLC   X,Y     SET THE CONDITION CODE
         JE    THERE   JUMP IF X = Y
         ...           OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
THERE    EQU   *
         CLC   X,Y     SET THE CONDITION CODE
         BRE   YON     JUMP IF X = Y (Equivalent to JE)
         ...
YON      EQU   *       OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
```

Trying It Out in VisibleZ:

6) Load the program **brc.obj** from the \Codes directory and cycle through the first instruction. The second instruction is BRC, and its immediate operand is X'FFFF'. This number is -1 in two's complement. Why is the first instruction in the program highlighted in red?

7) Load the program brc1.obj and cycle through the first three BRC instructions. What are the masks of each instruction? Do you understand why the branch was only taken on the third instruction?

8) Load the program brc2.obj. Why does the fourth BRC branch forward? Why does the fifth BRC branch backward?

Op Code	$M_1$4	I_2I_2	I_2I_2	I_2I_2	I_2I_2

The Branch Relative on Condition Long instruction (BRCL) works exactly like BRC but provides a larger two's complement integer to describe the length of the branch so that it can jump much further forward or backward than BRC. The instruction examines the 2-bit condition code in the PSW and branches (or not) based on the value it finds. Operand 1 is a self-defining term representing a 4-bit mask M_1 (binary pattern) indicating the conditions under which the branch should occur. Operand 2 is the target address to which the branch will be made if the condition indicated in Operand 1 occurs. Rather than representing the target as a base/displacement address, the number of halfwords from the current instruction to the target address is computed as a two's complement integer and stored in the instruction as I_2.

BRCL is similar to BC except for the mechanism of specifying the target address –this is the address that replaces the current PSW instruction address field. In the case of BRCL, the target address is computed by doubling the two's complement integer represented in I_2 and adding the result to the address of the current instruction (not the base register address). If the condition code has one of the values specified in the mask, the PSW's instruction address is replaced with this "relative" address. Since I_2 is a 32-bit two's complement integer that represents a number of halfwords, this instruction can be used to branch forward $(2^{32} – 1)$ x 2 (approximately 4G) and backward 2^{32} x 2 (approximately -4G).

There are four possible values for the condition code:

```
Condition Code    Meaning

00                Zero or Equal
01                Low or Minus
10                High or Plus
11                Overflow
```

When constructing a mask for Operand 1, each bit (moving from the high-order bit to the low-order bit) represents one of the four conditions in the following order: Zero/Equal, Low/Minus, High/Plus, Overflow. Consider the next instruction,

```
BRCL    4,THERE
```

The first operand,"4", is a decimal self-defining term representing the binary mask B'0100'. Since the second bit is a 1, the mask indicates that a branch should occur on a low

or minus condition. Since the other bits are all 0, no branch will be taken on the other conditions. The first operand could be designated as any equivalent self-defining term. For example, the following instruction is equivalent to the one above.

```
BRCL    B'0100',THERE
```

When relative branching was introduced, new sets of extended mnemonics were also developed to replace the awkward construction of having to code a mask. The extended mnemonics are converted to BRC's and are easier to code and read than using masks. Here is a list of the branch relative extended mnemonics and the equivalent jump extended mnemonics.

Equal/Zero	Low/Minus	High/Plus	Overflow	Decimal Condition	Extended Mnemonic
0	0	0	0	0	JLNOP
0	0	0	1	1	BROL,JLO
0	0	1	0	2	BRHL,JLH BRPL,JLP
0	0	1	1	3	NO MNEMONIC
0	1	0	0	4	BRML,JLL BRLL,JLM
0	1	0	1	5	NO MNEMONIC
0	1	1	0	6	NO MNEMONIC
0	1	1	1	7	BRNEL,JLNE BNZL,JLNZ
1	0	0	0	8	BREL,JLE BRZL,JLZ
1	0	0	1	9	NO MNEMONIC
1	0	1	0	10	NO MNEMONIC
1	0	1	1	11	BRNLL,JLNL BRNML,JLNM
1	1	0	0	12	NO MNEMONIC
1	1	0	1	13	BRNHL,JLNH BRNPL,JLNP
1	1	1	0	14	NO MNEMONIC
1	1	1	1	15	BRUL,JLU

Notice that branch relative mnemonics have equivalent jump mnemonics. Simply replacing "BR" with "J" produces the equivalent mnemonic in most cases. NOPs and unconditional branches are the exceptions.

Using extended mnemonics, we could replace the previous Branch Relative On Condition instruction (`BRC  B'0100',THERE`) with any of the following instructions,

