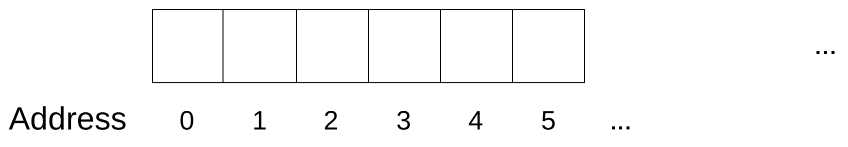
Chapter 4: How System/z Works

In which we examine how the machine finds a data field and become familiar with five kinds of instructions formats.

Where's My Data?

Internal memory on an IBM mainframe is organized as a sequential collection of bytes. The bytes are numbered starting with 0 as pictured below,



The term **address** refers to one of the bytes on our machine that are pictured above. Simple, right? This is complicated by the fact that we use the term address in at least three different ways.

1) There's the address that a programmer codes on an instruction to reference a field in memory. Let's call that a **field address**. Fields can be one or many bytes in length. In all cases, *the address of a field is the address of the first byte in the field.* Even this simple idea is complicated by the fact that programmers can code a field address in an instruction in two different ways: 1) **Symbolically** with a name that represents the address, or 2) **explicitly** by coding a combination of base register, index register, and a displacement. More on that later.

2) The assembler takes a field address and converts it into binary when you assemble a program. Let's call the converted address an **object code address**. Each object code address is represented in by a base register and a displacement, or a base register, an index register, and a displacement.

3) The object code address becomes part of an instruction that the machine must interpret when the instruction is executed. During interpretation, the machine converts each object code address into an **effective address**. Effective addresses are in binary. Originally, effective addresses were all 24 bits long. Today, System/z machines are **trimodal** and support effective addresses of 24, 31, and 64 bits. The programs we write in this book will all use 31-bit effective addresses. Thirty-one bits will provide us with a 2-gigabyte address space for our programs. The smallest effective address would be represented by 31 consecutive 0's and denotes address 0. The largest effective address would be represented by 31 consecutive 1's and has a value of $2^{31} - 1$ = 2,147,483,647.

The instructions we write can reference data that is stored in **registers** or in **memory**. Since there are only sixteen general purpose registers, specifying a register in a program is simple. We can use the decimal numbers 0 through 16 in our code, and the assembler converts each of them to 4-bit binary digits in object code. For example, consider the instruction below.

```
A     10,COST
```

The 10 in the instruction refers to register 10. The assembler converts this to '1010' in object code, a 4-bit binary number. In hexadecimal, we would refer to it as 'A'. What about `COST`? That's a symbolic reference to a field in memory – what I called a field address above. For this instruction, the assembler converts `COST` to a base/displacement address in the object code. The assembler would use 4 bits to denote a base register and 12 bits to denote a displacement. The object code address would occupy 16 bits. When this object code address was interpreted by the machine, it would add the contents of the base register and the displacement to arrive at a 31-bit effective address. We will return to this important idea shortly.

A programmer might have chosen to code the instruction above in this way,

```
A    10,4(8,11)
```

In this instruction, the second operand field address was coded explicitly rather than symbolically. The 10 still represents a register, but the 4, 8 and 11 comprise a single explicit address, and represent a displacement, an index register, and a base register respectively. Just as in the case of a symbolic address, the machine would interpret the explicit address to arrive at a 31-bit effective address. How does that happen? First, let's start by taking a closer look at base/displacement addresses.

