OSBORNE/McGRAW-HILL

Z8000®

Z8000® ASSEMBLY LANGUAGE PROGRAMMING

BY LANCE A. LEVENTHAL, ADAM OSBORNE & CHUCK COLLINS

Z8000 Assembly Language Programming

Lance A. Leventhal
Adam Osborne
Chuck Collins

OSBORNE/McGraw-Hill
Berkeley, California

Published by
OSBORNE/McGraw-Hill
630 Bancroft Way
Berkeley, California 94710
U. S. A

For information on translations and book distributors outside of the U. S. A., please write OSBORNE/McGraw-Hill at the above address.

Z8000 Assembly Language Programming

34567890 DODO 89876543

ISBN 0-931988-36-5

Technical editors for this book were Joaquin Miller of Lovelace Computing, and Susanna Jacobson of OSBORNE/McGraw-Hill.

Cover design by Timothy Sullivan.

Contents

Program Examples

1

Introduction to Assembly Language Programming

This book describes assembly language programming. It assumes that you are familiar with *An Introduction to Microcomputers: Volume 1 — Basic Concepts* (Berkeley: Osborne/McGraw-Hill, 1980). Chapters 6 and 7 of that book are especially relevant. This book does not discuss the general features of computers, microcomputers, addressing methods, or instruction sets; you should refer to *An Introduction to Microcomputers: Volume 1* for that information.

HOW THIS BOOK HAS BEEN PRINTED

Notice that text in this book has been printed in boldface type and lightface type. This has been done to help you skip those parts of the book that cover subject matter with which you are familiar. You can be sure that lightface type only expands on information presented in the previous boldface type. Therefore, read only boldface type until you reach a subject about which you want to know more, at which point start reading the lightface type.

THE MEANING OF INSTRUCTIONS

The instruction set of a microprocessor is the set of binary inputs which produce defined actions during an instruction cycle. An instruction set is to a microprocessor what a function table is to a logic device such as a gate, adder, or shift register. Of course, the actions that the microprocessor performs in response to the instruction inputs are far more complex than the actions that combinatorial logic devices perform in response to their inputs.

An instruction is simply a binary bit pattern — it must be available at the data inputs to the microprocessor at the proper time in order to be interpreted as an instruction. For example, when the Z8000 microprocessor receives the 16-bit binary pattern 1000000000001000 as the input during an instruction fetch operation, the pattern means:

"Add the contents of Register RH0 to the contents of Register RL0."

Similarly, the pattern 1100100011111111 means:

"Load 11111111 into Register RL0."

The microprocessor (like any other computer) recognizes only binary patterns as instructions or data; it does not recognize words or octal, decimal, or hexadecimal numbers.

A COMPUTER PROGRAM

A program is a series of instructions that cause a computer to perform a particular task.

Actually, a computer program includes more than instructions; it also contains the data and the memory addresses that the microprocessor needs to accomplish the task defined by the instructions. Clearly, if the microprocessor is to perform an addition, it must have two numbers to add and a destination for the result. The computer program must determine the sources of the data and the destination of the result as well as specifying the operation to be performed.

All microprocessors execute instructions sequentially unless one of the instructions changes the execution sequence or halts the computer (i.e., the processor gets the next instruction from the next consecutive memory address unless the current instruction specifically directs it to do otherwise).

Ultimately every program becomes translated into a set of binary numbers. For example, this is the Z8000 program that adds the contents of memory locations 6000_{16} and 6002_{16} and places the result in memory location 6004_{16}:

```
0110000100000000
0110000000000000
0100000100000000
0110000000000010
0110111100000000
0110000000000100
```

This is a machine language, or object, program. If this program were entered into the memory of a Z8000-based microcomputer, the microcomputer would be able to execute it directly.

THE BINARY PROGRAMMING PROBLEM

There are many difficulties associated with creating programs as object, or binary machine language, programs. These are some of the problems:

1. The programs are difficult to understand or debug (binary numbers all look the same, particularly after you have looked at them for a few hours).
2. The programs are slow to enter since you must enter each bit individually using front panel switches.
3. The programs do not describe the task which you want the computer to perform in anything resembling a human readable format.
4. The programs are long and tiresome to write.
5. The programmer often makes careless errors that are very difficult to find.

For example, the **following version of the addition program shown above has two numbers transposed. Try to find the error:**

```
0110000100000000
0110000000000000
0010000100000000
0110000000000010
0110111100000000
0110000000000100
```

Although the computer handles binary numbers with ease, people do not. People find binary programs long, tiresome, confusing, and meaningless. Eventually, a programmer may start remembering some of the binary codes, but such effort should be spent more productively.

USING OCTAL OR HEXADECIMAL

We can improve the situation somewhat by writing instructions using octal or hexadecimal, rather than binary, numbers. We will use hexadecimal numbers in this book because they are shorter, and because they are the standard for the microprocessor industry. Table 1-1 shows the hexadecimal digits and their binary equivalents. **The Z8000 program to add two numbers now becomes:**

```
6100
6000
4100
6002
6F00
6004
```

At the very least, the hexadecimal version is shorter to write and not quite so tiring to examine.

Errors are somewhat easier to find in a sequence of hexadecimal digits. The erroneous version of the addition program, in hexadecimal form, becomes:

```
6100
6000
2100
6002
6F00
6004
```

The mistake is easier to spot.

What do we do with this hexadecimal program? The microprocessor understands only binary instruction codes. If your front panel has a hexadecimal keyboard instead of bit switches, you can key the hexadecimal program directly into memory — the keyboard logic translates the hexadecimal digits into binary numbers. But what if your front panel has only bit switches? You can convert the hexadecimal digits to binary by yourself, but this is a repetitive, tiresome task. People who attempt it make all sorts of petty mistakes, such as looking at the wrong line, dropping a bit, or transposing a bit or a digit. Besides, once we have converted our hexadecimal program we must still place the bits in memory through the switches on the front panel.

