

programming the Z80

RODNAY ZAKS



PROGRAMMING THE Z80

RODNAY ZAKS

THIRD REVISED EDITION



"Z80" is a registered trademark of ZILOG Inc., with whom SYBEX is not connected in any way.

Cover Design by Daniel Le Noury

Every effort has been made to supply complete and accurate information. However, Sybex assumes no responsibility for its use; nor any infringements of patents or other rights of third parties which would result. No license is granted by the equipment manufacturers under any patent or patent rights. Manufacturers reserve the right to change circuitry at any time without notice.

In particular, technical characteristics and prices are subject to rapid change. Comparisons and evaluations are presented for their educational value and for guidance principles. The reader is referred to the manufacturer's data for exact specifications.

Copyright ©1982, SYBEX Inc. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to, photocopy, photograph, or magnetic or other record, without the prior written permission of the publisher.

Library of Congress Card Number: 80-5468

ISBN: 0-89588-094-6

First Edition published 1979. Third Revised Edition 1982.

Printed in the United States of America

Printing 10 9 8 7 6 5 4 3 2 1

ACKNOWLEDGEMENTS

Designing a programming textbook is always difficult. Designing it so that it will teach elementary programming as well as advanced concepts while covering both hardware and software aspects makes it a challenge. The author would like to acknowledge here the many constructive suggestions for improvements or changes made by: O.M. Barlow, Dennis L. Feick, Richard D. Reid, Stanley E. Erwin, Philip Hooper, Dennis B. Kitsz, R. Ratke, and Jim Crocker.

A special acknowledgement is also due to Chris Williams for his contribution to the instruction-set and the data structures section.

Any additional suggestions for improvements or changes should be sent to the author, and will be reflected in forthcoming editions.

Several tables in Chapter Four showing hexadecimal codes for the Z80 instructions have been reprinted by permission of Zilog Inc. Tables 2.26 and 2.27 have been reprinted by permission of Intel Corporation.

TABLE OF CONTENTS

PREFACE	13
I. BASIC CONCEPTS	15
<i>Introduction, What is programming?, Flowcharting, Information Representation</i>	
II. Z80 HARDWARE ORGANIZATION	46
<i>Introduction, System Architecture, Internal Organization of the Z80, Instruction Formats, Execution of Instructions with the Z80, Hardware Summary</i>	
III. BASIC PROGRAMMING TECHNIQUES	94
<i>Introduction, Arithmetic Programs, BCD Arithmetic Multiplication, Binary Division, Instruction Summary, Subroutines, Summary</i>	
IV. THE Z80 INSTRUCTION SET	154
<i>Introduction, Classes of Instructions, Summary, Individual Descriptions</i>	
V. ADDRESSING TECHNIQUES	438
<i>Introduction, Possible Addressing Modes, Z80 Addressing Modes, Using the Z80 Addressing Modes, Summary</i>	

VI. INPUT/OUTPUT TECHNIQUES **460**

Introduction, Input/output, Parallel Word Transfer, Bit Serial Transfer, Peripheral Summary, Input/Output Scheduling, Summary

VII. INPUT/OUTPUT DEVICES **511**

Introduction, The Standard PIO, The Internal Control Register, Programming a PIO, The Zilog Z80 PIO

VIII. APPLICATION EXAMPLES **520**

Introduction, Clearing a Section of Memory, Polling I/O Devices, Getting Characters In, Testing A Character, Bracket Testing, Parity Generation, Code Conversion: ASCII to BCD, Convert Hex to ASCII, Finding the Largest Element of a Table. Sum of N Elements, A Checksum Computation, Count the Zeroes, Block Transfer, BCD Block Transfer, Compare Two Signed 16-bit Numbers, Bubble-Sort, Summary

IX. DATA STRUCTURES **539****PART 1—THEORY**

Introduction, Pointers, Lists, Searching and Sorting, Section Summary

PART 2—DESIGN EXAMPLES

Introduction, Data Representation for the List, A Simple List, Alphabetic Set, Linked List, Summary

X. PROGRAM DEVELOPMENT **579**

Introduction, Basic Programming Choices, Software Support, The Program Development Sequence, Hardware Alternatives, The Assembler, Conditional Assembly, Summary

XI. CONCLUSION	602
<i>Technological Development, The Next Step</i>	
APPENDIX A	604
<i>Hexadecimal Conversion Table</i>	
APPENDIX B	605
<i>ASCII Conversion Table</i>	
APPENDIX C	606
<i>Relative Branch Tables</i>	
APPENDIX D	607
<i>Decimal to BCD Conversion</i>	
APPENDIX E	608
<i>Z80 Instruction Codes</i>	
APPENDIX F	615
<i>Z80 to 8080 Equivalence</i>	
APPENDIX G	616
<i>8080 to Z80 Equivalence</i>	
INDEX	617

PREFACE

This book has been designed as a complete self-contained text for learning programming, using the Z80. It can be used by a person who has never programmed before, and should also be of value to anyone using the Z80.

For the person who has already programmed, this book will teach specific programming techniques using (or working around) the specific characteristics of the Z80. This text covers the elementary to intermediate techniques required to start programming effectively.

This text aims at providing a true level of competence to the person who wishes to program using this microprocessor. Naturally, no book will effectively teach how to program, unless one actually practices. However, it is hoped that this book will take the reader to the point where he feels that he can start programming by himself and can solve simple or even moderately complex problems using a microcomputer.

This book is based on the author's experience in teaching more than 1000 persons how to program microcomputers. As a result, it is strongly structured. Chapters normally go from the simple to the complex. For readers who have already learned elementary programming, the introductory chapter may be skipped. For others who have never programmed, the final sections of some chapters may require a second reading. The book has been designed to take the reader systematically through all the basic concepts and techniques required to build increasingly complex programs. It is, therefore, strongly suggested that the ordering of the chapters be followed. In addition, for effective results, it is important that the reader attempt to solve as many exercises as possible. The difficulty within the exercises has been carefully graduated. They are designed to verify that the material which has been presented is really understood. Without doing the programming exercises, it will not be possible to realize the full value of this book as an educational medium. Several of the exercises may require time, such as the multiplication exercise. However, by doing them, you will actually program and *learn by doing*. This is indispensable.

For those who have acquired a taste for programming when reaching the end of this volume, a companion volume is planned: the *Z80 Applications Book*.

Other books in this series cover programming for other popular microprocessors.

For those who wish to develop their hardware knowledge, it is suggested that the reference books *From Chips to Systems: an Introduction to Microprocessors* (ref. C201A) and *Microprocessor Interfacing Techniques* (ref. C207) be consulted.

The contents of this book have been checked carefully and are believed to be reliable. However, inevitably, some typographical or other errors will be found. The author will be grateful for any comments by alert readers so that future editions may benefit from their experience. Any other suggestions for improvements, such as other programs desired, developed, or found of value by readers, will be appreciated.

1

BASIC CONCEPTS

INTRODUCTION

This chapter will introduce the basic concepts and definitions relating to computer programming. The reader already familiar with these concepts may want to glance quickly at the contents of this chapter and then move on to Chapter 2. It is suggested, however, that even the experienced reader look at the contents of this introductory chapter. Many significant concepts are presented here including, for example, two's complement, BCD, and other representations. Some of these concepts may be new to the reader; others may improve the knowledge and skills of experienced programmers.

WHAT IS PROGRAMMING?

Given a problem, one must first devise a solution. This solution, expressed as a step-by-step procedure, is called an *algorithm*. An algorithm is a step-by-step specification of the solution to a given problem. It must terminate in a finite number of steps. This algorithm may be expressed in any language or symbolism. A simple example of an algorithm is:

- 1—insert key in the keyhole
- 2—turn key one full turn to the left
- 3—seize doorknob
- 4—turn doorknob left and push the door

At this point, if the algorithm is correct for the type of lock involved, the door will open. This four-step procedure qualifies as an algorithm for door opening.

Once a solution to a problem has been expressed in the form of an algorithm, the algorithm must be executed by the computer. Unfortunately, it is now a well-established fact that computers cannot understand or execute ordinary spoken English (or any other human language). The reason lies in the *syntactic ambiguity* of all common human languages. Only a well-defined subset of natural language can be "understood" by the computer. This is called a *programming language*.

Converting an algorithm into a sequence of instructions in a programming language is called *programming*. To be more specific, the actual translation phase of the algorithm into the programming language is called *coding*. Programming really refers not just to the coding but also to the overall design of the programs and "data structures" which will implement the algorithm.

Effective programming requires not only understanding the possible implementation techniques for standard algorithms, but also the skillful use of all the computer hardware resources, such as internal registers, memory, and peripheral devices, plus a creative use of appropriate data structures. These techniques will be covered in the next chapters.

Programming also requires a strict documentation discipline, so that the programs are understandable to others, as well as to the author. Documentation must be both internal and external to the program.

Internal program documentation refers to the comments placed in the body of a program, which explain its operation.

External documentation refers to the design documents which are separate from the program: written explanations, manuals, and flowcharts.

FLOWCHARTING

One intermediate step is almost always used between the *algorithm* and the *program*. It is called a *flowchart*. A flowchart is simply a symbolic representation of the algorithm expressed as a sequence of rectangles and diamonds containing the steps of the algorithm. Rectangles are used for *commands*, or "executable statements." Diamonds are used for *tests* such as: If information

X is true, then take action A, else B. Instead of presenting a formal definition of flowcharts at this point, we will introduce and discuss flowcharts later on in the book when we present programs.

Flowcharting is a highly recommended intermediate step between the algorithm specification and the actual coding of the solution. Remarkably, it has been observed that perhaps 10% of the programming population can write a program successfully without having to flowchart. Unfortunately, it has also been observed that 90% of the population believes it belongs to this 10%! The result: 80% of these programs, on the average, will fail the first time they are run on a computer. (These percentages are naturally not meant to be accurate.) In short, most novice programmers seldom see the necessity of drawing a flowchart. This usually results in "unclean" or erroneous programs. They must then spend a long time testing and correcting their program (this is called the

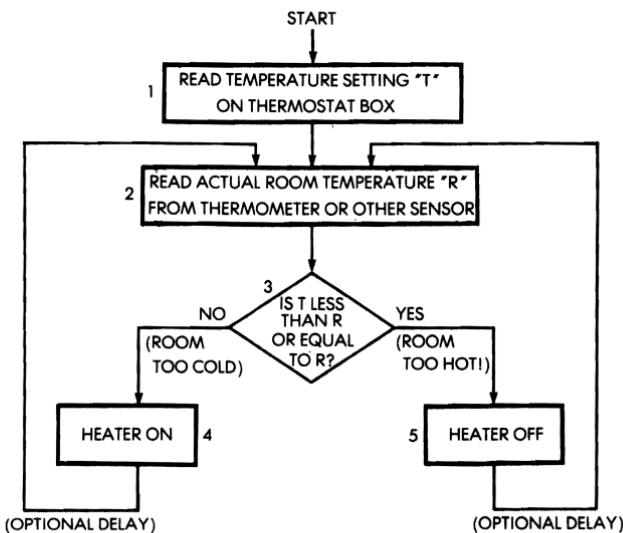


Fig. 1.1: A Flowchart for Keeping Room Temperature Constant

debugging phase). The discipline of flowcharting is therefore highly recommended in all cases. It will require a small amount of additional time prior to the coding, but will usually result in a clear program which executes correctly and quickly. Once flowcharting is well understood, a small percentage of programmers will be able to perform this step mentally without having to do it on paper. Unfortunately, in such cases the programs that they write will usually be hard to understand for anybody else without the documentation provided by flowcharts. As a result, it is universally recommended that flowcharting be used as a strict discipline for any significant program. Many examples will be provided throughout the book.

INFORMATION REPRESENTATION

All computers manipulate information in the form of numbers or in the form of characters. Let us examine here the external and internal representations of information in a computer.

INTERNAL REPRESENTATION OF INFORMATION

All information in a computer is stored as groups of bits. A *bit* stands for a *binary digit* ("0" or "1"). Because of the limitations of conventional electronics, the only practical representation of information uses two-state logic (the representation of the state "0" and "1"). The two states of the circuits used in digital electronics are generally "on" or "off", and these are represented logically by the symbols "0" or "1". Because these circuits are used to implement "logical" functions, they are called "binary logic." As a result, virtually all information-processing today is performed in binary format. In the case of microprocessors in general, and of the Z80 in particular, these bits are structured in groups of eight. A group of eight bits is called a *byte*. A group of four bits is called a *nibble*.

Let us now examine how information is represented internally in this binary format. Two entities must be represented inside the computer. The first one is the program, which is a sequence of instructions. The second one is the data on which the program will operate, which may include numbers or alphanumeric text. We will discuss below three representations: program, numbers, and alphanumerics.

Program Representation

All instructions are represented internally as single or multiple bytes. A so-called "short instruction" is represented by a single byte. A longer instruction will be represented by two or more bytes. Because the Z80 is an eight-bit microprocessor, it fetches bytes successively from its memory. Therefore, a single-byte instruction always has a potential for executing faster than a two- or three-byte instruction. It will be seen later that this is an important feature of the instruction set of any microprocessor and in particular the Z80, where a special effort has been made to provide as many single-byte instructions as possible in order to improve the efficiency of the program execution. However, the limitation to 8 bits in length has resulted in important restrictions which will be outlined. This is a classic example of the compromise between speed and flexibility in programming. The binary code used to represent instructions is dictated by the manufacturer. The Z80, like any other microprocessor, comes equipped with a fixed instruction set. These instructions are defined by the manufacturer and are listed at the end of this book, with their code. Any program will be expressed as a sequence of these binary instructions. The Z80 instructions are presented in Chapter 4.

Representing Numeric Data

Representing numbers is not quite straightforward, and several cases must be distinguished. We must first represent integers, then signed numbers, i.e., positive and negative numbers, and finally we must be able to represent decimal numbers. Let us now address these requirements and possible solutions.