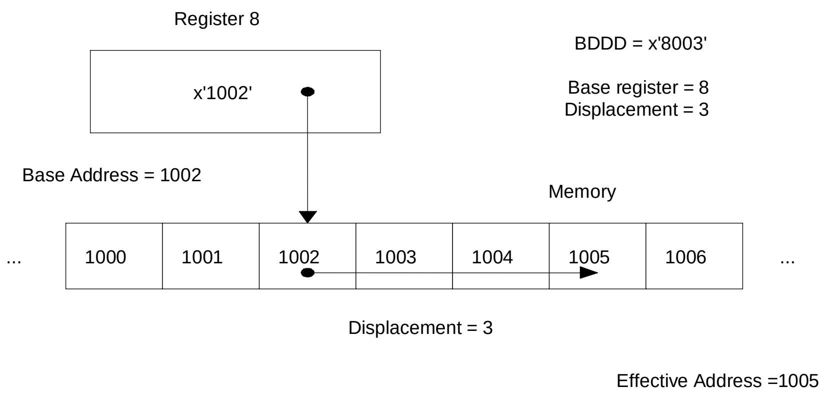
Base/Displacement (12-bit) Addressing

Object code addresses in this form consist of 4 hexadecimal digits (two bytes or sixteen bits), BDDD, where B – the first four bits - represents a base register, and DDD – the last twelve bits - represents a positive displacement. With a 12-bit displacement, the number is treated as an unsigned integer, so the smallest displacement is x'000' = 0 and the largest displacement is x'FFF' = 4095.

The machine interprets each base/displacement address as an effective address at runtime. How does this occur? The diagram below illustrates the process for the base/displacement address x'8003'. This address indicates the base register is 8, and the displacement is 003, so if we were to examine the instruction in object code, we would see x'8003' occupying two of its bytes. Let's assume that the contents of register 8 are x'00001002' (hexadecimal) at runtime.

The machine takes a base/displacement address and uses it to compute the "effective address," which is the direct address equivalent. The effective address is calculated by adding the base register's contents (x'1002') with the specified displacement (x'003'), producing the effective address (x'1005).

Effective address = Contents(Base Register) + Displacement



As a result:

A single base register using a 12-bit displacement can address 4096 = 4k bytes of storage from a given base address.

You might wonder why IBM engineers decided to use base/displacement addressing instead of the simpler direct addressing scheme. There are two advantages to using base/displacement addresses instead of direct addresses:

1. Every address is shorter in base/displacement format. For example, instead of being 4 bytes long using a direct address, base/displacement addresses are only 2 bytes. In 1962, the cost of a single byte was about $5. Saving a byte or two with every operand in an instruction represented substantial savings in memory when constructing object code. Today, memory is relatively cheap, and this is no longer a very compelling reason.

2. The base/displacement addresses generated by the assembler can be correct no matter where the program is loaded in memory. The location of each field is represented by a displacement from a fixed point inside the program. If the program is relocated in memory, the displacement from a fixed point to a given variable does not change. The base register remains fixed as well. As a result, the base/displacement address would still be correct when a program is relocated. The thing that changes after program relocation, is the contents of the base register – we can make that change at runtime.

On the other hand, if we relocate a program that uses direct addresses, every symbol would require a new direct address. Each program could only run at a specific address. We would be forced to reassemble a program every time it was run.

Base/Displacement (20-bit) Addressing

Some instructions provide for a 20-bit displacement – DDDDD. Additionally, the displacement is treated as a signed, two's complement integer instead of an unsigned value as in the case for 12-bit displacements. Since the displacement is signed, the specified address can occur before or after the address in the base register. The details on the range of values for the displacement are listed below.

$$-219 \le \text{displacement} \le +219-1, \text{ or } -524288 \le \text{displacement} \le +524287$$

The technique for generating an effective address is the same though, the displacement is added to the address in the base register to determine the effective address of the field.

As a result:

A single base register using a 20-bit displacement can provide a positive or negative displacement of 512K.

Long displacement instructions were added to make life easier for assembler programmers by providing larger areas that can be addressed by a single register. With 4k displacements, it may take many base registers to cover up all the data you need to address in your program. The use of long displacement instructions can help reduce the number of base registers needed in your program.