```
BRLL  THERE
JLL   THERE
BRML  THERE
JLM   THERE
```

When the assembler processes **BRLL, JLL**, **BRML**, or **JLM,** it generates the mask as B'0100'. The table below indicates the possible mask values and the equivalent extended mnemonics.

Here is an alternate listing of the extended mnemonics for `BRCL`, followed by the equivalent Jump mnemonics.

Branch Relative on Condition Long

Mnemonic		Name	Type	Condition
BROL	label	Br Rel Long on Overflow	RIL	BRCL 1,label
BRHL	label	Br Rel Long on High	RIL	BRCL 2,label
BRPL	label	Br Rel Long on Plus	RIL	BRCL 2,label
BRLL	label	Br Rel Long on Low	RIL	BRCL 4,label
BRML	label	Br Rel Long on Minus	RIL	BRCL 4,label
BRNEL	label	Br Rel Long on Not Equal	RIL	BRCL 7,label
BRNZL	label	Br Rel Long on Not Zero	RIL	BRCL 7,label
BREL	label	Br Rel Long on Equal	RIL	BRCL 8,label
BRZL	label	Br Rel Long on Zero	RIL	BRCL 8,label
BRNLL	label	Br Rel Long on Not Low	RIL	BRCL 11,label
BRNML	label	Br Rel Long on Not Minus	RIL	BRCL 11,label
BRNHL	label	Br Rel Long on Not High	RIL	BRCL 13,label
BRNPL	label	Br Rel Long on Not Plus	RIL	BRCL 13,label
BRNOL	label	Br Rel Long on Not Overflow	RIL	BRCL 14,label
BRUL	label	Unconditional Br Rel Long	RIL	BRCL 15,label

Jump on Condition Long

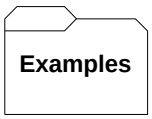
Mnemonic		Name	Type	Condition
JLNOP	label	No operation	RIL	BRCL 0,label
JLO	label	Jump Long on Overflow	RIL	BRCL 1,label
JLH	label	Jump Long on High	RIL	BRCL 2,label
JLP	label	Jump Long on Plus	RIL	BRCL 2,label
JLL	label	Jump Long on Low	RIL	BRCL 4,label
JLM	label	Jump Long on Minus	RIL	BRCL 4,label
JLNE	label	Jump Long on Not Equal	RIL	BRCL 7,label
JLNZ	label	Jump Long on Not Zero	RIL	BRCL 7,label
JLE	label	Jump Long on Equal	RIL	BRCL 8,label
JLZ	label	Jump Long on Zero	RIL	BRCL 8,label
JLNL	label	Jump Long on Not Low	RIL	BRCL 11,label
JLNM	label	Jump Long on Not Minus	RIL	BRCL 11,label
JLNH	label	Jump Long on Not High	RIL	BRCL 13,label
JLNP	label	Jump Long on Not Plus	RIL	BRCL 13,label
JLNO	label	Jump Long on Not Overflow	RIL	BRCL 14,label
JLU	label	Unconditional Jump Long	RIL	BRCL 15,label

Tips

1) BRCL and the extended mnemonics that generate BRCL's represent a distinct improvement in some respects over BC's. Branch instructions specify their target addresses using base/displacement format, so for long programs, several base registers may be needed to cover all the target addresses. With relative branches, the target address is specified as a number of halfwords from the current instruction. No base registers are needed for target addresses.

2) Use jump (JL) mnemonics instead of branch relative (BRL) mnemonics. Jumping is an accurate description of how these instructions operate.

3) Abandon BC's for any new code you develop – choose jumps. You can also easily replace many older branch instructions with jumps to reclaim the use of registers that were given over to base/displacement addressing. Registers are always at a premium, and saving a register by converting your code to use jumps is a reasonable choice.



Some Unrelated Branch Relative on Condition Longs

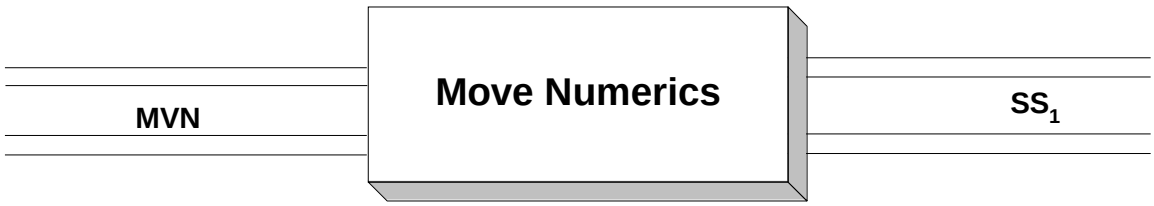
```
         LTR   R8,R8      SET THE CONDITION CODE
         JLP   HERE       JUMP IF CONDITION CODE IS POSITIVE
         ...              OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
HERE     EQU   *
         CLC   X,Y        SET THE CONDITION CODE
         JLH   THERE      JUMP IF X > Y
         ...              OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
THERE    EQU   *
         CLC   X,Y        SET THE CONDITION CODE
         BREL  YON        JUMP IF X = Y (EQUIVALENT TO JLE)
         ...              OTHERWISE FALL THROUGH TO NEXT INSTRUCTION
YON      EQU   *
```

Trying It Out in VisibleZ:

1. Load the program **brcl.obj** from the `\Codes` directory. The first instruction is BASR, which sets up addressability in R12. The second instruction, `CR`, sets the condition code to Equal/Zero. This is followed by a load that initializes R5 with a binary fullword. Finally, we encounter `BRCL`. What is the mask? What is the condition code? Why is the color of byte 14 light red?

2. Since the mask was zero, a branch is not taken. The next instruction is another BRCL. Why is byte 0 light red?

3. Load the program **brcl1.obj**. Why does the first **BRCL** fall through? Why does the second **BRCL** take the branch?



Op Code	LL_1	B_1D_1	D_1D_1	B_2D_2	D_2D_2

MVN performs in a way that is analogous to **MVC**. While **MVC** works on entire bytes, **MVN** only processes the numeric parts (rightmost 4 bits) of the bytes it references. The purpose of a move numeric instruction is to move the numeric parts of a consecutive collection of bytes from one location in memory to another location. As you can see from the instruction format above, the instruction carries with it the number of bytes to be copied (LL_1), as well as the beginning addresses of the source ($B_2D_2D_2D_2$) and target ($B_1D_1D_1D_1$) fields. Notice that the instruction does not specify the ending addresses of either field - the instruction is no respecter of fields. **MVN** copies the numeric parts of $LL_1 + 1$ consecutive bytes from the storage location designated by $B_2D_2D_2D_2$ to the storage location designated by $B_1D_1D_1D_1$.

The length (LL_1) determines the number of "half-bytes" which will be copied. The length is usually determined implicitly from the length of operand 1, but the programmer can provide an explicit length. Consider the two example MVN's below,