Representing integers may be performed by using a *direct binary* representation. The direct binary representation is simply the representation of the decimal value of a number in the binary system. In the binary system, the right-most bit represents 2 to the power 0. The next one to the left represents 2 to the power 1, the next represents 2 to the power 2, and the left-most bit represents 2 to the power $7 = 128$.

$$\begin{array}{c} b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \\ \text{represents} \end{array}$$

$$b_7 \cdot 2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

PROGRAMMING THE Z80

The powers of 2 are:

$$2^7 = 128, 2^6 = 64, 2^5 = 32, 2^4 = 16, 2^3 = 8, 2^2 = 4, 2^1 = 2, 2^0 = 1$$

The binary representation is analogous to the decimal representation of numbers, where "123" represents:

$$\begin{array}{r} 1 \times 100 = 100 \\ + 2 \times 10 = 20 \\ + 3 \times 1 = 3 \\ \hline = 123 \end{array}$$

Note that $100 = 10^2$, $10 = 10^1$, $1 = 10^0$.

In this "positional notation," each digit represents a power of 10.

In the binary system, each binary digit or "bit" represents a power of 2, instead of a power of 10 in the decimal system.

Example: "00001001" in binary represents:

$$\begin{array}{r} 1 \times 1 = 1 \quad (2^0) \\ 0 \times 2 = 0 \quad (2^1) \\ 0 \times 4 = 0 \quad (2^2) \\ 1 \times 8 = 8 \quad (2^3) \\ 0 \times 16 = 0 \quad (2^4) \\ 0 \times 32 = 0 \quad (2^5) \\ 0 \times 64 = 0 \quad (2^6) \\ 0 \times 128 = 0 \quad (2^7) \\ \hline \text{in decimal:} & = 9 \end{array}$$

Let us examine some more examples:

"10000001" represents:

$$\begin{array}{r} 1 \times 1 = 1 \\ 0 \times 2 = 0 \\ 0 \times 4 = 0 \\ 0 \times 8 = 0 \\ 0 \times 16 = 0 \\ 0 \times 32 = 0 \\ 0 \times 64 = 0 \\ 1 \times 128 = 128 \\ \hline \text{in decimal:} & = 129 \end{array}$$

"10000001" represents, therefore, the decimal number 129.

By examining the binary representation of numbers, you will understand why bits are numbered from 0 to 7, going from right to left. Bit 0 is “ b_0 ” and corresponds to 2^0 . Bit 1 is “ b_1 ” and corresponds to 2^1 , and so on.

Decimal	Binary	Decimal	Binary
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	•	
3	00000011	•	
4	00000100	•	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	•	
9	00001001	•	
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110	•	
15	00001111		
16	00010000	•	
17	00010001		
•			
•			
•			
31	00011111	254	11111110
		255	11111111

Fig. 1.2: Decimal-Binary Table

The binary equivalents of the numbers from 0 to 255 are shown in Fig. 1-2.

Exercise 1.1: What is the decimal value of “11111100”?

Decimal to Binary

Conversely, let us compute the binary equivalent of "11" decimal:

$$\begin{array}{rcl} 11 \div 2 = 5 \text{ remains } 1 & \longrightarrow & 1 \\ 5 \div 2 = 2 \text{ remains } 1 & \longrightarrow & 1 \\ 2 \div 2 = 1 \text{ remains } 0 & \longrightarrow & 0 \\ 1 \div 2 = 0 \text{ remains } 1 & \longrightarrow & 1 \end{array} \quad (\text{LSB}) \quad (\text{MSB})$$

The binary equivalent is 1011 (read right-most column from bottom to top).

The binary equivalent of a decimal number may be obtained by dividing successively by 2 until a quotient of 0 is obtained.

Exercise 1.2: What is the binary for 257?

Exercise 1.3: Convert 19 to binary, then back to decimal.

Operating on Binary Data

The arithmetic rules for binary numbers are straightforward. The rules for addition are:

$$\begin{array}{rcl} 0+0= & 0 \\ 0+1= & 1 \\ 1+0= & 1 \\ 1+1=(1) & 0 \end{array}$$

where (1) denotes a "carry" of 1 (note that "10" is the binary equivalent of "2" decimal). Binary subtraction will be performed by "adding the complement" and will be explained once we learn how to represent negative numbers.

Example:

$$\begin{array}{rcl} (2) & 10 \\ +(1) & +01 \\ \hline =(3) & 11 \end{array}$$

Addition is performed just like in decimal, by adding columns, from right to left:

Adding the right-most column:

$$\begin{array}{r} 10 \\ +01 \\ \hline (0 + 1 = 1. \text{ No carry.}) \end{array}$$

Adding the next column:

$$\begin{array}{r}
 10 \\
 +01 \\
 \hline
 11 \quad (1 + 0 = 1. \text{ No carry.})
 \end{array}$$

Exercise 1.4: Compute $5 + 10$ in binary. Verify that the result is 15.

Some additional examples of binary addition:

$$\begin{array}{r}
 0010 \quad (2) \\
 +0001 \quad (1) \\
 \hline
 =0011 \quad (3)
 \end{array}
 \qquad
 \begin{array}{r}
 0011 \quad (3) \\
 +0001 \quad (1) \\
 \hline
 =0100 \quad (4)
 \end{array}$$

This last example illustrates the role of the carry.

Looking at the right-most bits: $1 + 1 = (1) 0$

A carry of 1 is generated, which must be added to the next bits:

$$\begin{array}{r}
 001 - \text{column 0 has just been added} \\
 +000 - \\
 + \quad 1 \quad (\text{carry}) \\
 \hline
 = \quad (1)0 - \text{where (1) indicates a new} \\
 \qquad \qquad \qquad \text{carry into column 2.}
 \end{array}$$

The final result is: 0100

Another example:

$$\begin{array}{r}
 0111 \quad (7) \\
 +0011 \quad + (3) \\
 \hline
 1010 \quad =(10)
 \end{array}$$

In this example, a carry is again generated, up to the left-most column.

Exercise 1.5: Compute the result of:

$$\begin{array}{r}
 1111 \\
 +0001 \\
 \hline
 =?
 \end{array}$$

Does the result hold in four bits?

With eight bits, it is therefore possible to represent directly the numbers “00000000” to “11111111,” i.e., “0” to “255”. Two obstacles should be visible immediately. First, we are only representing positive numbers. Second, the magnitude of these numbers is limited to 255 if we use only eight bits. Let us address each of these problems in turn.

Signed Binary

In a signed binary representation, the left-most bit is used to indicate the sign of the number. Traditionally, “0” is used to denote a *positive* number while “1” is used to denote a *negative* number. Now “11111111” will represent -127, while “01111111” will represent +127. We can now represent positive and negative numbers, but we have reduced the maximum magnitude of these numbers to 127.

Example: “0000 0001” represents +1 (the leading “0” is “+”, followed by “000 0001” = 1).

“1000 0001” is -1 (the leading “1” is “-”).

Exercise 1.6: What is the representation of “-5” in signed binary?

Let us now address the *magnitude* problem: in order to represent larger numbers, it will be necessary to use a larger number of bits. For example, if we use sixteen bits (two bytes) to represent numbers, we will be able to represent numbers from -32K to +32K in signed binary (1K in computer jargon represents 1,024). Bit 15 is used for the sign, and the remaining 15 bits (bit 14 to bit 0) are used for the magnitude: $2^{15} = 32K$. If this magnitude is still too small, we will use 3 bytes or more. If we wish to represent large integers, it will be necessary to use a larger number of bytes internally to represent them. This is why most simple BASICs, and other languages, provide only a limited precision for integers. This way, they can use a shorter internal format for the numbers which they manipulate. Better versions of BASIC, or of these other languages, provide a larger number of significant decimal digits at the expense of a large number of bytes for each number.

Now let us solve another problem, the one of speed efficiency. We are going to attempt performing an addition in the signed

binary representation which we have introduced. Let us add “-5” and “+7”.

+7 is represented by 00000111

-5 is represented by 10000101

The binary sum is:
$$\begin{array}{r} 00000111 \\ 10000101 \\ \hline 10001100 \end{array}$$
, or -12

This is not the correct result. The correct result should be +2. In order to use this representation, special actions must be taken, depending on the sign. This results in increased complexity and reduced performance. In other words, the binary addition of signed numbers does not “work correctly.” This is annoying. Clearly, the computer must not only represent information, but also perform arithmetic on it.

The solution to this problem is called the *two's complement* representation, which will be used instead of the *signed binary* representation. In order to introduce two's complement let us first introduce an intermediate step: *one's complement*.

One's Complement

In the one's complement representation, all positive integers are represented in their correct binary format. For example “+3” is represented as usual by 00000011. However, its complement “-3” is obtained by complementing every bit in the original representation. Each 0 is transformed into a 1 and each 1 is transformed into a 0. In our example, the one's complement representation of “-3” will be 11111100.

Another example:

+2 is 00000010

-2 is 11111101

Note that, in this representation, positive numbers start with a “0” on the left, and negative ones with a “1” on the left.

Exercise 1.7: The representation of “+6” is “00000110”. What is the representation of “-6” in one's complement?

As a test, let us add minus 4 and plus 6:

$$\begin{array}{r} -4 \text{ is } 11111011 \\ +6 \text{ is } 00000110 \\ \hline \end{array}$$

the sum is: (1) 00000001 where (1) indicates a carry

The "correct result" should be "2", or "00000010".

Let us try again:

$$\begin{array}{r} -3 \text{ is } 11111100 \\ -2 \text{ is } 11111101 \\ \hline \end{array}$$

The sum is: (1) 11111001

or "-6," plus a carry. The correct result should be "-." The representation of "-5" is 11111010. It did not work.

This representation does represent positive and negative numbers. However the result of an ordinary addition does not always come out "correctly." We will use still another representation. It is evolved from the one's complement and is called the two's complement representation.

Two's Complement Representation

In the two's complement representation, positive numbers are still represented, as usual, in signed binary, just like in one's complement. The difference lies in the representation of *negative numbers*. A negative number represented in two's complement is obtained by first computing the one's complement, and then *adding one*. Let us examine this in an example:

+3 is represented in signed binary by 00000011. Its one's complement representation is 11111100. The two's complement is obtained by adding one. It is 11111101.

Let us try an addition:

$$\begin{array}{r} (3) \quad 00000011 \\ +(5) \quad +00000101 \\ \hline = (8) \quad = 00001000 \end{array}$$

The result is correct.

Let us try a subtraction:

$$\begin{array}{r} (3) \quad 00000011 \\ (-5) \quad +11111011 \\ \hline =11111110 \end{array}$$

Let us identify the result by computing the two's complement:

the one's complement of 11111110 is 00000001
 Adding 1 + 1

therefore the two's complement is 00000010 or +2

Our result above, "11111110" represents "-2". It is correct.

We have now tried addition and subtraction, and the results were correct (ignoring the carry). It seems that two's complement works!

Exercise 1.8: What is the two's complement representation of "+127"?

Exercise 1.9: What is the two's complement representation of "-128"?

Let us now add +4 and -3 (the subtraction is performed by adding the two's complement):

$$\begin{array}{r} +4 \text{ is } 00000100 \\ -3 \text{ is } 11111101 \\ \hline \end{array}$$

The result is: (1) 00000001

If we ignore the carry, the result is 00000001, i.e., "1" in decimal. This is the correct result. Without giving the complete mathematical proof, let us simply state that this representation does work. In two's complement, it is possible to add or subtract signed numbers regardless of the sign. Using the usual rules of binary addition, the result comes out correctly, including the sign. The carry is ignored. This is a very significant advantage. If it were not the case, one would have to correct the result for sign every time, causing a much slower addition or subtraction time.

For the sake of completeness, let us state that two's complement is simply the most convenient representation to use for simpler processors such as microprocessors. On complex processors, other representations may be used. For example, one's complement may be used, but it requires special circuitry to "correct the result."

PROGRAMMING THE Z80

From this point on, all signed integers will implicitly be represented internally in two's complement notation. See Fig. 1.3 for a table of two's complement numbers.

Exercise 1.10: *What are the smallest and the largest numbers which one may represent in two's complement notation, using only one byte?*

Exercise 1.11: *Compute the two's complement of 20. Then compute the two's complement of your result. Do you find 20 again?*

The following examples will serve to demonstrate the rules of two's complement. In particular, C denotes a possible carry (or borrow) condition. (It is bit 8 of the result.)

V denotes a two's complement overflow, i.e., when the sign of the result is changed "accidentally" because the numbers are too large. It is an essentially internal carry from bit 6 into bit 7 (the sign bit). This will be clarified below.

Let us now demonstrate the role of the carry "C" and the overflow "V".

The Carry C

Here is an example of a carry:

$$\begin{array}{r} (128) & 10000000 \\ +(129) & +10000001 \\ \hline (257) = (1) & 00000001 \end{array}$$

where (1) indicates a carry.

The result requires a ninth bit (bit "8", since the right-most bit is "0"). It is the carry bit.

If we assume that the carry is the ninth bit of the result, we recognize the result as being $100000001 = 257$.

However, the carry must be recognized and handled with care. Inside the microprocessor, the registers used to hold information are generally only eight-bit wide. When storing the result, only bits 0 to 7 will be preserved.

A carry, therefore, always requires special action: it must be detected by special instructions, then processed. Processing the carry means either storing it somewhere (with a special instruction), or ignoring it, or deciding that it is an error (if the largest authorized result is "11111111").