Index Registers

Some instructions allow for a third component, an index register, when a programmer is coding a field address in an instruction. For these instructions, the effective addresse is computed by adding the contents of the base and index registers along with the displacement.

Effective address = Contents(Base Register) + Contents (Index Register) + Displacement

Consider the following add fullword instruction,

```
A    10,4(8,11)
```

If register 8 contains x'00001000', register 11 contains x'00002000', the effective address is computed by adding x'00001000' + x'00002000' + 4 = x'00003004'.

Loading a Base Register

Base/displacement addresses only work correctly at runtime if the associated base registers contain the correct addresses. Let's discuss that in more detail now. In the first program in Chapter 3, we introduced some "standard entry code". One of the purposes of that code was establish and load a base register. Here is the code that did that:

```
BASR  R12,R0     ESTABLISH...
USING *,R12      ADDRESSABILITY
```

`BASR` is an interesting instruction. It stands for *branch and save register*. I have always thought that this instruction was misnamed because logically, the *saving* happens first, and this is followed by the *branch*. So, what is being saved? It's the address of the next executable instruction. Where is that address being saved? It's saved in the first operand – register 12. Where will we branch? Normally we branch to the address in operand two – except! – if operand two is register 0, then no branching occurs. So, what is really happening in this instruction? The address of the next executable instruction is being saved in register 12, and the execution sequence continues with that next instruction.

In a real sense, the machine is looking down at memory and asking "Where am I?" The answer to that question – the address of the next instruction - is being stored in register 12. Now that the machine knows where it is in memory, what happens next?
The next line in our code is this:

```
USING *,R12
```

USING is not an executable instruction. This line disappears when the code is assembled. Using is a message (a directive) to the assembler that tells it to use operand two – in this case register 12 – as the base register. It also tells the assembler to measure all the displacements from the base address specified by operand 1. The * is shorthand for the address of the next executable instruction.

It's important to think deeply about this before we move on. What did we load in register 12? The address of the next executable instruction after the BASR. What did the USING do? It told the assembler to pick register 12 as our base register and to measure the displacements starting from the instruction following the BASR. We are in sync!

Now ask yourself why this code would **not** work:

```
      USING *,R12      ADDRESSABILITY
      BASR  R12,R0     ESTABLISH...
```

In this case, the USING tells the assembler to pick register 12 as our base register, and to measure all the displacements from the BASR (* refers to the next location). Fine. What about the BASR? It loads the address of the next executable instruction (the one that follows our BASR) into register 12. The address we are using for the displacements doesn't agree with the address in our base register. This means that every base/displacement address will produce an incorrect address at runtime. If you have followed the reasoning in this paragraph, you are well on your way to becoming an assembler programmer. If it's still puzzling you, don't despair. It takes time for some of the complexity to sink it – and it will. Just keep going. Use the working version of the code above and wait for the light to pop on later.

Some programmers prefer a slightly different approach at the beginning:

```
ASMSKEL  CSECT                   STANDARD ENTRY CODE
STM      R14,R12,12(R13)         SAVE ALL THE REGISTERS (EXCEPT 13)
LR       R12,R15                 ESTABLISH...
USING    ASMSKEL,R12             ADDRESSABILITY
ST       R13,SAVEAREA+4          BACKWARD CHAIN CALLER (SAVE REGISTER 13,TOO)
LA       R13,SAVEAREA            POINT AT MY SAVE AREA
```

In this case, LR is copying the contents of register 15 into register 12. By convention, when a program starts execution, R15 should contain the address of the program in memory. After executing the LR, register 12 contains the address of our program. The USING is telling the assembler to pick register 12 as the base register and to compute the displacements from the beginning of our program. Once again, we are in sync. This technique has the advantage that the base address for our register is also the start of the program.

Many programs need multiple base registers, but for the moment, I'm happy to just have one. We will consider programs that need multiple base registers later.

