Reading Object Code

A Visible/Z Lesson

The Idea:

When programming in a high-level language, we rarely have to think about the specific code that is generated for each instruction by a compiler. But as an assembly programmer, it is critical that we be able to read the machine code that is produced for each instruction. Otherwise, there will be programs that are extremely difficult to understand and debug. VisibleZ was built to help you visualize each instruction as it appears in object code and watch it execute.

When the assembler processes an instruction, it converts the instruction from its mnemonic form to a standard machine-language (binary) format called an “instruction format”. In the process of conversion, the assembler must determine the type of instruction, convert symbolic labels and explicit notation to a base/displacement format, determine lengths of certain operands, and parse any literals and constants. Consider the following “Move Character” instruction,

```
MVC   COSTOUT,COSTIN
```

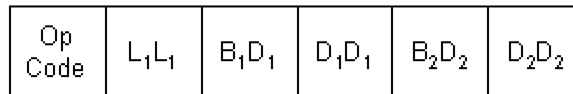
The assembler must determine the operation code (x’D2’) for MVC, determine the length of COSTOUT, and compute base/displacement addresses for both operands. After assembly, the result which is called “object code”, might look something like the following in hexadecimal,

```
D207C008C020
```

The assembler generated 6 bytes (12 hex digits) of object code in a Storage to Storage (SS) type one format. In order to understand the object code which an assembler will

generate, we need some familiarity with 5 basic instruction formats (there are other instruction types covering privileged, non-privileged, and semiprivileged instructions which are beyond the scope of this discussion).

First we consider the **Storage to Storage** type one (**SS_1**) format listed below. This is the instruction format for the MVC instruction above.



Op Code	L_1L_1	B_1D_1	D_1D_1	B_2D_2	D_2D_2

Byte 1 - machine operation code

Byte 2 - length -1 in bytes associated with operand 1

Byte 3 and 4 - the base/displacement address associated with operand 1

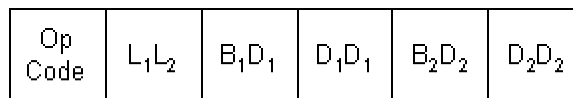
Byte 5 and 6 - the base/displacement address associated with operand 2

Each box represents one byte or 8 bits and each letter represents a single hexadecimal digit or 4 bits. The subscripts indicate the number of the operand used in determining the contents of the byte. For example, the instruction format indicates that operand 1 is used to compute L_1L_1 , the length associated with the instruction. If we reconsider the assembled form of the MVC instruction above we see that the op-code is x'D2', and the length, derived from COSTOUT, is listed as x'07'. Since the assembler always decrements the length by 1 when converting to machine code, we determine that COSTOUT is 8 bytes long - 8 bytes will be moved by this instruction. Additionally we see that the base register for COSTOUT is x'C' (register 12) and the displacement is x'008'. The base/displacement address for COSTIN is x'C020'.

Why was register 12 chosen as the base register? How were the displacements computed? These parts of the object code could not be determined by the information given in the example above. In order to determine base/displacement addresses we must examine the "USING" and "DROP" directives that are coded in the program. These directives are discussed in the topic called **BASE DISPLACEMENT ADDRESSING** on the website.

Being able to read object code is a necessary skill for an assembler programmer as knowledge of an instruction's format gives several important clues about semantics of the instruction. For example, knowing that MVC is a storage to storage type one instruction, informs us that both operands are fields in memory and that the first operand will determine the number of bytes that will be moved. Since the length (L_1L_1) occupies one byte or 8 bits, the maximum length we can create is $2^8 - 1 = 255$. Recall that the assembler decrements the length when assembling, so the instruction is capable of moving a maximum of 256 bytes. The 256 byte limitation is shared by all storage to storage type one instructions.

Storage to Storage type two (**SS_2**) is a variation on SS_1 .



Byte 1 - machine operation code

Byte 2 - L_1 - the length associated with operand 1 (4 bits)

L_2 - the length associated with operand 2 (4 bits)

Byte 3 and 4 - the base/displacement address associated with operand 1

Byte 5 and 6 - the base/displacement address associated with operand 2

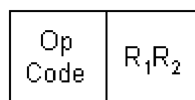
The only difference between SS_1 and SS_2 is the length byte. Notice that both operands contribute a length in the second byte. Since each length is 4 bits, the maximum value that could be represented is $2^4 - 1 = 15$. Again, since the assembler decrements the length by 1, the instruction can process operands that are large as 16 bytes. There are many arithmetic instructions that require the machine to use the length of both operands. Consider the example below,

Object Code **Source Code**

```
                    AFIELD   DS    PL4
                    BFIELD   DS    PL2
                             ...
FA31C300C304                 AP    AFIELD,BFIELD
```

AP (Add Packed) is an instruction whose format is SS_2. Looking at the object code that was generated, we see that x'FA' is the op-code and that the length of the first operand is x'3' which was computed by subtracting 1 from the length of AFIELD. Similarly, the length of BFIELD was used to generate the second length of x'1'. In executing this instruction, the machine makes use of the size of both fields. In this case, a 2 byte field is added to a 4 byte field.