+	2's complement code	-	2's complement code
+ 127	01111111	- 128	10000000
+ 126	01111110	- 127	10000001
+ 125	01111101	- 126	10000010
...		- 125	10000011
		...	
+ 65	01000001	- 65	10111111
+ 64	01000000	- 64	11000000
+ 63	00111111	- 63	11000001
...		...	
+ 33	00100001	- 33	11011111
+ 32	00100000	- 32	11100000
+ 31	00011111	- 31	11100001
...		...	
+ 17	00010001	- 17	11101111
+ 16	00010000	- 16	11110000
+ 15	00001111	- 15	11110001
+ 14	00001110	- 14	11110010
+ 13	00001101	- 13	11110011
+ 12	00001100	- 12	11110100
+ 11	00001011	- 11	11110101
+ 10	00001010	- 10	11110110
+ 9	00001001	- 9	11110111
+ 8	00001000	- 8	11111000
+ 7	00000111	- 7	11111001
+ 6	00000110	- 6	11111010
+ 5	00000101	- 5	11111011
+ 4	00000100	- 4	11111100
+ 3	00000011	- 3	11111101
+ 2	00000010	- 2	11111110
+ 1	00000001	- 1	11111111
+ 0	00000000		

Fig. 1.3: 2's Complement Table

Overflow V

Here is an example of overflow:

$$\begin{array}{r}
 \text{bit 6} \\
 \text{bit 7} \\
 \downarrow \\
 \begin{array}{r}
 01000000 \quad (64) \\
 + 01000001 \quad +(65) \\
 \hline
 = 10000001 \quad =(-127)
 \end{array}
 \end{array}$$

An internal carry has been generated from bit 6 into bit 7. This is called an overflow.

The result is now negative, "by accident." This situation must be detected, so that it can be corrected.

Let us examine another situation:

$$\begin{array}{r}
 11111111 \quad (-1) \\
 + 11111111 \quad +(-1) \\
 \hline
 =(1) \quad 11111110 \quad =(-2) \\
 \downarrow \\
 \text{carry}
 \end{array}$$

In this case, an internal carry has been generated from bit 6 into bit 7, and also from bit 7 into bit 8 (the formal "Carry" C we have examined in the preceding section). The rules of two's complement arithmetic specify that this carry should be ignored. The result is then correct.

This is because the carry from bit 6 into bit 7 did not change the sign bit.

This is not an *overflow* condition. When operating on negative numbers, the overflow is not simply a carry from bit 6 into bit 7. Let us examine one more example.

$$\begin{array}{r}
 11000000 \quad (-64) \\
 + 10111111 \quad (-65) \\
 \hline
 =(1) \quad 01111111 \quad (+127) \\
 \downarrow \\
 \text{carry}
 \end{array}$$

This time, there has been no internal carry from bit 6 into bit 7, but there has been an external carry. The result is incorrect, as bit 7 has been changed. An overflow condition should be indicated.

Overflow will occur in four situations:

- 1—adding large positive numbers
- 2—adding large negative numbers
- 3—subtracting a large positive number from a large negative number
- 4—subtracting a large negative number from a large positive number.

Let us now improve our definition of the overflow:

Technically, the overflow indicator, a special bit reserved for this purpose, and called a “flag,” will be set when there is a carry from bit 6 into bit 7 and no external carry, or else when there is no carry from bit 6 into bit 7 but there is an external carry. This indicates that bit 7, i.e., the sign of the result, has been accidentally changed. For the technically-minded reader, the overflow flag is set by Exclusive ORing the carry-in and carry-out of bit 7 (the sign bit). Practically every microprocessor is supplied with a special overflow flag to automatically detect this condition, which requires corrective action.

Overflow indicates that the result of an addition or a subtraction requires more bits than are available in the standard eight-bit register used to contain the result.

The Carry and the Overflow

The carry and the overflow bits are called “flags.” They are provided in every microprocessor, and in the next chapter we will learn to use them for effective programming. These two indicators are located in a special register called the flags or “status” register. This register also contains additional indicators whose function will be clarified in Chapter 4.

Examples

Let us now illustrate the operation of the carry and the overflow in actual examples. In each example, the symbol V denotes the overflow, and C the carry.

If there has been no overflow, $V = 0$. If there has been an overflow, $V = 1$ (same for the carry C). Remember that the rules of two's complement specify that the carry be ignored. (The mathematical proof is not supplied here.)

PROGRAMMING THE Z80

Positive-Positive

$$\begin{array}{r} 00000110 \quad (+6) \\ + \quad 00001000 \quad (+8) \\ \hline = \quad 00001110 \quad (+14) \quad V:0 \quad C:0 \end{array}$$

(CORRECT)

Positive-Positive with Overflow

$$\begin{array}{r} 01111111 \quad (+127) \\ + \quad 00000001 \quad (+1) \\ \hline = \quad 10000000 \quad (-128) \quad V:1 \quad C:0 \end{array}$$

The above is invalid because an overflow has occurred.

(ERROR)

Positive-Negative (result positive)

$$\begin{array}{r} 00000100 \quad (+4) \\ + \quad 11111110 \quad (-2) \\ \hline =(1)00000010 \quad (+2) \quad V:0 \quad C:1 \text{ (disregard)} \end{array}$$

(CORRECT)

Positive-Negative (result negative)

$$\begin{array}{r} 00000010 \quad (+2) \\ + \quad 11111100 \quad (-4) \\ \hline = \quad 11111110 \quad (-2) \quad V:0 \quad C:0 \end{array}$$

(CORRECT)

Negative-Negative

$$\begin{array}{r} 11111110 \quad (-2) \\ + \quad 11111100 \quad (-4) \\ \hline =(1)11111010 \quad (-6) \quad V:0 \quad C:1 \text{ (disregard)} \end{array}$$

(CORRECT)

Negative-Negative with Overflow

$$\begin{array}{r} 10000001 \quad (-127) \\ + \quad 11000010 \quad (-62) \\ \hline =(1)01000011 \quad (67) \quad V:1 \quad C:1 \end{array}$$

(ERROR)

This time an “underflow” has occurred, by adding two large negative numbers. The result would be -189 , which is too large to reside in eight bits.

Exercise 1.12: Complete the following additions. Indicate the result, the carry C, the overflow V, and whether the result is correct or not:

10111111 (____)	11111010 (____)
+11000001 (____)	+11111001 (____)
<hr/>	
= _____	= _____
V: _____ C: _____	V: _____ C: _____
<input type="checkbox"/> CORRECT	<input type="checkbox"/> CORRECT
_____	_____
□ ERROR	□ ERROR

00010000 (____)	01111110 (____)
+01000000 (____)	+00101010 (____)
<hr/>	
= _____	= _____
V: _____ C: _____	V: _____ C: _____
<input type="checkbox"/> CORRECT	<input type="checkbox"/> CORRECT
_____	_____
□ ERROR	□ ERROR

Exercise 1.13: Can you show an example of overflow when adding a positive and a negative number? Why?

Fixed Format Representation

Now we know how to represent signed integers. However, we have not yet resolved the problem of magnitude. If we want to represent larger integers, we will need several bytes. In order to perform arithmetic operations efficiently, it is necessary to use a fixed number of bytes rather than a variable one. Therefore, once the number of bytes is chosen, the maximum magnitude of the number which can be represented is fixed.

Exercise 1.14: What are the largest and the smallest numbers which may be represented in two bytes using two's complement?

The Magnitude Problem

When adding numbers we have restricted ourselves to eight bits because the processor we will use operates internally on eight bits at a time. However, this restricts us to the numbers in the range -128 to $+127$. Clearly, this is not sufficient for many applications.

Multiple precision will be used to increase the number of digits which can be represented. A two-, three-, or N-byte format may

then be used. For example, let us examine a 16-bit, "double-precision" format:

00000000	00000000	is "0"
00000000	00000001	is "1"
...		
01111111	11111111	is "32767"
11111111	11111111	is "-1"
11111111	11111110	is "-2"

Exercise 1.15: What is the largest negative integer which can be represented in a two's complement triple-precision format?

However, this method will result in disadvantages. When adding two numbers, for example, we will generally have to add them eight bits at a time. This will be explained in Chapter 3 (Basic Programming Techniques). It results in slower processing. Also, this representation uses 16 bits for any number, even if it could be represented with only eight bits. It is, therefore, common to use 16 or perhaps 32 bits, but seldom more.

Let us consider the following important point: whatever the number of bits N chosen for the two's complement representation, it is fixed. If any result or intermediate computation should generate a number requiring more than N bits, some bits will be lost. The program normally retains the N left-most bits (the most significant) and drops the low-order ones. This is called truncating the result.

Here is an example in the decimal system, using a six digit representation:

$$\begin{array}{r}
 123456 \\
 \times \quad 1.2 \\
 \hline
 246912 \\
 123456 \\
 \hline
 = 148147.2
 \end{array}$$

The result requires 7 digits! The "2" after the decimal point will be dropped and the final result will be 148147. It has been truncated. Usually, as long as the position of the decimal point is not lost, this method is used to extend the range of the operations which may be performed, at the expense of precision.

The problem is the same in binary. The details of a binary multi-

plication will be shown in Chapter 4.

This fixed-format representation may cause a loss of precision, but it may be sufficient for usual computations or mathematical operations.

Unfortunately, in the case of accounting, no loss of precision is tolerable. For example, if a customer rings up a large total on a cash register, it would not be acceptable to have a five figure amount to pay, which would be approximated to the dollar. Another representation must be used wherever precision in the result is essential. The solution normally used is *BCD*, or binary-coded decimal.

BCD Representation

The principle used in representing numbers in BCD is to encode each decimal digit separately, and to use as many bits as necessary to represent the complete number exactly. In order to encode each of the digits from 0 through 9, four bits are necessary. Three bits would only supply eight combinations, and can therefore not encode the ten digits. Four bits allow sixteen combinations and are therefore sufficient to encode the digits "0" through "9". It can also be noted that six of the possible codes will not be used in the BCD representation (see Fig. 1-4). This will result later on in a potential problem during additions and subtractions, which we will have to solve.

CODE	BCD SYMBOL	CODE	BCD SYMBOL
0000	0	1000	8
0001	1	1001	9
0010	2	1010	unused
0011	3	1011	unused
0100	4	1100	unused
0101	5	1101	unused
0110	6	1110	unused
0111	7	1111	unused

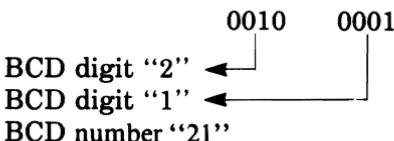
Fig. 1.4: BCD Table

PROGRAMMING THE Z80

Since only four bits are needed to encode a BCD digit, two BCD digits may be encoded in every byte. This is called "*packed BCD*."

As an example, "00000000" will be "00" in BCD. "10011001" will be "99".

A BCD code is read as follows:

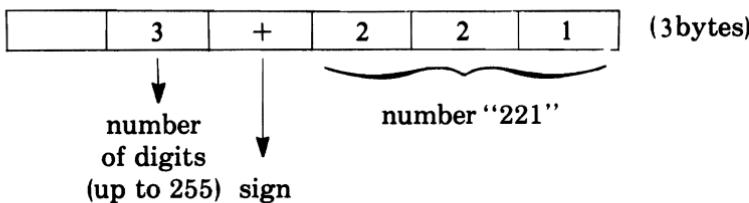


Exercise 1.16: What is the BCD representation for "29"? "91"?

Exercise 1.17: Is "10100000" a valid BCD representation? Why?

As many bytes as necessary will be used to represent all BCD digits. Typically, one or more nibbles will be used at the beginning of the representation to indicate the total number of nibbles, i.e., the total number of BCD digits used. Another nibble or byte will be used to denote the position of the decimal point. However, conventions may vary.

Here is an example of a representation for multibyte BCD integers:



This represents +221

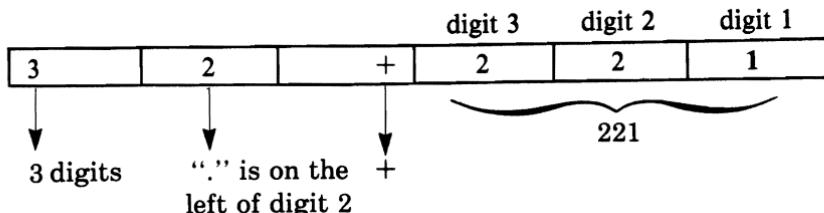
(The sign may be represented by 0000 for +, and 0001 for -, for example.)

Exercise 1.18: Using the same convention, represent "-23123". Show it in BCD format, as above, then in binary.

Exercise 1.19: Show the BCD for "222" and "111", then for the result of 222×111 . (Compute the result by hand, then show it in the above representation.)

The BCD representation can easily accommodate decimal numbers.

For example, +2.21 may be represented by:



The advantage of BCD is that it yields absolutely correct results. Its disadvantage is that it uses a large amount of memory and results in slow arithmetic operations. This is acceptable only in an accounting environment and is normally not used in other cases.

Exercise 1.20: How many bits are required to encode "9999" in BCD? And in two's complement?

We have now solved the problems associated with the representation of integers, signed integers and even large integers. We have even already presented one possible method of representing decimal numbers, with BCD representation. Let us now examine the problem of representing decimal numbers in a fixed length format.

Floating-Point Representation

The basic principle is that decimal numbers must be represented with a fixed format. In order not to waste bits, the representation will *normalize* all the numbers.

For example, "0.000123" wastes three zeros on the left of the number, which have no meaning except to indicate the position of the decimal point. Normalizing this number results in $.123 \times 10^{-3}$. ".123" is called a *normalized mantissa*, "-3" is called the *exponent*. We have normalized this number by eliminating all the meaningless zeros on the left of it and adjusting the exponent.

Let us consider another example:

22.1 is normalized as $.221 \times 10^2$

or $M \times 10^E$ where M is the mantissa, and E is the exponent.

PROGRAMMING THE Z80

It can be readily seen that a normalized number is characterized by a mantissa less than 1 and greater or equal to .1 in all cases where the number is not zero. In other words, this can be represented mathematically by:

$$.1 \leq M < 1 \text{ or } 10^{-1} \leq M < 10^0$$

Similarly, in the binary representation:

$$2^{-1} \leq M < 2^0 \text{ (or } .5 \leq M < 1)$$

Where M is the absolute value of the mantissa (disregarding the sign).

For example:

111.01 is normalized as: .11101 $\times 2^3$.

The mantissa is 11101.

The exponent is 3.

Now that we have defined the principle of the representation, let us examine the actual format. A typical floating-point representation appears below.