Programmers refer to the techniques above as *establishing addressability*. That term means that you have declared one or more base registers in your program, and you have loaded those registers with the correct addresses, so that the machine can correctly compute the effective addresses your program needs.

Relative Addressing

Another newer technique for creating an effective address avoids base registers altogether. **Relative addressing** involves the assembler measuring the number of half-words (forward or backward) from the first byte of the current instruction. Consider the following code and how the assembler would process the LARL instruction which is trying to load the address of LOOP into register 8.

```
LOOP      EQU   *
          MVC   COSTOUT,COSTIN
          LARL  R8,LOOP
```

The number of bytes from the first byte of the LARL (Load Address Relative Long) instruction, to the address denoted by LOOP, is six – the length of the intervening MVC instruction. Six bytes is equivalent to three half-words. Since LOOP denotes an address that occurs before the LARL instruction, the three is represented as a negative integer. Instead of encoding the address of LOOP as a base/displacement address, the assembler encodes the address as a two's complement integer (-3). At run-time, the machine doubles -3 to get -6. The -6 is then added to the address of LARL to determine the address of LOOP. This kind of address is called a *relative address*. The address is specified relative to the address of the current instruction rather than as a displacement from a fixed base address. This has the advantage of not needing a base register for the instruction.

Some branch instructions use base/displacement addresses. Others use relative addresses. With a relative address, the location of a branch target address is measured (again in halfwords) from the address of the branch instruction. In this case, the assembled address is simply a two's complement integer. (Positive integers represent addresses that occur after the branch, negative integers represent addresses that occur before the branch.) That integer displacement is first doubled, and then added to the beginning address of the branch instruction to determine the target address of the branch. By using relative addresses which are computed by doubling a value, IBM engineers made a conscious decision to ignore odd-numbered addresses. Since instructions always start on even-numbered boundaries, this makes sense. By doubling the two's complement integer, we can jump twice as far, and none of the odd-numbered values are wasted.

Historically, instructions that use relative addresses were added to the machine architecture much later than those that use base/displacement addressing. As programs grew larger, there was a need to provide programmers with some "register relief" from the constraints of base/displacement addressing.

Instruction Formats

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 field addresses to a base/displacement format, determine lengths of individual operands, and parse any literals and constants. Consider the following **Move** instruction,

```
MVC   COSTOUT,COSTIN
```

To assemble the instruction, the assembler must look up 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-and-storage (type one) format.

To understand the object code that the assembler generates, we will develop some familiarity with five basic instruction formats. These formats existed on the System/360 and continue to exist on current machines. Additionally, we will look at a more recent instruction type – RIL. The number of instruction formats has expanded considerably with the evolution of this architecture. With the explosion in the number of instructions, there are now many other instruction types for privileged and semi-privileged instructions that are beyond the scope of this discussion.

Important tip for learning assembler: The challenge for learning assembler is to come to grips with a large number of instructions (over 100 instructions in this course, and over 2000 instructions on recent machines). Instruction formats provide an important organizing principle for dealing with this complexity. We start by learning the characteristics of six important instruction formats. Whenever we learn a fact about a single instruction format, it applies to all the instructions of that type. In the discussion that follows, the emphasis is on learning a few facts about instruction types (not specific instructions). We will cover the details of particular instructions in much greater detail later.

First, we consider the **Storage and Storage** type one (**SS$_1$**) format listed below.

Op Code	L_1L_1	B_1D_1	D_1D_1	B_2D_2	D_2D_2

Byte 1 - machine operation code
Byte 2 – the number of bytes associated with operand 1 minus 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

Now that we see the layout of a storage and storage instruction, let's decipher the machine code from above:

```
D207C008C020
```

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. This is important. If we examine 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 - this instruction will move (or copy) 8 bytes. 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'. Register 12 is the base register and the x'020' represents a 32-byte displacement.