```
      Object code Assembler code
                FIELDA       DS  CL8
                FIELDB       DS  CL5
                             ...
  D107C008C010               MVN FIELDA,FIELDB       Implicit length
  D102C008C010               MVN FIELDA(3),FIELDB    Explicit length
```

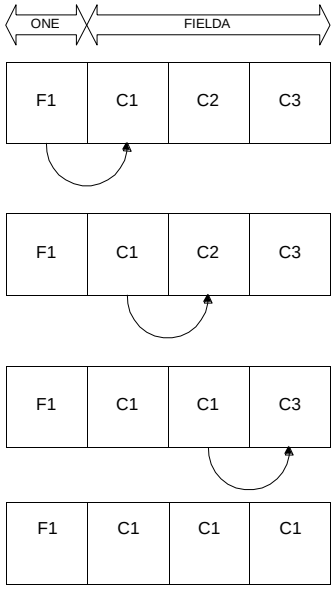
In the first example, the length implicitly defaults to 8, the length of `FIELDA`. In the second example, the length is explicitly 3. Notice that the assembled length (LL1) is one less than the implicit or explicit length. This can be seen in the object code above, where the assembled lengths are x'07' and x'02'.

The copying operation is usually straightforward, but can be complicated by overlapping the source and target fields. Keep in mind that the copy is made one byte at a time. Consider the following examples,