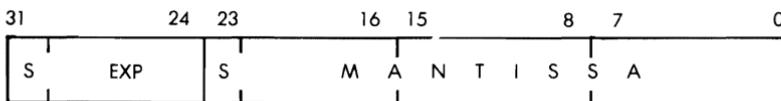


Fig. 1.5: Typical Floating-Point Representation

In the representation used in this example, four bytes are used for a total of 32 bits. The first byte on the left of the illustration is used to represent the exponent. Both the exponent and the mantissa will be represented in two's complement. As a result, the maximum exponent will be -128. "S" in Fig. 1-5 denotes the sign bit.

Three bytes are used to represent the mantissa. Since the first bit in the two's complement representation indicates the sign, this leaves 23 bits for the representation of the magnitude of the mantissa.

Exercise 1.21: How many decimal digits can the mantissa represent with the 23 bits?

This is only one example of a floating point representation. It is possible to use only three bytes, or it is possible to use more. The four-byte representation proposed above is just a common one which represents a reasonable compromise in terms of accuracy, magnitude of numbers, storage utilization, and efficiency in arithmetic operation.

We have now explored the problems associated with the representation of numbers and we know how to represent them in integer form, with a sign, or in decimal form. Let us now examine how to represent alphanumeric data internally.

Representing Alphanumeric Data

The representation of alphanumeric data, i.e. characters, is completely straightforward: all characters are encoded in an eight-bit code. Only two codes are in general use in the computer world, the ASCII Code, and the EBCDIC Code. ASCII stands for "American Standard Code for Information Interchange," and is universally used in the world of microprocessors. EBCDIC is a variation of ASCII used by IBM, and therefore not used in the microcomputer world unless one interfaces to an IBM terminal.

Let us briefly examine the ASCII encoding. We must encode 26 letters of the alphabet for both upper and lower case, plus 10 numeric symbols, plus perhaps 20 additional special symbols. This can be easily accomplished with 7 bits, which allow 128 possible codes. (See Fig.1-6.) All characters are therefore encoded in 7 bits. The eighth bit, when it is used, is the *parity bit*. Parity is a technique for verifying that the contents of a byte have not been accidentally changed. The number of 1's in the byte is counted and the eighth bit is set to one if the count was odd, thus making the total even. This is called even parity. One can also use odd parity, i.e. writing the eighth bit (the left-most) so that the total number of 1's in the byte is odd.

Example: let us compute the parity bit for "0010011" using even parity. The number of 1's is 3. The parity bit must therefore be a 1 so that the total number of bits is 4, i.e. even. The result is 10010011, where the leading 1 is the parity bit and 0010011 identifies the character.

PROGRAMMING THE Z80

The table of 7-bit ASCII codes is shown in Fig. 1-6. In practice, it is used “as is,” i.e. without parity, by adding a 0 in the left-most position, or else with parity, by adding the appropriate extra bit on the left.

Exercise 1.22: Compute the 8-bit representation of the digits “0” through “9”, using even parity. (This code will be used in application examples of Chapter 8.)

Exercise 1.23: Same for the letters “A” through “F”.

Exercise 1.24: Using a non-parity ASCII code (where the left-most bit is “0”), indicate the binary contents of the 4 characters below:

“A”
“?”
“3”
“b”

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	-	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	“	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	\$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	,	7	G	W	g	w
8	1000	BS	CAN	(8	H	X	h	x
9	1001	HT	EM)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[k	{
C	1100	FF	FS	,	<	L	\	l	--
D	1101	CR	GS	-	=	M]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

Fig. 1.6: ASCII Conversion Table
(see Appendix B for abbreviations).

In specialized situations such as telecommunications, other codings may be used such as error-correcting codes. However they are beyond the scope of this book.

We have examined the usual representations for both program and data inside the computer. Let us now examine the possible external representations.

EXTERNAL REPRESENTATION OF INFORMATION

The external representation refers to the way information is presented to the *user*, i.e. generally to the programmer. Information may be presented externally in essentially three formats: binary, octal or hexadecimal and symbolic.

1. Binary

It has been seen that information is stored internally in *bytes*, which are sequences of eight *bits* (0's or 1's). It is sometimes desirable to display this internal information directly in its binary format and this is called *binary representation*. One simple example is provided by Light Emitting Diodes (LEDs) which are essentially miniature lights, on the front panel of the microcomputer. In the case of an eight-bit microprocessor, a front panel will typically be equipped with eight LEDs to display the contents of any internal register. (A register is used to hold eight bits of information and will be described in Chapter 2). A lighted LED indicates a one. A zero is indicated by an LED which is not lighted. Such a binary representation may be used for the fine debugging of a complex program, especially if it involves input/output, but is naturally impractical at the human level. This is because in most cases, one likes to look at information in symbolic form. Thus "9" is much easier to understand or remember than "1001". More convenient representations have been devised, which improve the person-machine interface.

2. Octal and Hexadecimal

"Octal" and "hexadecimal" encode respectively three and four binary bits into a unique symbol. In the octal system, any combination of three binary bits is represented by a number between 0 and 7.

"Octal" is a format using three bits, where each combination of three bits is represented by a symbol between 0 and 7:

binary	octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Fig. 1.7: Octal Symbols

For example, “00 100 100” binary is represented by:

$\begin{matrix} \downarrow & \downarrow & \downarrow \\ 0 & 4 & 4 \end{matrix}$

or “044” in octal.

Another example: 11 111 111 is:

$\begin{matrix} \downarrow & \downarrow & \downarrow \\ 3 & 7 & 7 \end{matrix}$

or “377” in octal.

Conversely, the octal “211” represents:

010 001 001

or “10001001” binary.

Octal has traditionally been used on older computers which were employing various numbers of bits ranging from 8 to perhaps 64. More recently, with the dominance of eight-bit microprocessors, the eight-bit format has become the standard, and another more practical representation is used. This is *hexadecimal*.

In the hexadecimal representation, a group of four bits is encoded as one hexadecimal digit. Hexadecimal digits are represented by the symbols from 0 to 9, and by the letters A, B, C, D, E, F. For example, “0000” is represented by “0”, “0001” is represented by “1” and “1111” is represented by the letter “F” (see Fig. 1-8).

DECIMAL	BINARY	HEX	OCTAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Fig. 1.8: Hexadecimal Codes

Example: 1010 0001 in binary is represented by
A 1 in hexadecimal.

Exercise 1.25: What is the hexadecimal representation of “10101010?”

Exercise 1.26: Conversely, what is the binary equivalent of “FA” hexadecimal?

Exercise 1.27: What is the octal of “01000001”?

Hexadecimal offers the advantage of encoding eight bits into only two digits. This is easier to visualize or memorize and faster to type into a computer than its binary equivalent. Therefore, on most new microcomputers, hexadecimal is the preferred method of representation for groups of bits.

Naturally, whenever the information present in the memory has a meaning, such as representing text or numbers, hexadecimal is not convenient for representing the meaning of this information when it is brought out for use by humans.

Symbolic Representation

Symbolic representation refers to the external representation of information in actual symbolic form. For example, decimal numbers are represented as decimal numbers, and not as sequences of hexadecimal symbols or bits. Similarly, text is represented as such. Naturally, symbolic representation is most practical to the user. It is used whenever an appropriate display device is available, such as a CRT display or a printer. (A CRT display is a television-type screen used to display text or graphics.) Unfortunately, in smaller systems such as one-board microcomputers, it is uneconomical to provide such displays, and the user is restricted to hexadecimal communication with the computer.

Summary of External Representations

Symbolic representation of information is the most desirable since it is the most natural for a human user. However, it requires an expensive interface in the form of an alphanumeric keyboard, plus a printer or a CRT display. For this reason, it may not be

available on the less expensive systems. An alternative type of representation is then used, and in this case hexadecimal is the dominant representation. Only in rare cases relating to fine de-bugging at the hardware or the software level is the binary representation used. *Binary* directly displays the contents of registers of memory in binary format.

(The utility of a direct binary display on a front panel has always been the subject of a heated emotional controversy, which will not be debated here.)

We have seen how to represent information internally and externally. We will now examine the actual microprocessor which will manipulate this information.

Additional Exercises

Exercise 1.28: What is the advantage of two's complement over other representations used to represent signed numbers?

Exercise 1.29: How would you represent “1024” in direct binary? Signed binary? Two's complement?

Exercise 1.30: What is the V-bit? Should the programmer test it after an addition or subtraction?

Exercise 1.31: Compute the two's complement of “+16”, “+17”, “+18”, “-16”, “-17”, “-18”.

Exercise 1.32: Show the hexadecimal representation of the following text, which has been stored internally in ASCII format, with no parity: = “MESSAGE”.

2

Z80 HARDWARE ORGANIZATION

INTRODUCTION

In order to program at an elementary level, it is not necessary to understand in detail the internal structure of the processor that one is using. However, in order to do efficient programming, such an understanding is required. The purpose of this chapter is to present the basic hardware concepts necessary for understanding the operation of the Z80 system. The complete microcomputer system includes not only the microprocessor unit (here the Z80), but also other components. This chapter presents the Z80 proper, while the other devices (mainly input/output) will be presented in a separate chapter (Chapter 7).

We will review here the basic architecture of the microcomputer system, then study more closely the internal organization of the Z80. We will examine, in particular, the various registers. We will then study the program execution and sequencing mechanism. From a hardware standpoint, this chapter is only a simplified presentation. The reader interested in gaining detailed understanding is referred to our book ref. C201 ("Microprocessors," by the same author).

The Z80 was designed as a replacement for the Intel 8080, and to offer additional capabilities. A number of references will be made in this chapter to the 8080 design.

SYSTEM ARCHITECTURE

The architecture of the microcomputer system appears in Figure 2.1. The microprocessor unit (MPU), which will be a Z80 here, appears on the left of the illustration. It implements the functions of a *central-processing unit* (CPU) within one chip: it includes an *arithmetic-logical unit* (ALU), plus its internal registers, and a *control unit* (CU), in

charge of sequencing the system. Its operation will be explained in this chapter.

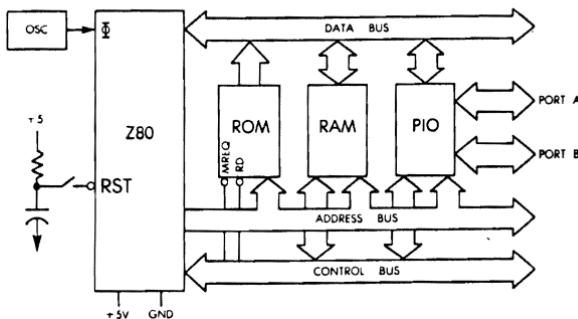


Fig. 2.1: Standard Z80 System

The MPU creates three *buses*: an 8-bit bidirectional *data bus*, which appears at the top of the illustration, a 16-bit unidirectional *address bus*, which appears at the bottom of the illustration. Let us describe the function of each of the buses.

The *data bus* carries the data being exchanged by the various elements of the system. Typically, it will carry data from the memory to the MPU or from the MPU to the memory or from the MPU to an input/output chip. (An input/output chip is a component in charge of communicating with an external device.)

The *address bus* carries an address generated by the MPU, which will select one internal register within one of the chips attached to the system. This address specifies the source, or the destination, of the data which will transit along the data bus.

The *control bus* carries the various synchronization signals required by the system.

Having described the purpose of buses, let us now connect the additional components required for a complete system.

Every MPU requires a precise timing reference, which is supplied by a *clock* and a *crystal*. In most "older" microprocessors, the clock-oscillator is external to the MPU and requires an extra chip. In most recent microprocessors, the clock-oscillator is usually incorporated within the MPU. The quartz crystal, however, because of its bulk, is always exter-

PROGRAMMING THE Z80

nal to the system. The crystal and the clock appear on the left of the MPU box in Figure 2.1.

Let us now turn our attention to the other elements of the system. Going from left to right on the illustration, we distinguish:

The *ROM* is the *read-only memory* and contains the *program* for the system. The advantage of the ROM memory is that its contents are permanent and do not disappear whenever the system is turned off. The ROM, therefore, always contains a *bootstrap* or a *monitor* program (their function will be explained later) to permit initial system operation. In a process-control environment, nearly all the programs will reside in ROM, as they will probably never be changed. In such a case, the industrial user has to protect the system against power failures; programs must not be volatile. They must be in ROM.

However, in a hobbyist environment, or in a program-development environment (when the programmer tests his program), most of the programs will reside in RAM so that they can be easily changed. Later, they may remain in RAM, or be transferred into ROM, if desired. RAM, however, is volatile. Its contents are lost when power is turned off.

The *RAM (random-access memory)* is the read/write memory for the system. In the case of a control system, the amount of RAM will typically be small (for data only). On the other hand, in a program-development environment, the amount of RAM will be large, as it will contain programs plus development software. All RAM contents must be loaded prior to use from an external device.

Finally the system will contain one or more interface chips so that it may communicate with the external world. The most frequently used interface chip is the PIO or *parallel input/output* chip. It is the one shown on the illustration. This PIO, like all other chips in the system, connects to all three buses and provides at least two 8-bit ports for communication with the outside world. For more details on how an actual PIO works, refer to book C201 or, for specifics of the Z80 system, refer to Chapter 7 (Input/Output Devices).

All the chips are connected to all three buses, including the control bus.

The functional modules which have been described need not necessarily reside on a single LSI chip. In fact, we could use *combination chips*, which may include both PIO and a limited amount of ROM or RAM.

Still more components will be required to build a real system. In par-

ticular, the buses usually need to be *buffered*. Also, *decoding logic* may be used for the memory RAM chips, and, finally, some signals may need to be amplified by *drivers*. These auxiliary circuits will not be described here as they are not relevant to programming. The reader interested in specific assembly and interfacing techniques is referred to book C207 "Microprocessor Interfacing Techniques."

INSIDE A MICROPROCESSOR

The large majority of all microprocessor chips on the market today implement the same architecture. This "standard" architecture will be described here. It is shown in Figure 2.2. The modules of this standard microprocessor will now be detailed, from right to left.