Why was register 12 chosen as the base register? How were the displacements computed? These parts of the object code cannot be determined by the information given in the example above. To determine base/displacement addresses, we must examine the **USING** and **DROP** directives coded in the program. These directives are discussed in detail in a later chapter. For the moment, we have learned that the symbols `COSTOUT` and `COSTIN` have been converted to base/displacement addresses.

The instruction format contains everything the machine needs to know to execute an instruction. It's worth taking a moment to consider what the machine *does and doesn't* know about the MVC instruction:

1) It knows this is an MVC operation.
2) It knows to move (copy) eight bytes.
3) It knows the beginning address of operand 1.
4) It knows the beginning address of operand 2.
5) It **doesn't** know the ending address of operand 1.
6) It **doesn't** know the ending address of operand 2.
7) It **doesn't** know if operand 1 and operand 2 overlap.

As a result, MVC's behavior is simple to describe: Copy one byte at a time (consecutively) from operand 2 into operand 1, until LL + 1 bytes have been moved, where LL is the assembled length.

Being able to read object code is a necessary skill for an assembler programmer. Knowledge of an instruction's object code format gives several important clues about the instruction. For example, knowing that MVC is a storage-and-storage (type one) instruction tells us several things:

1. The instruction uses two operands.
2. Both operands are fields in memory.
3. We know the base/displacement addresses of the operands.
4. The first operand will determine the number of bytes that are copied.

Since the length (L_1L_1) occupies one byte or 8 bits, the maximum length we can encode is $2^8 - 1 = 255$. Recall that the assembler decrements the length when assembling, so the instruction can move from 1 byte to a maximum of 256 bytes. The 256-byte limitation is shared by all storage-and-storage (type one) instructions.

Storage-and-Storage type two (**SS_2**) is a variation on SS_1 that contains two lengths.

Op Code	L_1L_2	B_1D_1	D_1D_1	B_2D_2	D_2D_2

Byte 1 - machine operation code
Byte 2 - L_1 - the number of bytes associated with operand 1 minus 1 (4 bits)
L_2 - the number of bytes associated with operand 2 minus 1 (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 in the length byte. Notice that both operands contribute a length in the second byte. Since each length is 4 bits, the maximum value that can be represented is $2^4 - 1 = 15$. Again, since the assembler decrements the length by 1, this kind of instruction can process operands that are large as 16 bytes. Many arithmetic instructions 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 generated object code, 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 one 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.

The second type of instruction format we consider is **Register and Register (RR)**.

Op Code	R_1R_2

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 would cause the contents of register 12 to be copied into register 3.

```
LR  R3,R12
```

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 four bits are enough to represent any of the registers numbered 0 through 15.

The third type of instruction format we consider is **Register and Indexed Storage (RX)**.

Op Code	R_1X_2	B_2D_2	D_2D_2

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. 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'. From the limited information given in the example above, it is impossible to determine how the displacement value of x'008' was computed.

Related to the RX type is a similar instruction format called **Register and Storage (RS)**. There are only a few instructions that have this type. In this type, the index register is replaced by a register reference or a 4-bit mask (pattern). One unusual feature of RX instructions is the use of three operands.

Op Code	R_1R_3	B_2D_2	D_2D_2

An example of an instruction that has a Register and Storage format is STM (Store Multiple). An example of how STM can be coded follows below,

```
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 instruction format we consider is called **Storage Immediate (SI)**. In this format, the second operand, the immediate constant, resides in the instruction's second byte. This constant is usually specified as a self-defining term (a term whose meaning should be obvious).

The format for SI instructions is listed below.

Op Code	I_2I_2	B_1D_1	D_1D_1

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 inside the instruction. 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' (a character 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.

The sixth and final instruction format we consider is **RIL**, listed below.

Op Code	R_1OP	I_2I_2	I_2I_2	I_2I_2	I_2I_2

Byte 1 - machine operation code – first two digits
Byte 2 – R1 - the register associated operand 1
OP – the third digit in the operation code for the instruction
Byte 3, 4, 5, and 6 - the number of half-words (two's complement from the first byte of the instruction to the address associated with operand 2

The instruction format for a LARL instruction is RIL. Here is an example of a LARL instruction in object code.