```
        Object code             Assembler code

                    ONE         DC   C'1'           ONE = X'F1'
                    FIELDA      DC   CL3'ABC'       FIELDA = X'C1C2C3'
                    FIELDB      DC   XL3'123456'    FIELDB = X'123456'
                                ...
D102C008C00B                    MVN FIELDA,FIELDB  After FIELDA = X'C2C4C6'
D102C00EC008                    MVN FIELDB,FIELDA  After FIELDB = X'113253'
D102C008C007                    MVN FIELDA,ONE     After FIELDA = X'C1C1C1'
```

In the first **MVN** above, three consecutive numeric half-bytes in FIELDB are simply copied to the numeric portions of FIELDA. The half-bytes are copied, one at a time, moving left to right within both operands. In the second example, three consecutive bytes are copied into FIELDB (implicit length = 3) from FIELDA. The third **MVN** is complicated by the fact that the source and target fields overlap. We will examine the third move in some detail.

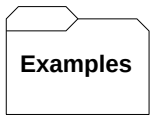
MVN FIELDA,ONE THIS IS A 3-BYTE MOVE



First half-byte of source copied to first half-byte of target.

Second half-byte of source copied to second half-byte of target.

Third half-byte of source copied to third half-byte of target.



Some Unrelated MVN's:

```
A       DC    X'123456'
B       DC    X'ABCDEF'
C       DC    X'A1B2'
        MVN   A,B          A = X'1B3D5F' B = X'ABCDEF'
        MVN   A+1,B        A = X'123B5D' B = X'AFCDEF'
        MVN   A+1(2),B     A = X'123B5D' B = X'ABCDEF'
        MVN   B,=X'D1E2'   B = X'A1C2E?' One half-byte
                                         copied from the literal pool
        MVN   B,B+1        B = 'ADCFE1' Left shift
        MVN   B+1(2),B     B = 'ABCBEB' 1st byte is propagated
        MVN   C,A          C = 'A2B4' A = X'123456'
        MVN   A(L'C),C     A = '113256' Explicit Length attribute
        MVN   A(1000),B    Assembly Error - max length is 256 for SS1
        MVN   A,B(20)      Assembly Error - Op-1 determines the length
```

 *Tips*

1. Pay attention to the lengths of the fields involved in any **MVN** statement. If the target field is longer than the source field, bytes following the source may be transferred. If the target field is shorter than the source field, bytes in the source may be truncated.

2. Consider using Shift and Round Packed, **SRP**, instead of **MVN** if you are working with packed fields.

Trying It Out in VisibleZ:

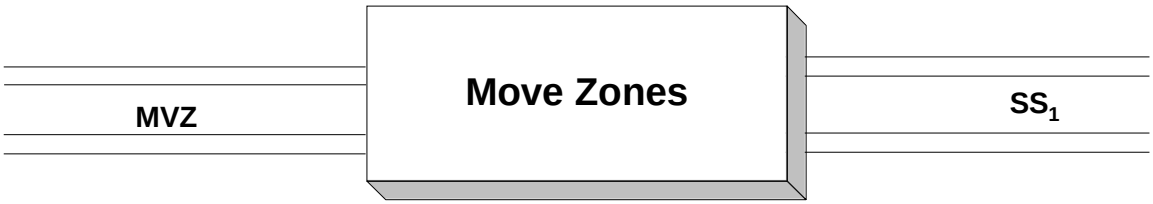
1. Load the program **mvn.obj** from the \Codes directory and single-step through each instruction until you are about to execute the MVN instruction.

2. What is the length associated with the MVN instruction?

3. What are the numeric parts of the source field?

4. What are the numeric parts of the target field?

5. Does this instruction change the zone parts of the target?



Op Code	LL_1	B_1D_1	D_1D_1	B_2D_2	D_2D_2

MVZ performs in a way that is analogous to **MVC**. While **MVC** works on entire bytes, **MVZ** only processes the zoned portions (leftmost 4 bits) of the bytes it references. The purpose of a move zones instruction is to move (copy) the zoned parts of a consecutive collection of bytes from one location in memory to another location. As you can see from the instruction format above, the instruction carries with it the number of bytes to be copied (LL_1), as well as the beginning addresses of the source ($B_2D_2D_2D_2$) and target ($B_1D_1D_1D_1$) fields. Notice that the instruction does not specify the ending addresses of either field - the instruction is no respecter of fields. **MVZ** copies the zoned parts of $LL_1 + 1$ consecutive bytes from the storage location designated by $B_2D_2D_2D_2$ to the storage location designated by $B_1D_1D_1D_1$.

The length (LL_1) determines the number of "half-bytes" which will be copied. The length is usually determined implicitly from the length of operand 1, but the programmer can provide an explicit length. Consider the two example MVZ's below,