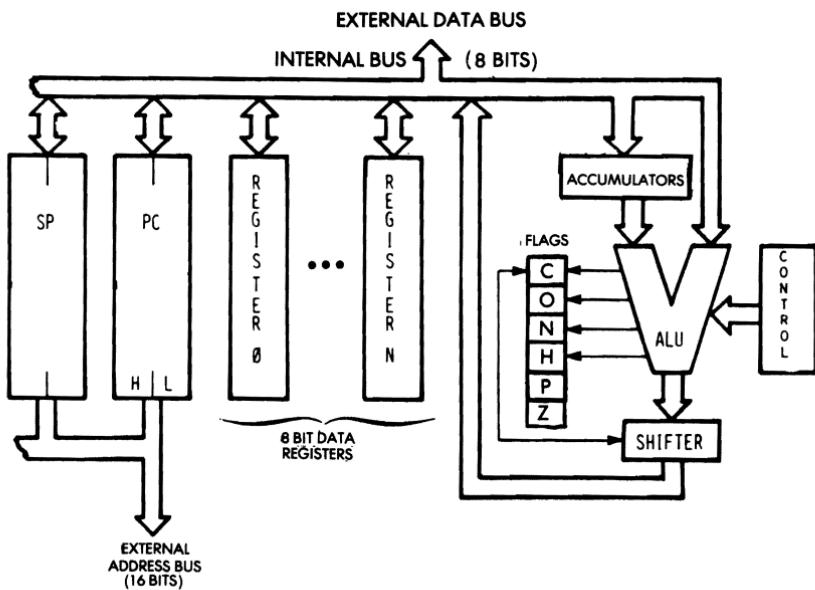


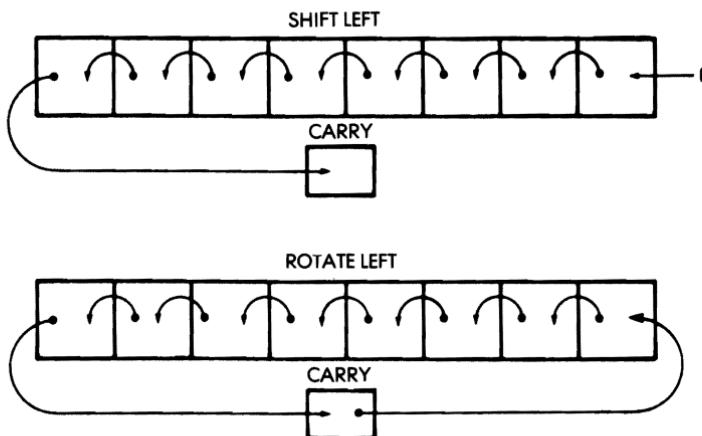
Fig. 2.2: "Standard" Microprocessor Architecture

The *control box* on the right represents the control unit which synchronizes the entire system. Its role will be clarified within the remainder of this chapter.

The *ALU* performs arithmetic and logic operations. A special register equips one of the inputs of the ALU, the left input here. It is called the accumulator. (Several accumulators may be provided.) The accumulator may be referenced both as input and output (source and destination) within the same instruction.

The ALU must also provide *shift* and *rotate* facilities.

A shift operation consists of moving the contents of a byte by one or more positions to the left or to the right. This is illustrated in Figure 2.3. Each bit has been moved to the left by one position. The details of shifts and rotations will be presented in the next chapter.



Note: Some Shift and Rotate instructions do not include the Carry.

Fig. 2.3: Shift and Rotate

The shifter may be on the ALU output, as illustrated in Figure 2.2, or may be on the accumulator input.

To the left of the ALU, the *flags* or *status register* appear. Their role is to store exceptional conditions within the microprocessor. The contents of the flags register may be tested by specialized instructions, or may be read on the internal data bus. A *conditional* instruction will cause the execution of a new program, depending on the value of one of these bits.

The role of the status bits in the Z80 will be examined later in this chapter.

Setting Flags

Most of the instructions executed by the processor will modify some or all of the flags. It is important to always refer to the chart provided by the manufacturer listing which bits will be modified by the instructions. This is essential in understanding the way a program is being executed. Such a chart for the Z80 is shown in Figure 4-17.

The Registers

Let us look now at Figure 2.2. On the left of the illustration, the registers of the microprocessor appear. Conceptually, one can distinguish the *general purpose registers* and the *address registers*.

The General-Purpose Registers

General-purpose registers must be provided in order for the ALU to manipulate data at high speed. Because of restrictions on the number of bits which it is reasonable to provide within an instruction, the number of (directly addressable) registers is usually limited to fewer than eight. Each of these registers is a set of eight flip-flops, connected to the bidirectional internal data bus. These eight bits can be transferred simultaneously to or from the data bus. The implementation of these registers in MOS flip-flops provides the fastest level of memory available, and their contents can be accessed within tens of nanoseconds.

Internal registers are usually labelled from 0 to n. The role of these registers is not defined in advance: they are said to be "general purpose." They may contain any data used by the program.

These general-purpose registers will normally be used to store eight-bit data. On some microprocessors, facilities exist to manipulate *two* of these registers at a time. They are then called "register pairs." This arrangement facilitates the storage of 16-bit quantities, whether data or addresses.

The Address Registers

Address registers are 16-bit registers intended for the storage of addresses. They are also often called *data counters* or *pointers*. They are double registers, i.e., two eight-bit registers. Their essential characteristic is to be connected to the address bus. The address registers create the address bus. The address bus appears on the left and the bottom part of the illustration in Figure 2.4.

The only way to load the contents of these 16-bit registers is via the data bus. Two transfers will be necessary along the data bus in order to transfer 16 bits. In order to differentiate between the lower half and the higher half of each register, they are usually labelled as L (low) or H (high), denoting bits 0 through 7, and 8 through 15 respectively. This label is used whenever it is necessary to differentiate the halves of these registers. At least two address registers are present within most microprocessors. “MUX” in Fig. 2.4 stands for multiplexer.

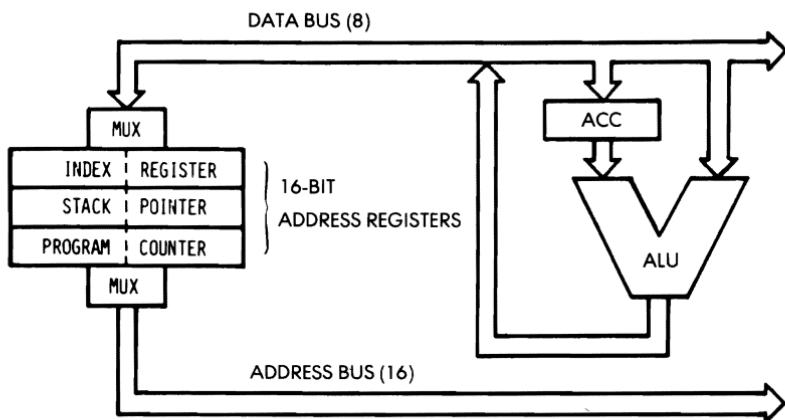


Fig. 2.4: The 16-bit Address Registers Create the Address Bus

Program Counter (PC)

The *program counter* must be present in any processor. It contains the address of the next instruction to be executed. The presence of the program counter is indispensable and fundamental to program execution. The mechanism of program execution and the automatic sequencing implemented with the program counter will be described in the next section. Briefly, execution of a program is normally sequential. In order to access the next instruction, it is necessary to bring it from the memory into the microprocessor. The contents of the PC will be deposited on the address bus, and transmitted towards the memory. The memory will then read the contents specified by this address and send back the corresponding word to the MPU. This is the instruction.

In a few exceptional microprocessors, such as the two-chip F8, there is no PC on the microprocessor. This does not mean that the system does not have a program counter. The PC happens to be implemented directly on the memory chip, for reasons of efficiency.

Stack Pointer (SP)

The *stack* has not been introduced yet and will be described in the next section. In most powerful, general-purpose microprocessors, the stack is implemented in "software," i.e., within the memory. In order to keep track of the top of this stack within the memory, a 16-bit register is dedicated to the *stack pointer* or *SP*. The SP contains the address of the top of the stack within the memory. It will be shown that the stack is indispensable for interrupts and for subroutines.

Index Register (IX)

Indexing is a memory-addressing facility which is not always provided in microprocessors. The various memory-addressing techniques will be described in Chapter 5. Indexing is a facility for accessing blocks of data in the memory with a single instruction. An *index register* will typically contain a displacement which will be automatically added to a base (or it might contain a base which would be added to a displacement). In short, indexing is used to access any word within a block of data.

The Stack

A *stack* is formally called an LIFO structure (last-in, first-out). A stack is a set of registers, or memory locations, allocated to this data structure. The essential characteristic of this structure is that it is a *chronological* structure. The first element introduced into the stack is always at the bottom of the stack. The element most recently deposited in the stack is on the top of the stack. The analogy can be drawn with a stack of plates on a restaurant counter. There is a hole in the counter with a spring in the bottom. Plates are piled up in the hole. With this organization, it is guaranteed that the plate which has been put first in the stack (the oldest) is always at the bottom. The one that has been placed most recently on the stack is the one which is on top of it. This example also illustrates another characteristic of the stack. In normal use, a stack is only accessible via two instructions: "push" and "pop" (or "pull"). The *push* operation results in depositing one element on

top of the stack (two in the case of the Z80). The *pull* operation consists of removing one element from the stack. In the case of a microprocessor, it is the *accumulator* that will be deposited on top of the stack. The *pop* will result in a transfer of the top element of the stack into the accumulator. Other specialized instructions may exist to transfer the top of the stack between other specialized registers, such as the status register. The Z80 is more versatile than most in this respect.

The availability of a stack is required to implement three programming facilities within the computer system: subroutines, interrupts, and temporary data storage. The role of the stack during subroutines will be explained in Chapter 3 (Basic Programming Techniques). The role of the stack during interrupts will be explained in Chapter 6 (Input/Output Techniques). Finally, the role of the stack in saving data at high speed will be explained during specific application programs.

We will simply assume at this point that the stack is a required facility in every computer system. A stack may be implemented in two ways:

1. A fixed number of registers may be provided within the microprocessor itself. This is a "hardware stack." It has the advantage of high speed. However, it has the disadvantage of a limited number of registers.

2. Most general-purpose microprocessors choose another approach, the software stack, in order not to restrict the stack to a very small number of registers. This is the approach chosen in the Z80. In the software approach, a dedicated register within the microprocessor, here register SP, stores the stack pointer, i.e., the address of the top element of the stack (or, sometimes, the address of the top element of the stack plus one). The stack is then implemented as an area of memory. The stack pointer will therefore require 16 bits to point anywhere in the memory.

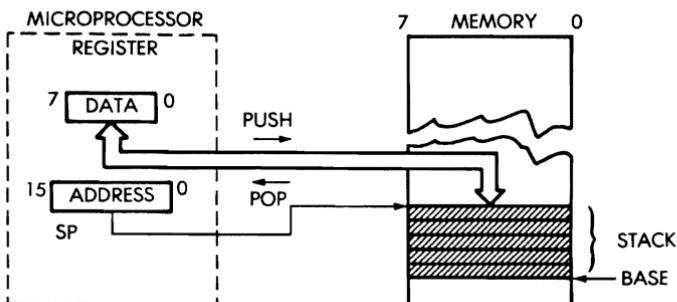


Fig. 2.5: The Two-Stack Manipulation Instructions

The Instruction Execution Cycle

Let us refer now to Figure 2.6. The microprocessor unit appears on the left, and the memory appears on the right. The memory chip may be a ROM or a RAM, or any other chip which happens to contain memory. The memory is used to store instructions and data. Here, we will fetch one instruction from the memory to illustrate the role of the program counter. We assume that the program counter has valid contents. It now holds a 16-bit address which is the address of the next instruction to fetch in the memory. Every processor proceeds in three cycles:

- 1—fetch the next instruction
- 2—decode the instruction
- 3—execute the instruction

Fetch

Let us now follow the sequence. In the first cycle, the contents of the program counter are deposited on the address bus and gated to the memory (on the address bus). Simultaneously, a read signal may be issued on the control bus of the system, if required. The memory will receive the address. This address is used to specify one location within the memory. Upon receiving the read signal, the memory will decode the address it has received, through internal decoders, and will select the location specified by the address. A few hundred nanoseconds later, the memory will deposit the eight-bit data corresponding to the specified address on its data bus. This eight-bit word is the instruction that we want to fetch. In our illustration, this instruction will be deposited on top of the MPU box.

Let us briefly summarize the sequencing: the contents of the program counter are output on the address bus. A read signal is generated. *The memory cycles*, and perhaps 300 nanoseconds later, the instruction at the specified address is deposited on the data bus (assuming a single byte instruction). The microprocessor then reads the data bus and deposits its contents into a specialized internal register, the IR register. The IR is the *instruction register*: it is eight-bits wide and is used to contain the instruction just fetched from the memory. The fetch cycle is now completed. The 8 bits of the instruction are now physically in the special internal register of the MPU, the IR register. The IR appears on the left of Figure 2.7. It is not accessible to the programmer.