```
        C080FFFFFFFD
```

The op-code is `C00` and was gathered from the first, second and fourth digits in the object code. LARL is loading an address into register 8 which was denoted by the third digit in the object code. The address that is being loaded is determined by doubling FFFFFFFD which is -3 in two's complement. The result, -6, is added to the runtime address of the LARL instruction to determine the effective address that is loaded into register 8.

You are not expected to memorize the instruction formats we just discussed. You may not grok all the details above. That's understandable. At this point, I just want you to become aware that the assembler converts the code you write into a collection of specific patterns called *instruction formats*. Over time, it's helpful to become more familiar with these types. An instruction format provides details about how an instruction works and makes learning new instructions easier.

Going Old School

The term **explicit addressing** refers to a programming practice of representing the address of a byte in memory by explicitly specifying a base register and a displacement, or a base register, an index register, and a displacement. It can also involve coding an explicit length. Whenever possible, we would prefer to represent a byte in memory by using a symbolic name rather than by using an explicit address. Usually, this *is* possible. For instance, consider the following declaration,

```
 XFIELD    DS CL5
```

This field declaration specifies a length attribute of five. The symbol `XFIELD` represents the address of the **first** byte of that field. If you are using an MVC instruction which has an SS1 format, the assembler will convert this symbolic address to a base/displacement format representing the address of the **first** byte of the field. This base/displacement address (BDDD in hexadecimal) occupies two bytes in the assembled code. (See the instruction format for an SS1 instruction.) Unfortunately, using a symbolic name like `XFIELD` to reference memory is sometimes inconvenient, or even impossible. At these times, we must resort to an explicit address to reference a location in memory.

The explicit format for each instruction in this text is provided at the top left of my descriptions of each instruction. The explicit formats are also provided for each instruction in IBM's Principles of Operation manual. For example, the MVC instruction (which we discuss in detail later) lists the following explicit format.

```
  MVC     D1(L1,B1),D2(B2)

D1 - Operand 1 displacement          D2 – Operand 2 displacement
L1 – Operand 1 length                B2 – Operand 2 base register
B1 – Operand 1 base register
```

The following example MVC instruction, uses explicit addressing.

```
MVC 4(8,12),32(8)
```

With the explicit format above, we can decipher what the instruction will do. This an MVC instruction which copies 8 bytes (L1 = 8). The number of bytes is determined by the length of operand 1. Why didn't we subtract one and code 4(7,12)? The answer is that the assembler subtracts one when the explicit code is assembled into object code.

What is the target location of the move? The answer is 4(12) – a four-byte displacement off register 12. The effective address is computed by adding the contents of base register 12 plus the 4-byte displacement:

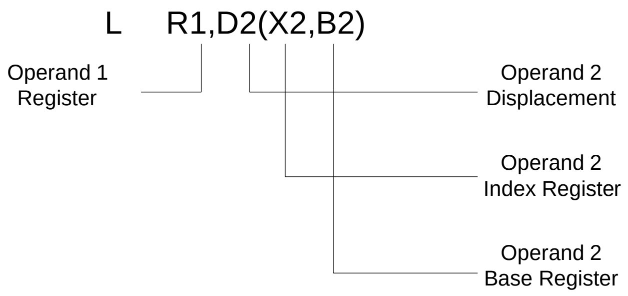
Effective address = Contents(Base register 12) + 4

What is the source of the data being moved? The answer is given explicitly as 32(8). This is a 32-byte displacement off register 8. The effective address is computed by adding the contents of base register 8 plus the 32-byte displacement:

Effective address = Contents(Base register 8) + 32

At execution time, it is imperative that registers 8 and 12 contain the correct base addresses.

Every instruction format has its own explicit format, so besides learning the instruction formats for each instruction, we eventually need to know (or look up) how to explicitly code each instruction. For instance, the load instruction explicit format appears below.



Consider the following explicitly coded instruction,