```
Object code        Assembler code

                  FIELDA   DS    CL8
                  FIELDB   DS    CL5

D307C008C010              MVZ FIELDA,FIELDB     Implicit length
D302C008C010              MVZ FIELDA(3),FIELDB  Explicit length
```

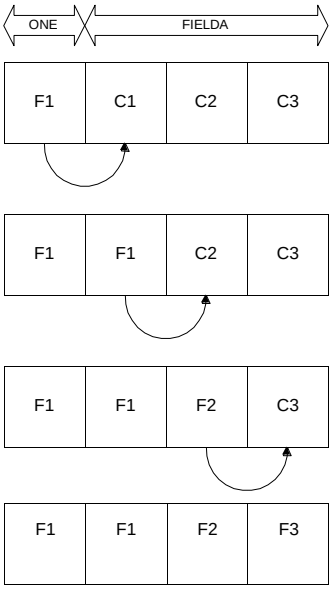
In the first example, the length implicitly defaults to eight, the length of FIELDA. In the second example, the length is explicitly three. Notice that the assembled length (LL_1) is one less than the implicit or explicit length. This can be seen in the object code above, where the assembled lengths are x'07' and x'02'.

The copying operation is usually straightforward, but can be complicated by overlapping the source and target fields. Keep in mind that the copy is made one byte at a time. Consider the following examples,

```
Object code          Assembler code
                     ONE      DC    C'1'           ONE = X'F1'
                     FIELDA   DC    CL3'ABC'       FIELDA = X'C1C2C3'
                     FIELDB   DC    XL3'123456'    FIELDB = X'123456'
                              ...
D302C008C00B                 MVZ FIELDA,FIELDB    After FIELDA = X'113253'
D302C00EC008                 MVZ FIELDB,FIELDA    After FIELDB = X'C2C4C6'
D302C008C007                 MVZ FIELDA,ONE       After FIELDA = X'F1F2F3'
```

In the first **MVZ** above, three consecutive zoned half-bytes in FIELDB are simply copied to the zoned portions of FIELDA. The half-bytes are copied, one at a time, moving left to right within both operands. In the second example, three consecutive bytes are copied into FIELDB (implicit length = 3) from FIELDA. The third **MVZ** is complicated by the fact that the source and target fields overlap. We will examine the third move in some detail.

MVZ FIELDA,ONE THIS IS A 3-BYTE MOVE



First half-byte of source copied to first half-byte of target.

Second half-byte of source copied to second half-byte of target.

Third half-byte of source copied to third half-byte of target.

Examples

Some Unrelated MVZ's:

```
A DC   X'123456'
B DC   X'ABCDEF'
C DC   X'A1B2'
```

```
         ...                  Result:
         MVZ      A,B         A = X'A2C4E6'  B = X'ABCDEF'
         MVZ      A+1,B       A = X'12A4C6'  B = X'EBCDEF'
         MVZ      A+1(2),B    A = X'12A4C6'  B = X'ABCDEF'
         MVZ      B,=X'D1E2'  B = X'DBED?F'  One byte copied from literal pool
         MVZ      B,B+1       B = 'CBEDAF'   Left shift
         MVZ      B+1(2),B    B = 'ABADAF'   1st byte propagated
         MVZ      C,A         C = '1132'     A = X'123456'
         MVZ      A(L'C),C    A = 'A2B456'   Explicit Length attribute
         MVZ      A(1000),B   Assembly Error - max length is 256 bytes
         MVZ      A,B(20)     Assembly Error - Op-1 determines length
```

Tips

1. Pay attention to the lengths of the fields involved in any **MVZ** statement. If the target field is longer than the source field, bytes following the source may be transferred. If the target field is shorter than the source field, bytes in the source may be truncated.

2. Consider using Shift and Round Packed, **SRP**, instead of **MVZ** if you are working with packed fields.

Trying It Out in VisibleZ:

1. Load the program **mvz.obj** from the `\Codes` directory and single-step through each instruction until you are about to execute the `MVZ` instruction.

2. What is the length associated with the `MVZ` instruction?

3. What are the zone parts of the source field?

4. What are the zone parts of the target field?

5. Does this instruction change the numeric parts of the target?

Programming Exercise

Use the following data to create an 80-byte record input file that consists of three consecutive five-byte character fields in each record. The last 65 bytes in each record is ignored.

```
AAAAABBBBBCCCCC
AAAAACCCCCBBBBB
BBBBBAAAAACCCCC
BBBBBCCCCCAAAAA
CCCCCAAAAABBBBB
CCCCCBBBBBAAAAA
    A    B    C
    A    C    B
    B    A    C
    B    C    A
    C    A    B
    C    B    A
```

Write a program that reads each record and prints each field in the record in lexicographic order. Print each field in a column. The first three records might be printed like this:

```
      AAAAA     BBBBB     CCCCC
      AAAAA     BBBBB     CCCCC
      AAAAA     BBBBB     CCCCC
```

The last record might be printed like this:

```
           A         B         C
```

PROBLEMS

5-1. If an instruction is of type SS_1, where are its two operands stored?
5-2. Where are the two operands of an SI instruction stored?
5-3. If an instruction is of type SS_1, what determines the number of bytes that will be affected by the instruction?
5-4. What is the maximum number of bytes that can be copied using MVC?
5-5. How many bytes can be copied by MVI?
5-6. In the instruction below, where is the storage for the character 'A'?

```
MVI     INITIAL,C'A'
```

5-7. What is the effect of the following three instructions?

```
MVI     INITIAL,C'A'
MVI     INITIAL,X'C1'
MVI     INITIAL,193
```

5-8. Assume the following definitions,

```
X       DS      CL10
Y       DS      CL10
Z       DS      CL10
```

Write the code that would swap the contents of X and Y.

5-9. Assume the following definitions,

```
X       DC    CL5'AABBC'
Y       DC    CL5'AABCD'
```

How is the condition code set by the following comparison operations?

```
        CLC   X,Y
        CLC   Y,X
        CLC   X,X
        CLI   X,C'B'
        CLC   X(3),Y
        CLC   X+4,Y+3
        CLI   Y+3,X'C3'
```

5-10. Why should you usually avoid using relative addresses like X+10 and RECIN+37?
5-11. Assume the following definitions,

```
X       DC    C'A'
Y       DC    CL5
```

What are the contents of Y after executing the instruction below?

```
        MVC   Y,X
```

5-12. Why should you usually avoid using explicit lengths as in the instruction below?

```
        MVC   X(37),Y
```

5-13. What are the four values of the condition code? What condition does each value represent?
5-14. What advantages does BRC offer compared to BC?
5-15. Branch instruction BZ and BE generate the same object code. How is that possible?