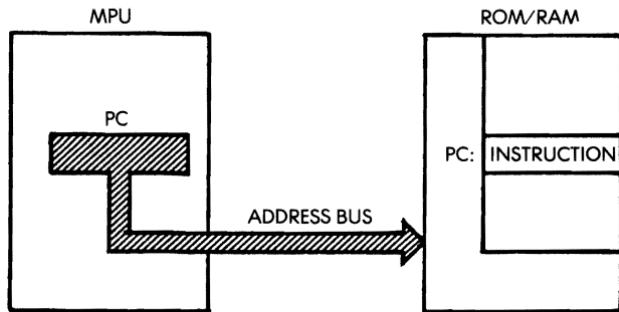


Fig. 2.6: Fetching an Instruction from the Memory

Decoding and Execution

Once the instruction is contained in IR, the control unit of the microprocessor will decode the contents and will be able to generate the correct sequence of internal and external signals for the execution of the specified instruction. There is, therefore, a short decoding delay followed by an execution phase, the length of which depends on the nature of the instruction specified. Some instructions will execute entirely within the MPU. Other instructions will fetch or deposit data from or into the memory. This is why the various instructions of the MPU require various lengths of time to execute. This duration is expressed as a number of (clock) cycles. Refer to Chapter 4 for the number of

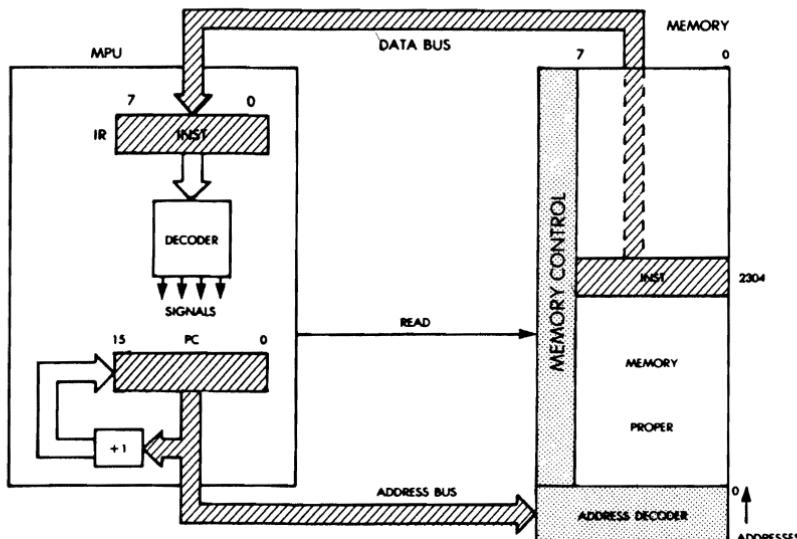


Fig. 2.7: Automatic Sequencing

cycles required by each instruction. Since various clock rates may be used, speed of execution is normally expressed in number of cycles rather than in number of nanoseconds.

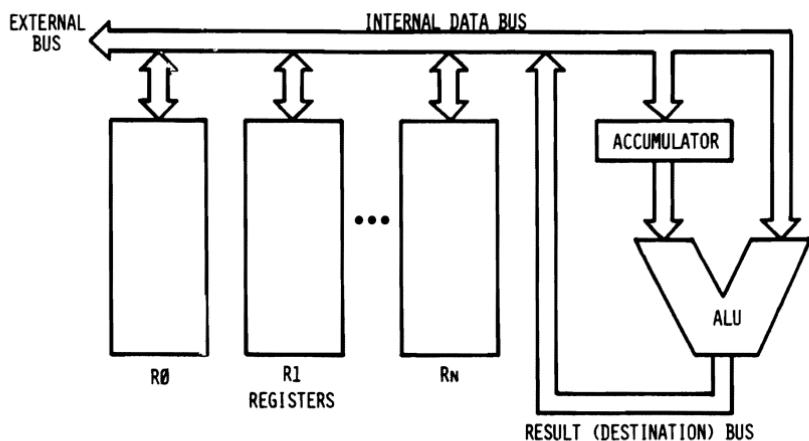


Fig. 2.8: Single-Bus Architecture

Fetching the Next Instruction

We have described how, using the program counter, an instruction can be fetched from the memory. During the execution of a program, instructions are fetched *in sequence* from the memory. An automatic mechanism must therefore be provided to fetch instructions in sequence. This task is performed by a simple incrementer attached to the program counter. This is illustrated in Figure 2.7. Every time that the contents of the program counter (at the bottom of the illustration) are placed on the address bus, its contents will be incremented and written back into the program counter. As an example, if the program counter contained the value "0", the value "0" would be output on the address bus. Then the contents of the program counter would be incremented and the value "1" would be written back into the program counter. In this way, the next time that the program counter is used, it is the instruction at address 1 that will be fetched. We have just implemented an *automatic mechanism for sequencing instructions*.

It must be stressed that the above descriptions are simplified. In reality, some instructions may be two- or even three-bytes long, so that successive bytes will be fetched in this manner from the memory. However, the mechanism is identical. The program counter is used to fetch

PROGRAMMING THE Z80

successive bytes of an instruction as well as to fetch successive instructions themselves. The program counter, together with its incrementer, provides an automatic mechanism for pointing to successive memory locations.

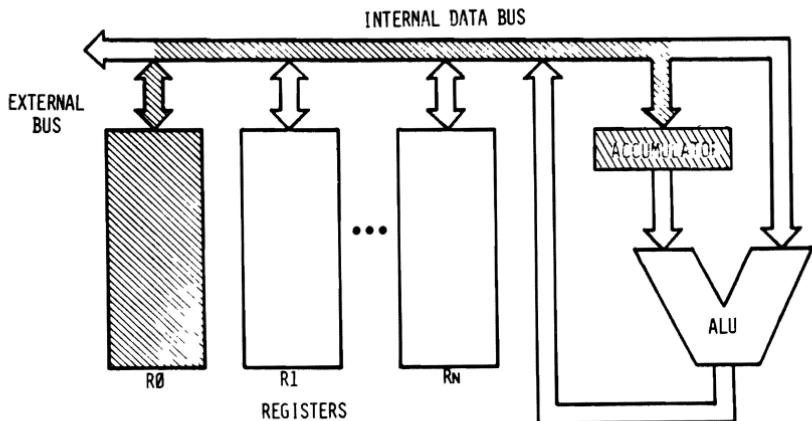


Fig. 2.9: Execution of an Addition—R0 into ACC

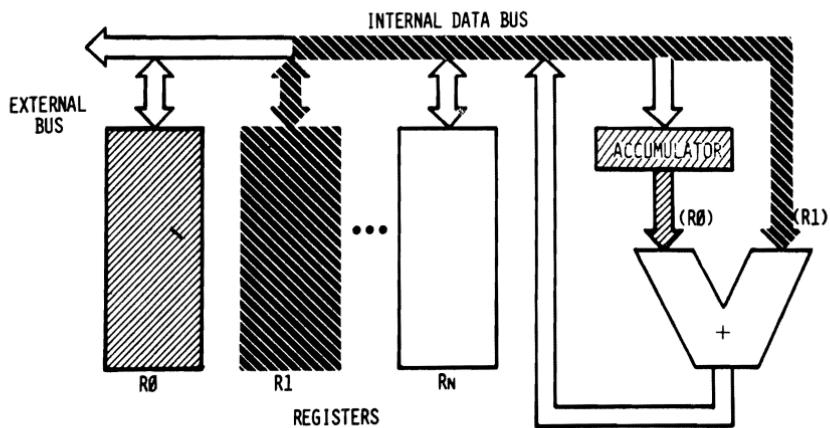


Fig. 2.10: Addition—Second Register R1 into ALU

We will now execute an instruction within the MPU (see Figure 2.8). A typical instruction will be, for example: $R0 = R0 + R1$. This means: "ADD the contents of R0 and R1, and store the results in R0." To perform this operation, the contents of R0 will be read from register R0, carried via the single bus to the left input of the ALU, and stored in the buffer register there. R1 will then be selected and its contents will be read onto the bus, then transferred to the right input of the ALU. This sequence is illustrated in Figures 2.9 and 2.10. At this point, the right input of the ALU is conditioned by R1, and the left input of the ALU is conditioned by the buffer register, containing the previous value of R0. The operation can be performed. The addition is performed by the ALU, and the results appear on the ALU output, in the lower right-hand corner of Fig. 2.11. The results will be deposited on the single bus, and will be propagated back to R0. This means, in practice, that the input latch of R0 will be enabled, so that data can be written into it. Execution of the instruction is now complete. The results of the addition are in R0. It should be noted that the contents of R1 have not been modified by this operation. This is a general principle: the contents of a register, or of any read/write memory, are not modified by a read operation.

The buffer register on the left input of the ALU was necessary in order to *memorize* the contents of R0, so that the single bus could be used again for another transfer. However, a problem remains.

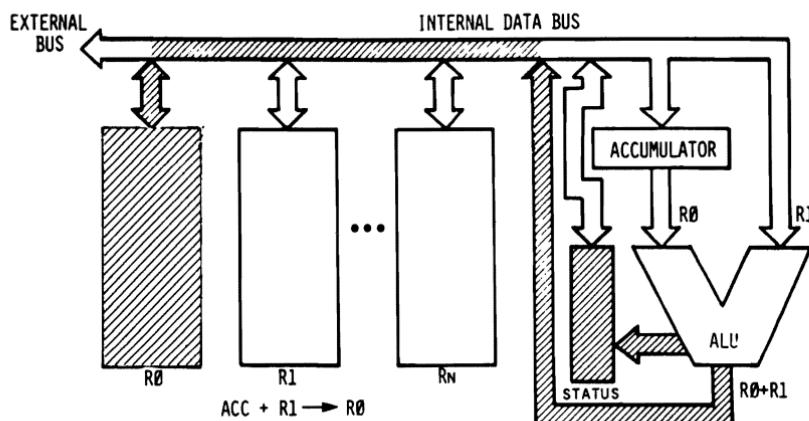


Fig. 2.11: Result Is Generated and Goes into R0

The Critical Race Problem

The simple organization shown in Figure 2.8 will not function correctly.

Question: *What is the timing problem?*

Answer: The problem is that the result which will be propagated out of the ALU will be deposited back on the single bus. It will not propagate just in the direction of R0, but along all of the bus. In particular, it will recondition the right input of the ALU, changing the result coming out of it a few nanoseconds later. This is a *critical race*. The output of the ALU must be isolated from its input (see Figure 2.12).

Several solutions are possible which will isolate the input of the ALU from the output. A buffer register must be used. The buffer register could be placed on the output of the ALU, or on its input. It is usually placed on the input of the ALU. Here it would be placed on its right input. The buffering of the system is now sufficient for a correct operation. It will be shown later in this chapter that if the left register which appears in this illustration is to be used as an accumulator (permitting the use of one-byte long instructions), then the accumulator will require a buffer too, as shown in Figure 2.13.

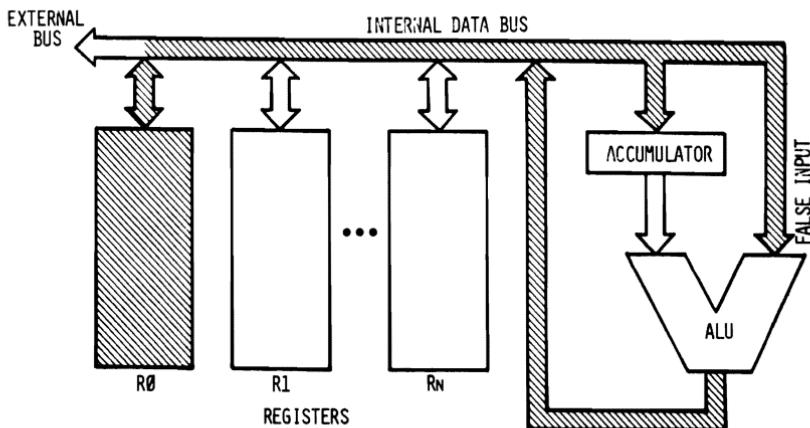


Fig. 2.12: The Critical Race Problem

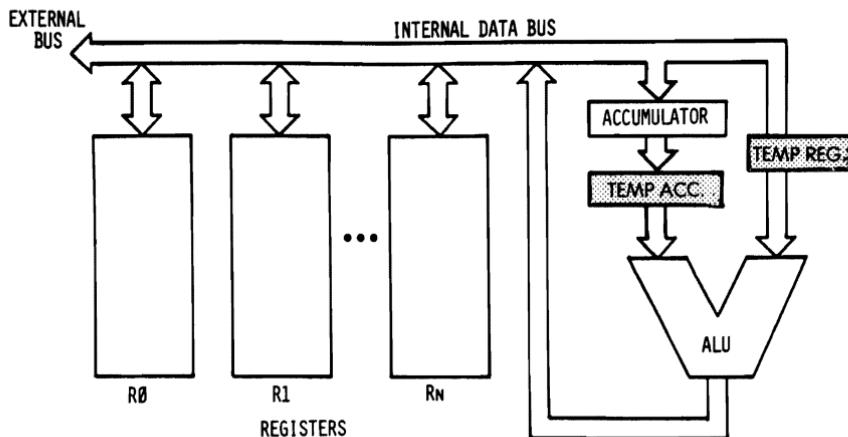


Fig. 2.13: Two Buffers Are Required (Temp Registers)

INTERNAL ORGANIZATION OF THE Z80

The terms necessary in order to understand the internal elements of the microprocessor have been defined. We will now examine in more detail the Z80 itself, and describe its capabilities. The internal organization of the Z80 is shown in Figure 2.14. This diagram presents a logical description of the device. Additional interconnections may exist but are not shown. Let us examine the diagram from right to left.

On the right part of the illustration, the *arithmetic-logical unit* (the ALU) may be recognized by its characteristic "V" shape. The accumulator register, which has been described in the previous section, is identified as A on the right input path of the ALU. It has been shown in the previous section that the accumulator should be equipped with a *buffer register*. This is the register labeled ACT (temporary accumulator). Here, the left input of the ALU is also equipped with a *temporary register*, called TMP. The operation of the ALU will become clear in the next section, where we will describe the execution of actual instructions.

The *flags register* is called "F" in the Z80, and is shown on the right of the accumulator register. The contents of the flags register are essentially conditioned by the ALU, but it will be shown that some of its bits may also be conditioned by other modules or events.

The accumulator and the flags registers are shown as double registers labelled respectively A, A' and F, F'. This is because the Z80 is

PROGRAMMING THE Z80

equipped internally with two sets of registers: A + F, and A' + F'. However, only *one* set of these registers may be used at any one time. A special instruction is provided to exchange the contents of A and F with A' and F'. In order to simplify the explanations, only A and F will be shown on most of the diagrams which follow. The reader should remember that he has the option of switching to the alternate register set A' and F' if desired.