```
L     10,4(3,5)
```

Register 10 is loaded with a fullword in memory. What is the source of the load? The explicit format provides the answer. The base register is 5, register 3 is an index register, and 4 is a displacement. The effective address can be computed by adding the contents of the base register, plus the contents of the index register, plus the displacement:

Effective Address = Contents(register 5) + Contents(register 3) + 4

Since this is a beginning text, I will use equate names to designate registers when coding explicitly, so I would normally code the instruction above as

```
L     R10,4(R3,R5)
```

I think this makes it clear that 4 is a displacement and not a register. Many programmers prefer a simpler, less-adorned style, and avoid symbolic EQU names for registers. This is a matter of personal style.

Knowledge of the explicit format notation is critical to understanding many instructions. For instance, it is common practice to code an instruction similar to the one below.

```
LA  R5,10
```

How do we interpret this code? First, we could look up a **Load Address** instruction and discover that it is an RX instruction, and that it has an explicit format similar to the load instruction described above. The first operand, R5 (equated to 5), is the target register of the load. The second operand of an RX instruction appears as D2(X2,B2). But the instruction above has no parentheses - they were omitted, and therefore the base and index registers are absent. This means that 10 is a displacement.

So, what is the effective address? Since the base and index registers were omitted, zero was chosen for the base and index registers.

Important fact: The machine never uses register zero as a base or index register.

This means that the effective address is simply 10. The instruction's effect is to load a “small” number (address) into register 5. Experienced programmers understand this and often choose this method as an effective way to load “small” numbers. What is meant by “small”? Since the number we are loading is a displacement, the largest number we could code in this way is 4095 = X’FFF’, which is the maximum displacement allowed for this instruction.

Mixing Symbolic and Explicit Notation

It is also possible to mix symbols and explicit notation within the same instruction. For example,

```
 L    R8,XWORD(5)
```

In the load instruction above, 8 is the Operand 1 register into which a fullword will be loaded. What does the notation `XWORD(5)` mean? The assembler can always process a symbol like `XWORD` and produce a base register and a displacement. Operand 2, according to the explicit format for a Load instruction, uses a base register, an index register, and a displacement. This means that the 5 is an index register. A related example appears below.

```
MVC  AFIELD(8),BFIELD
```

Operand 1 appears as `AFIELD(8)`. Again, the assembler can process the symbol `AFIELD` and produce a base register and a displacement. Checking the explicit notation for a move character instruction, we see that the 8 must be the length of Operand 1. Normally, the length would be taken from the length attribute of `AFIELD`. Mixing notations allows us to override the “implicit” length with an “explicit” length.

Parting Words

This chapter has been heavy on details. I don’t won’t you to be overwhelmed by them. Instead, try to take away a few principles:

- A field’s address can be specified symbolically or explicitly.
- The machine can locate a byte in memory using a combination of a base register, index register, and a displacement (base/displacement addressing).
- The machine can locate a byte in memory by measuring the number of halfwords from the start of an instruction (relative addressing).
- Symbolic names are converted by the assembler into object code.
- The object code follows certain patterns called instruction formats.
- Instruction formats encode all the information needed for the machine to interpret the instruction.
- It’s smart to puzzle out how the assembler converts each instruction to object code.
- Some instructions use 12-bit displacements and other use 20-bit displacements.
- Establishing addressability means loading a base register with a value that allows the machine to find the variables and target addresses in your program.

Examples

Some Explicitly-Coded Instructions

```
LM     R14,R12,12(R13) 14 AND 12 REPRESENT A RANGE OF REGISTERS.
                       WORDS ARE LOADED BEGINNING WITH THE WORD
                       12 BYTES OFF REGISTER 13.