A second type of instruction format is **Register to Register** (**RR**).



Byte 1 - machine operation code

Byte 2 - R1 - the register which is operand 1

R2 - the register which is operand 2

Instructions of this type have two operands, both of which are registers. An example of an instruction of this type is LR (Load Register). The effect of the instruction is to copy the contents of the register specified by operand 2 into the register specified by operand 1. The following LR (Load Register) instruction,

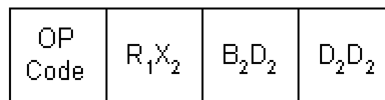
```
LR    R3,R12
```

would cause register 12 to be copied into register 3. The assembler would produce the object code listed below as a result of the LR instruction.

```
183C
```

Examining the object code we see that the op-code is x'18' , operand 1 is register 3, and operand 2 is register 12. You should note that 4 bits are enough to represent any of the registers which are numbered 0 through 15.

A third type of instruction format is **Register to Indexed Storage** (**RX**).



Byte 1 - machine operation code

Byte 2 - R_1 - the register which is operand 1

X_2 - the index register associated with operand 2

Byte 3 and 4 - the base/displacement address associated with operand 2

For instructions of this type, the first operand is a register and the second operand is a storage location. The storage location is designated by a base/displacement address as well as an index register. The subject of index registers is discussed in the topic **BASE DISPLACEMENT ADDRESSING**. L (Load) is an example of an instruction of type RX. Consider the example below.

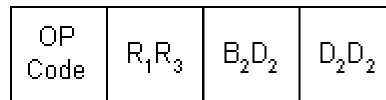
```
L     R5,TABLE(R7)
```

The Load instruction copies a fullword from memory into a register. The above instruction might assemble as follows,

```
5857C008
```

The op-code is x’58’, operand 1 is specified as x’5’, the index register is denoted x’7’ and Operand 2 generates the base/displacement address x’C008’. Again, from the information given in the example above, there is no way to determine how the base/displacement address was computed.

Related to the RX type is a similar instruction format called **Register to Storage** (**RS**). In this type the index register is replaced by a register reference or a 4-bit mask (pattern).



One instruction which has a Register to Storage format is STM (Store Multiple). An example of how STM can be coded is as follows,

```
STM    R14,R12,12(R13)
```

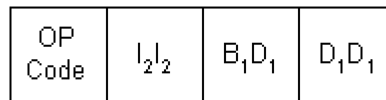
The previous instruction would generate the following object code,

90ECD00C

where x'90' is the op-code, x'E' = 14 is operand 1, x'C' = 12, is treated as R_3, and x'D00C' is generated from an explicit base/displacement address (12(R13)).

The fifth and final instruction format that we will consider is called **Storage Immediate** (**SI**). In this format, the second operand, called the immediate constant, resides in the second byte of the instruction. This constant is usually specified as a self-defining term.

The format for SI instructions is listed below.



Byte 1 - machine operation code

Byte 2 - I_2I_2 - the immediate constant denoted in operand 2

Byte 3 and 4 - the base/displacement address associated with operand 1

An example of a storage immediate instruction is Compare Logical Immediate (CLI). This instruction will compare one byte in storage to the immediate byte which resides in the instruction itself. We see from the instruction format that operand 2 is the immediate constant. For example, consider the instruction below.

```
CLI    CUSTTYPE,C'A'
```

When assembled, the object code might look like the following,

```
95C1C100
```

The op-code is x'95', the self-defining term C'A' is converted to the EBCDIC representation x'C1', and the variable CUSTTYPE would generate the base/displacement address x'C100'. Again, there is not enough information provided to determine the exact base/displacement address for CUSTTYPE. The x'C100' address is merely an example of what might be generated.

Trying It Out in VisibleZ:

1) Load the program **readingObjectCode.obj** from the \Codes directory and single step through each instruction. Identify the type and parts of each instruction below. The code doesn't "do" anything but is representative of the different instruction formats you will encounter.

90 ec d0 0c Instruction Type = ____

90 = ____

e = ____

c = ____

d = ____

00c = ____

0d c0 Instruction Type = ____

0d = ____

c = ____

0 = ____

41 20 c0 0e Instruction Type = ____

41 = ____

2 = ____

0 = ____

c = ____

00e = ____

d2 03 c0 16 c0 1a Instruction Type = ____

d2 = ____

03 = ____

c = ____

016 = ____

c = ____

01a = ____

92 c1 c0 26 Instruction Type = ____

92 = ____

c1 = ____

c = ____

026 = ____

fa 42 c0 1e c0 23 Instruction Type = ____

fa = ____

4 = ____

2 = ____

c = ____

01e = ____

c = ____

023 = ____

07 fc

Instruction Type = ____

07 = ____

f = ____

c = ____