The role of each flag in the flags register will be described in Chapter 3 (Basic Programming Techniques).

A large block of registers is shown at the center of the illustration. On top of the block of registers, two identical groups can be recognized. Each one includes six registers labeled B, C, D, E, H, L. These are the *general-purpose eight-bit registers* of the Z80. There are two peculiarities of the Z80 with respect to the standard microprocessor which has been described at the beginning of this chapter.

First, the Z80 is equipped with *two* banks of registers, i.e., two identical groups of 6 registers. Only six registers may be used at any one time. However, special instructions are provided to switch between the two banks of registers. One bank, therefore, behaves as an internal memory, while the other one behaves as a working set of internal registers. The possible uses of this special facility will be described in the next chapter.

Conceptually, it will be assumed, for the time being, that there are only six working registers, B, C, D, E, H, and L, and the second register bank will temporarily be ignored, in order to avoid confusion.

The MUX symbol which appears above the memory bank is an abbreviation for *multiplexer*. The data coming from the internal data bus will be gated through the multiplexer to the selected register. However, only one of these registers can be connected to the internal data bus at any one time.

A second characteristic of these six registers, in addition to being general-purpose eight-bit registers, is that they are equipped with a connection to the *address bus*. This is why they have been grouped in *pairs*. For example, the contents of B and C can be gated simultaneously onto the 16-bit address bus which appears at the bottom of the illustration. As a result, this group of 6 registers may be used to store either eight-bit data or else 16-bit *pointers* for memory addressing.

The third group of registers, which appears below the two previous ones in the middle of Figure 2.14, contains four "pure" address registers. As in any microprocessor, we find the program counter (PC) and the stack pointer (SP). Recall that the program counter contains

the address of the next instruction to be executed.

The stack pointer points to the top of the stack in the memory. In the case of the Z80, the stack pointer points to the *last actual entry* in the stack. (In other microprocessors, the stack pointer points just above the last entry.) Also, the stack grows “*downwards*,” i.e. towards the lower addresses.

This means that the stack pointer must be *decremented* any time a new word is *pushed* on the stack. Conversely, whenever a word is *removed* (popped) from the stack, the stack pointer must be *incremented* by one. In the case of the Z80, the “push” and “pop” always involve *two words* at the same time, so that the contents of the stack pointer will be decremented or incremented by two.

Looking at the remaining two registers of this group of four registers, we find a new type of register which has not been described yet: two *index-registers*, labeled IX (Index Register X) and IY (Index Register Y). These two registers are equipped with a special adder shown as a miniature V-shaped ALU on the right of these registers in Figure 2.14. A byte brought along the internal data bus may be added to the contents of IX or IY. This byte is called the *displacement*, when using an indexed instruction. Special instructions are provided which will automatically add this displacement to the contents of IX or IY and generate an address. This is called *indexing*. It allows convenient access to any sequential block of data. This important facility will be described in Chapter 5 on addressing techniques.

Finally, a special box labeled “ ± 1 ” appears below and to the left of the block of registers. This is an increment/decrement. The contents of any of the register pairs SP, PC, BC, DE, HL (the “pure address” registers) may be automatically incremented or decremented every time they deposit an address on the internal address bus. This is an essential facility for implementing automated *program loops* which will be described in the next section. Using this feature it will be possible to access successive memory locations conveniently.

Let us move now to the left of the illustration. One register pair is shown, isolated on the left: I and R. The I register is called the *interrupt-page address register*. Its role will be described in the section on interrupts of Chapter 6 (Input/Output Techniques). It is used only in a special mode where an indirect call to a memory location is generated in response to an interrupt. The I register is used to store the high-order part of the indirect address. The lower part of the address is supplied by the device which generated the interrupt.

PROGRAMMING THE Z80

The R register is the *memory-refresh register*. It is provided to refresh dynamic memories automatically. Such a register has traditionally been located outside the microprocessor, since it is associated with the dynamic memory. It is a convenient feature which minimizes the amount of external hardware for some types of dynamic memories. It will not be used here for any programming purposes, as it is essentially a hardware feature (see reference C207 "Microprocessor Interfacing Techniques" for a detailed description of memory refresh techniques). However, it is possible to use it as a software clock, for example.

Let us move now to the far left of the illustration. There the control section of the microprocessor is located. From top to bottom, we find first the *instruction register* IR, which will contain the instruction to be executed. The IR register is totally distinct from the "I, R" register pair described above. The instruction is received from the memory via the data bus, is transmitted along the internal data bus and is finally deposited into the instruction register. Below the instruction register appears the *decoder* which will send signals to the controller-sequencer and cause the execution of the instruction within the microprocessor and outside it. The *control section* generates and manages the control bus which appears at the bottom part of the illustration.

The three buses managed or generated by the system, i.e., the data bus, the address bus, and the control bus, propagate outside the microprocessor through its pins. The external connections are shown on the right-most part of the illustration. The buses are isolated from the outside through buffers shown in Figure 2.14.

All the logical elements of the Z80 have now been described. It is not essential to understand the detailed operation of the Z80 in order to start writing programs. However, for the programmer who wishes to write efficient codes, the speed of a program and its size will depend upon the correct choice of registers as well as the correct choice of techniques. To make a correct choice, it is necessary to understand how instructions are executed within the microprocessor. We will therefore examine here the execution of typical instructions inside the Z80 to demonstrate the role and use of the internal registers and buses.

Z80 HARDWARE ORGANIZATION

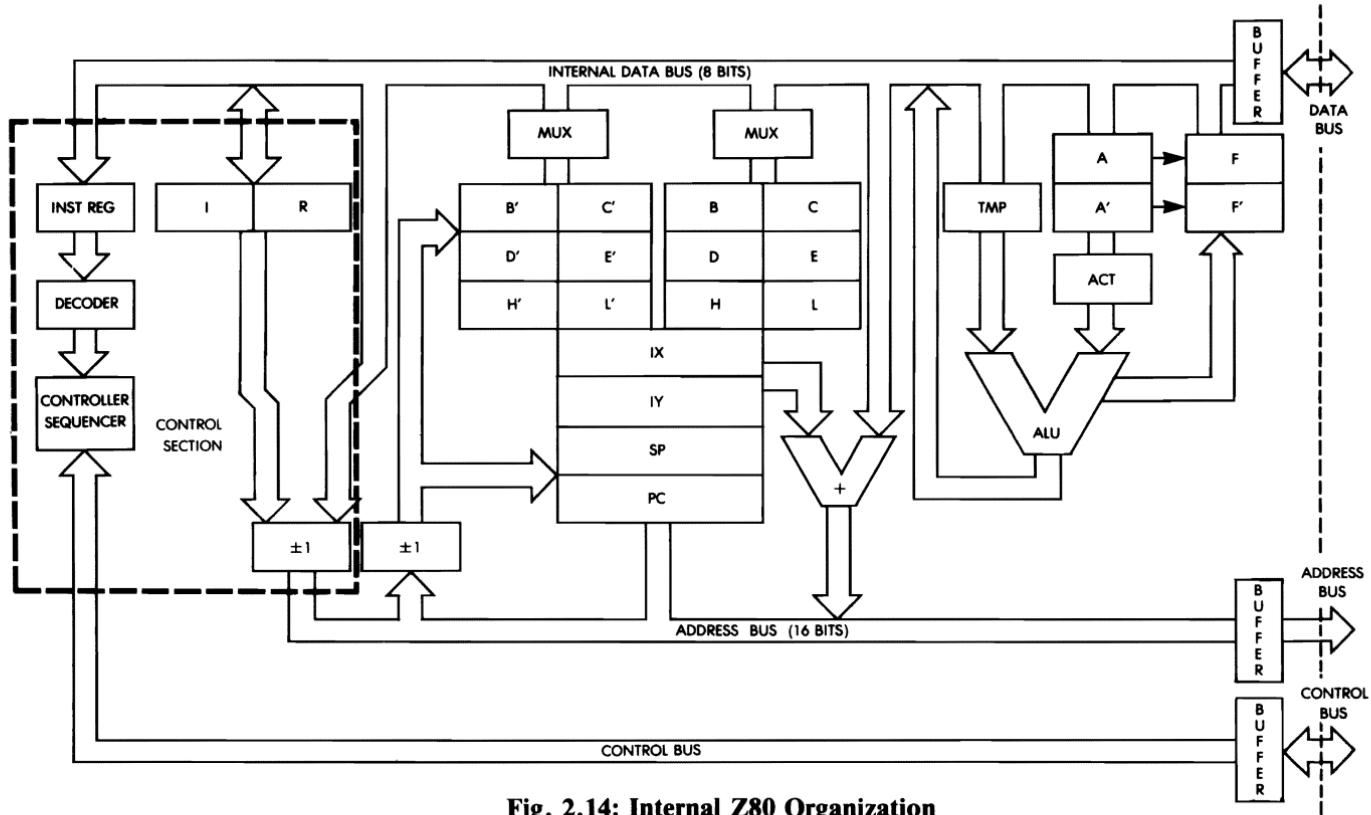


Fig. 2.14: Internal Z80 Organization

INSTRUCTION FORMATS

The Z80 instructions are listed in Chapter 4. Z80 instructions may be formated in one, two, three or four bytes. An instruction specifies the operation to be performed by the microprocessor. From a simplified standpoint, every instruction may be represented as an opcode followed by an optional literal or address field, comprising one or two words. The opcode field specifies the operation to be carried out. In strict computer terminology, the opcode represents only those bits which specify the operation to be performed, exclusive of the register pointers that might be necessary. In the microprocessor world, it is convenient to call opcode the operation code itself, as well as any register pointers which it might incorporate. This "generalized opcode" must reside in an eight-bit word for efficiency (this is the limiting factor on the number of instructions available in a microprocessor).

The 8080 uses instructions which may be one, two, three, bytes long (see Figure 2.15). However, the Z80 is equipped with additional indexed instructions, which require one more byte. In the case of the Z80, opcodes are, in general, one byte long, except for special instructions which require a two-byte opcode.

Some instructions require that one byte of data follow the opcode. In such a case, the instruction will be a two-byte instruction, the second byte of which is data (except for indexing, which adds an extra byte).

In other cases, the instruction might require the specification of an address. An address requires 16 bits and, therefore, two bytes. In that case, the instruction will be a three-byte or a four-byte instruction.

For each byte of the instruction, the control unit will have to perform a memory fetch, which will require four clock cycles. The shorter the instruction, the faster the execution.

A One-Word Instruction

One-word instructions are, in principle, fastest and are favored by the programmer. A typical such instruction for the Z80 is:

LD r, r'

This instruction means: "Transfer the contents of register r' into r." This is a typical "register-to-register" operation. Every microprocessor must be equipped with such instructions, which allow the programmer to transfer information from any of the machine's registers into another one. Instructions referencing special registers of the machine,

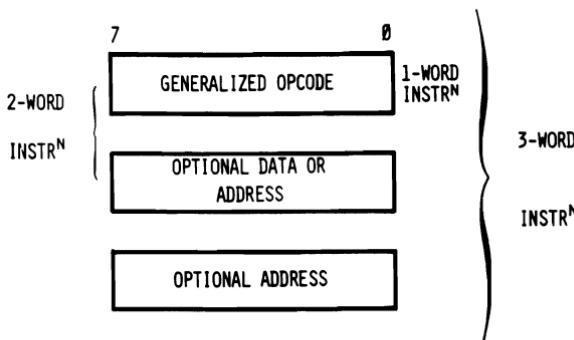


Fig. 2.15 Typical Instruction Formats

such as the accumulator or other special-purpose registers, may have a special opcode.

After execution of the above instruction, the contents of r will be equal to the contents of r' . The contents of r' will *not* have been modified by the read operation.

Every instruction must be represented internally in a binary format. The above representation "LD r,r'" is symbolic or *mnemonic*. It is called the *assembly-language* representation of an instruction. It is simply meant as a convenient symbolic representation of the actual binary encoding for that instruction. The binary code which will represent this instruction inside the memory is: 0 1 D D D S S S (bits 0 to 7).

This representation is still partially symbolic. Each of the letters S and D stands for a binary bit. The three D's, "D D D", represent the three bits pointing to the *destination* register. Three bits allow selection of one out of eight possible registers. The codes for these registers appear in Figure 2.16. For example, the code for register B is "0 0 0", the code for register C is "0 0 1", and so on.

Similarly, "S S S" represents the three bits pointing to the *source* register. The convention here is that register r' is the source, and that register r is the destination. The placement of the bits in the binary representation of an instruction is not meant for the convenience of the programmer, but for the convenience of the control section of the microprocessor, which must decode and execute the instruction. The *assembly-language* representation, however, is meant for the convenience of the programmer. It could be argued that LD r,r' should really mean: "Transfer contents of r into r' ." However, the convention has

PROGRAMMING THE Z80

been chosen in order to maintain compatibility with the binary representation in this case. It is naturally arbitrary.

Exercise 2.1: Write below the binary code which will transfer the contents of register C into register B. Consult Fig. 2.16 for the codes corresponding to C and B.

Another simple example of a one-word instruction is:

ADD A, r

This instruction will result in adding the contents of a specified register (r) to the accumulator (A). Symbolically, this operation may be represented by: $A = A + r$. It can be verified in Chapter 4 that the binary representation of this instruction is:

1 0 0 0 S S S

where S S S specifies the register to be added to the accumulator. Again, the register codes appear in Figure 2.16.

Exercise 2.2: What is the binary code of the instruction which will add the contents of register D to the accumulator?

CODE	REGISTER
0 0 0	B
0 0 1	C
0 1 0	D
0 1 1	E
1 0 0	H
1 0 1	L
1 1 0	- (MEMORY)
1 1 1	A

Fig. 2.16: The Register Codes

A Two-Word Instruction

ADD A, n

This simple two-word instruction will add the contents of the second byte of the instruction to the accumulator. The contents of the second

word of the instruction are said to be a “literal.” They are data and are treated as eight bits without any particular significance. They could happen to be a character or numerical data. This is irrelevant to the operation. The code for this instruction is:

1 1 0 0 0 1 1 0 followed by the 8-bit byte “n”

This is an *immediate* operation. “Immediate,” in most programming languages, means that the next word, or words, within the instruction contains a piece of data which should not be *interpreted* (the way an opcode is). It means that the next one or two words are to be treated as a *literal*.