MVC    X(9),Y          A 9-BYTE EXPLICIT LENGTH
L      R8,9            ASSEMBLES FINE. 9 DISPLACMENT,0 INDEX, 0 BASE
                       PRODUCES A RUNTIME ERROR - TRIES TO LOAD THE WORD AT
                       ADDRESS 9 AND GENERATES A S0C4 ABEND
LA     R8,9            BETTER! LOADS 9 AS AN ADDRESS INTO REGISTER 8
AP     X(4),Y(3)       SS2 INSTRUCTIONS CAN HAVE TWO LENGTHS
MVI    0(R8),C' '      MOVES A BLANK TO THE STORAGE ADDRESS CONTAINED IN R8
BASR   R12,R0          RR INSTRUCTIONS ARE ALWAYS EXPLICIT
```

Tips

1. Avoid the use of explicit notation whenever possible to simplify your code.

2. If you must use explicit notation, consider coding base and index registers using equate names (R0, R1, ...). The use of equate names for registers is a stylistic decision.

3. One important use of explicit addressing is during parameter passing. We will revisit explicit notation when we study how one program can call another program and pass parameters.

4. Consider the use of a DSECT as a means of creating symbolic names instead of coding explicitly. DSECTs are a powerful tool for providing symbolic names. DSECTs will be covered in a later chapter.

5. Explicit addresses must always be used if you have not yet issued a **USING** directive or if no base register is available. Consider the beginning lines of our standard entry code for each program:

   ```
   STM   R14,R12,12(R13)
   BASR  R12,R0
   USING *,R12
   ST    R13,SAVE+4
   ```

   The first use of a symbol (SAVE) can only occur after coding a USING directive. Before the USING statement, any use of a symbol would generate an assembly error. The `STM` and `BASR` instructions are coded explicitly.

Trying It Out in VisibleZ:

1. Load the program **readingobjectcode.obj** from the \Codes directory. This program demonstrates examples of instructions in five different formats.

2. The STM instruction moves data from a range of registers into memory. The register range is indicated by the two registers named in the second byte. What are these registers? Can you see these registers indicated by the highlighting in VisibleZ?

3. What is the base displacement in object code that is mentioned in the STM instruction? Can you see this address reflected in the highlighting?

4. Cycle the machine to the second instruction, a BASR. What is the instruction format? What two registers are named in the object code?

5. Cycle to the third instruction, an LA. What is the instruction format? LA instructions are used to put an address in a register. Which register is targeted by the LA?

6. Cycle to the fourth instruction an MVC. What is the instruction format? What are the base/displacement addresses of the source and target? What are the effective addresses? Show the arithmetic that converts each base/displacement address into an effective address.

7. Cycle to the fifth instruction, an MVI. This instruction contains an immediate constant. What is that constant in object code? What character does it represent?

8. Cycle to the sixth instruction, an AP, which is an SS2 instruction. This type of instruction encodes two lengths – one for the source field and one for the target. What are these two lengths?

PROBLEMS

4-1. What is the address of the first byte on a machine?
4-2. What are the two techniques a programmer can employ to specify the address of a field?
4-3. How is the object code address x’9032’ used to compute an effective address?
4-4. How is the explicit address 4(5,6) used to compute an effective address?
4-5. What are the advantages of relative addressing?
4-6. How many bytes can be addressed using a base register and a 12-bit displacement?
4-7. How many bytes can be addressed using a base register and a 20-bit displacement?
4-8. How many bytes can be addressed from a single instruction using relative addressing?
4-9. Which register never functions as a base register?
4-10. What happens if a base register becomes corrupted?
4-11. Why would a program require more than one base register?
4-12. Explain what the following code does,

```
BASR  R12,R0
USING *,R12
```

4-13. Explain what’s wrong with this code,

```
USING *,R12
BASR  R12,R0
```