Hexadecimal Loader

These repetitive, grueling tasks are, however, perfect jobs for a computer. The computer never gets tired or bored and never makes mistakes. **The idea then is to write a program that accepts hexadecimal numbers, converts them into binary numbers, and places them in memory. This is a standard program provided with many microcomputers; it is called a hexadecimal loader.**

The hexadecimal loader is a program like any other. It occupies memory space: in some systems, only long enough to load another program; in others, it occupies a reserved, read-only section of memory. Your microcomputer may not have bit switches on its front panel; it may not even have a front panel. This reflects the machine designer's decision that binary programming is not only impossibly tedious but also wholly unnecessary. The hexadecimal loader in your system may be part of a larger program called a monitor, which also provides a number of tools for program debugging and analysis.

A hexadecimal loader certainly does not solve every programming problem. The hexadecimal version of the program is still difficult to read or understand; for example, it does not distinguish instructions from data or addresses, nor does the program listing provide any suggestion as to what the program does. What does 6100 or 6F00 mean? Memorizing a card full of codes is hardly an appetizing proposition. Furthermore, the codes will be entirely different for a different microprocessor, and the program will require a large amount of documentation.

Table 1-1. Hexadecimal Conversion Table

Hexadecimal Digit	Binary Equivalent	Decimal Equivalent
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

INSTRUCTION CODE MNEMONICS

An obvious programming improvement is to assign a name to each instruction code. The instruction code name is called a "mnemonic," or memory jogger. The instruction mnemonic should describe in some way what the instruction does.

Devising Mnemonics

In fact, every microprocessor manufacturer (they can't remember hexadecimal codes either) provides a set of mnemonics for the microprocessor instruction set. **You do not have to abide by the manufacturer's mnemonics**; there is nothing sacred about them. However, they are standard for a given microprocessor and therefore understood by all users. These are the instruction names that you will find in manuals, cards, books, articles, and programs. The problem with selecting instruction mnemonics is that not all instructions have "obvious" names. Some instructions do have obvious names (e.g., ADD, AND, OR), others have obvious contractions (e.g., SUB for subtraction, XOR for exclusive OR), while still others have neither. The result is such mnemonics as WMP, PCHL, and even SOB (try and guess what that means!). Most manufacturers come up with mostly reasonable names and a few hopeless ones. However, users who devise their own mnemonics rarely seem to do much better than the manufacturer.

Along with the instruction mnemonics, the manufacturer will usually assign names to the CPU registers. As with the instruction names, some register names are obvious (e.g., A for Accumulator) while others may have only historical significance. Again, we will use the manufacturer's suggestions simply to promote standardization.

An Assembly Language Program

If we use standard Z8000 instruction and register mnemonics, as defined by Zilog, our Z8000 addition program becomes:

```
LD      R0,%6000
ADD     R0,%6002
LD      %6004,R0
```

The program is still far from obvious, but at least some parts are comprehensible. ADD R0,%6002 is a considerable improvement over 4100, and LD does suggest loading data into a register or memory location. **Such a program is an assembly language program.**

THE ASSEMBLER PROGRAM

How do we get the assembly language program into the computer? We have to translate it, either into hexadecimal or into binary numbers. **You can translate an assembly language program by hand,** instruction by instruction. This is called hand assembly.

Hand assembly of the addition program's instruction codes may be illustrated as follows:

Instruction Name		Hexadecimal Equivalent
LD	R0,%6000	61006000
ADD	R0,%6002	41006002
LD	%6004,R0	6F006004

As in the case of hexadecimal-to-binary conversion, hand assembly is a rote task which is uninteresting, repetitive, and subject to numerous minor errors. Picking the wrong line, transposing digits, omitting instructions, and misreading the codes are only a few of the mistakes that you may make. Most microprocessors complicate the task even further by having instructions with different word lengths. Some instructions are one word long while others are two or three words long. Some instructions require data in the second and third words; others require memory addresses, register numbers, or who knows what?

Assembly is another rote task that we can assign to the microcomputer. The microcomputer never makes any mistakes when translating codes; it always knows how many words and what format each instruction requires. The program that does this job is called an "assembler." The assembler program translates a user program, or "source" program written with mnemonics, into a machine language program, or "object" program, which the microcomputer can execute. The assembler's input is a source program and its output is an object program.

An assembler is a program, just as the hexadecimal loader is. However, assemblers are more expensive, occupy more memory, and require more peripherals and execution time than do hexadecimal loaders. While users may (and often do) write their own loaders, few care to write their own assemblers.

Assemblers have their own rules that you must learn to abide by. These include the use of certain markers (such as spaces, commas, semicolons, or colons) in appropriate places, correct spelling, the proper control information, and perhaps even the correct placement of names and numbers. These rules typically are a minor hindrance that can be quickly overcome.

ADDITIONAL FEATURES OF ASSEMBLERS

Early assembler programs did little more than translate the mnemonic names of instructions and registers into their binary equivalents. However, most assemblers now provide such additional features as:

1. Allowing the user to assign names to memory locations, input and output devices, and even sequences of instructions.
2. Converting data or addresses from various number systems (e.g., decimal or hexadecimal) to binary and converting characters into their ASCII or EBCDIC binary codes.
3. Performing some arithmetic as part of the assembly process.
4. Telling the loader program where in memory parts of the program or data should be placed.
5. Allowing the user to assign areas of memory as temporary data storage and to place fixed data in areas of program memory.
6. Providing the information required to include standard programs from program libraries, or programs written at some other time, in the current program.
7. Allowing the user to control the format of the program listing and the input and output devices employed.