The control unit is programmed to “know” how many words each instruction has. It will, therefore, always fetch and execute the right number of words for each instruction. However, the longer the possible number of words for the instruction, the more complex it is for the control unit to decode.

A Three-Word Instruction

LD A, (nn)

The instruction requires three words. It means: “Load the accumulator from the memory address specified in the next two bytes of the instruction.” Since addresses are 16-bits long, they require two words. In binary, this instruction is represented by:

0 0 1 1 1 0 1 0:
Low address:
High address:

8 bits for the opcode
8 bits for the lower part of the address
8 bits for the upper part of the address

EXECUTION OF INSTRUCTIONS WITHIN THE Z80

We have seen that all instructions are executed in three phases: FETCH, DECODE, EXECUTE. We now need to introduce some definitions. Each of these phases will require several clock cycles. The Z80 executes each phase in one or more logical cycles, called a “machine cycle.” The shortest machine cycle lasts three clock cycles.

Accessing the memory requires three cycles for any operands, four clock cycles for the initial fetch. Since each instruction must be fetched first from the memory, the fastest instruction will require four clock cycles. Most instructions will require more.

Each machine cycle is labeled as M1, M2, etc., and will require three or more clock cycles, or “states,” labeled T1, T2, etc.

The FETCH Phase

The FETCH phase of an instruction is implemented during the first three states of machine cycle M1; they are called T1, T2, and T3. These three states are common to all instructions of the microprocessor, as all instructions must be fetched prior to execution. The FETCH mechanism is the following:

T1 : PC OUT

The first step is to present the address of the next instruction to the memory. This address is contained in the program counter (PC). As the first step of any instruction fetch, the contents of the PC are placed on the address bus (see Figure 2.17). At this point, an address is presented to the memory, and the memory address decoders will decode this address in order to select the appropriate location within the memory. Several hundred ns (a nanosecond is 10^{-9} second) will elapse before the contents of the selected memory location become available on the out-

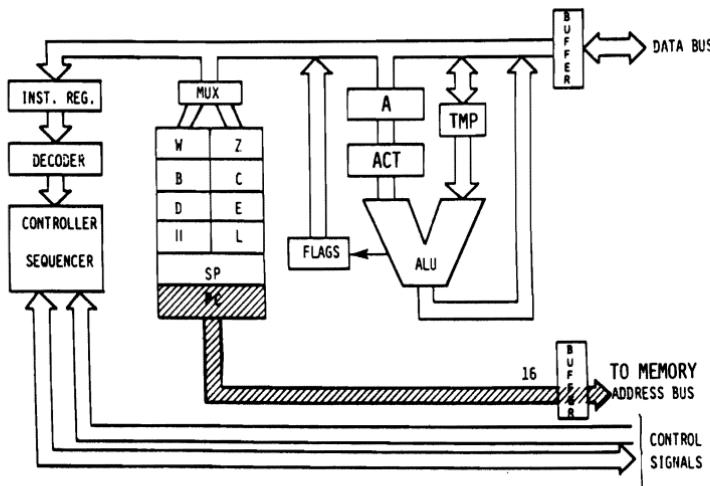


Fig. 2.17: Instruction Fetch—(PC) Is Sent to the Memory

put pins of the memory, which are connected to the data bus. It is standard computer design to use the memory read time to perform an operation within the microprocessor. This operation is the incrementation of the program counter:

$$T2 : PC = PC + 1$$

While the memory is reading, the contents of the PC are incremented by 1 (see Figure 2.18). At the end of state T2, the contents of the memory are available and can be transferred within the microprocessor:

$$T3 : \text{INST} \text{ into IR}$$

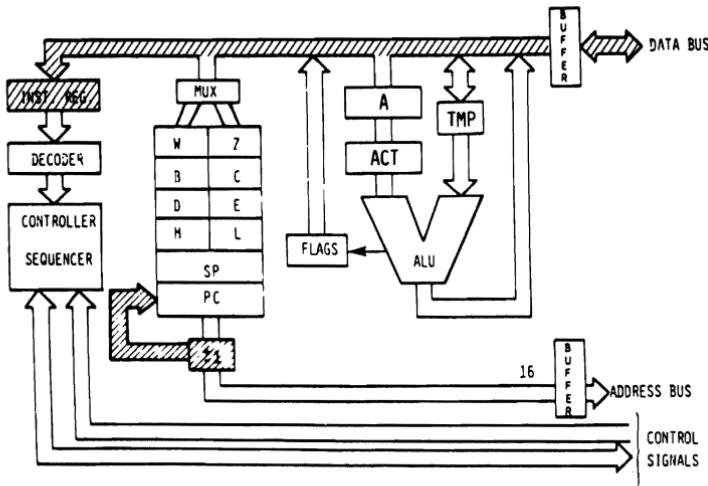


Fig 2.18: PC Is Incremented

The DECODE and EXECUTE Phases

During state T3, the instruction which has been read out of the memory is deposited on the data bus and transferred into the instruction register of the Z80, from which point it is decoded.

PROGRAMMING THE Z80

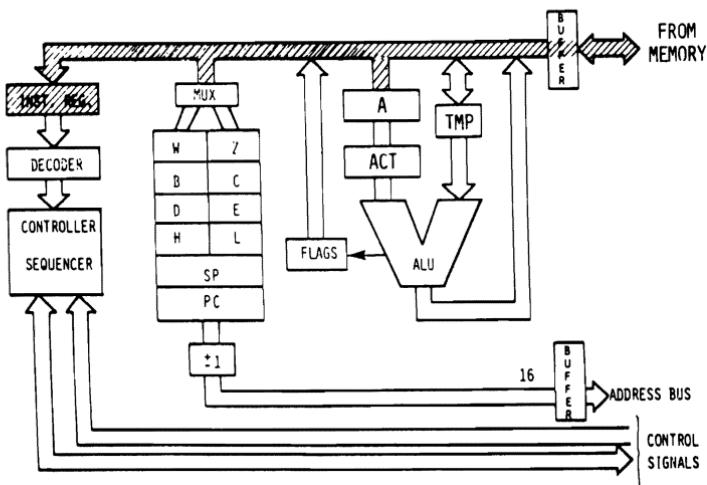


Fig. 2.19: The Instruction Arrives from the Memory into IR

It should be noted that state T4 of M1 will always be required. Once the instruction has been deposited into IR during T3, it is necessary to *decode* and *execute* it. This will require at least one machine state, T4.

A few instructions require an extra state of M1 (state T5). It will be skipped by the processor for most instructions. Whenever the execution of an instruction requires more than M1, i.e., M1, M2 or more cycles, the transition will be directly from state T4 of M1 into state T1 of M2. Let us examine an example. The detailed internal sequencing for each example is shown in the tables of Figure 2.27. As these tables have not been released for the Z80, the 8080 tables are used instead. They provide an in-depth understanding of instruction execution.

LD D, C

This corresponds to MOV r1, r2 for the 8080. Refer to line 1 of Fig. 2.27.

By coincidence, the destination register in this example happens to be named “D”. The transfer is illustrated in Figure 2.20.

This instruction has been described in the previous section. It transfers the contents of register C, denoted by “C”, into register D.

The first three states of cycle M1 are used to fetch the instruction from the memory. At the end of T3, the instruction is in IR, the Instruction Register, from which point it can be decoded (see Figure 2.19).

During T4: (S S S) ▶ TMP.

The contents of C are deposited into TMP (See Figure 2.21).

During T5: (TMP) ▶ DDD.

The contents of TMP are deposited into D. This is shown in Figure 2.22.

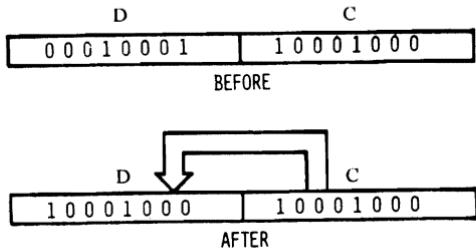


Fig. 2.20: Transferring C into D

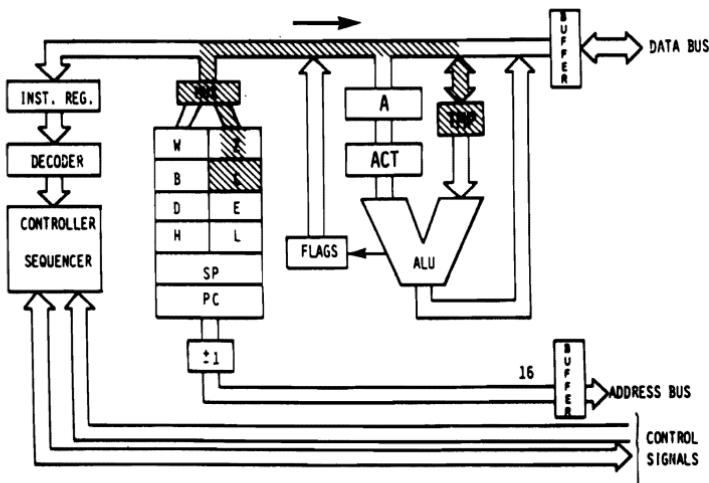


Fig. 2.21: The Contents of C Are Deposited into TMP

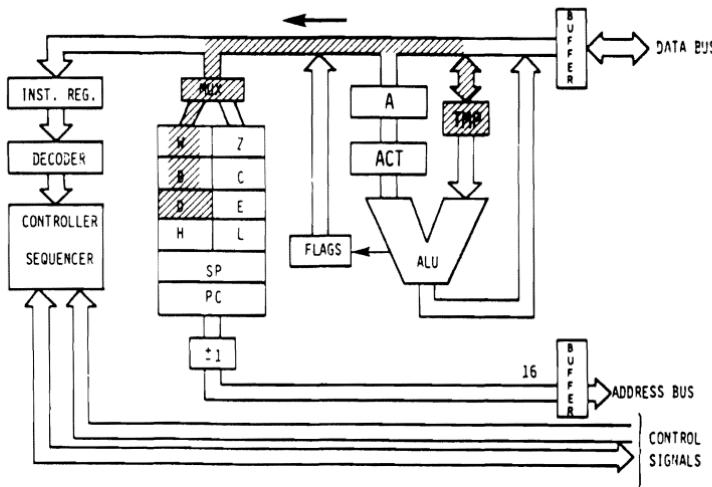


Fig. 2.22: The Contents of TMP are Deposited into D

Execution of the instruction is now complete. The contents of register C have been transferred into the specified destination register D. This terminates execution of the instruction. The other machine cycles M2, M3, M4, and M5 will not be necessary and execution stops with M1.

It is possible to compute the duration of this instruction easily. The duration of every state for the standard Z80 is the duration of the clock: 500 ns. The duration of this instruction is the duration of five states, or $5 \times 500 = 2500$ ns = 2.5 us. With a 400 ns clock, $5 \times 400 = 2000$ ns = 2.0 us.

Question: Why does this instruction require two states, T4 and T5, in order to transfer the contents C into D, rather than just one? It transfers the contents of C into TMP, and then the contents of TMP into D. Wouldn't it be simpler to transfer the contents of C into D directly within a single state?

Answer: This is not possible because of the implementation chosen for the internal registers. All the internal registers are, in fact, part of a

single RAM, a read/write memory internal to the microprocessor chip. Only one word may be addressed or selected at a time within an RAM (single-port). For this reason, it is not possible to both read and write into, or from, an RAM at two different locations. Two RAM cycles are required. It becomes necessary first to read the data out of the register RAM, and store it in a temporary register, TMP, then, to write it back into the final destination register, here D. This is a design inadequacy. However, this limitation is common to virtually all monolithic microprocessors. A dual-port RAM would be required to solve the problem. This limitation is not intrinsic to microprocessors and it normally does not exist in the case of bit-slice devices. It is a result of the constant search for logic density on the chip and may be eliminated in the future.

Important Exercise:

At this point, it is highly recommended that the user review by himself the sequencing of this simple instruction before we proceed to more complex ones. For this purpose, go back to Figure 2.14. Assemble a few small-sized "symbols" such as matches, paperclips, etc. Then move the symbols on Figure 2.14 to simulate the flow of data from the registers into the buses. For example, deposit a symbol into PC. T1 will move the symbol contained in PC out on the address bus towards the memory. Continue simulated execution in this fashion until you feel comfortable with the transfers along the buses and between the registers. At this point, you should be ready to proceed.

Progressively more complex instructions will now be studied:

ADD A, r

This instruction means: "Add the contents of register r (specified by a binary code S S S) to the accumulator (A), and deposit the result in the accumulator." This is an *implicit* instruction. It is called implicit as it does not explicitly reference a second register. The instruction explicitly refers only to register r. It implies that the other register involved in the operation is the accumulator. The accumulator, when used in such an implicit instruction, is referenced both as source and destination. Data will be deposited in the accumulator as a result of this addition. The advantage of such an implicit instruction is that its complete opcode is only eight bits in length. It requires only a three-bit register field for the specification of r. This is a fast way to perform an addition operation.

Other implicit instructions exist in the system which will reference

other specialized registers. More complex examples of such implicit instructions are, for example, the PUSH and POP operations, which will transfer information between the top of the stack and the accumulator, and will at the same time update the stack pointer (SP), decrementing it or incrementing it. They implicitly manipulate the SP register.

The execution of the ADD A, r instruction will now be examined in detail. This instruction will require two machine cycles, M1 and M2. As usual, during the first three states of M1, the instruction is fetched from the memory and deposited in the IR register. At the beginning of T4, it is decoded and can be executed. It will be assumed here that register B is added to the accumulator. The code for the instruction will then be: 1 0 0 0 0 0 0 (the code for register B is 0 0 0). The 8080 equivalent is ADD r.

T4: (S S S) ► TMP, (A) ► ACT

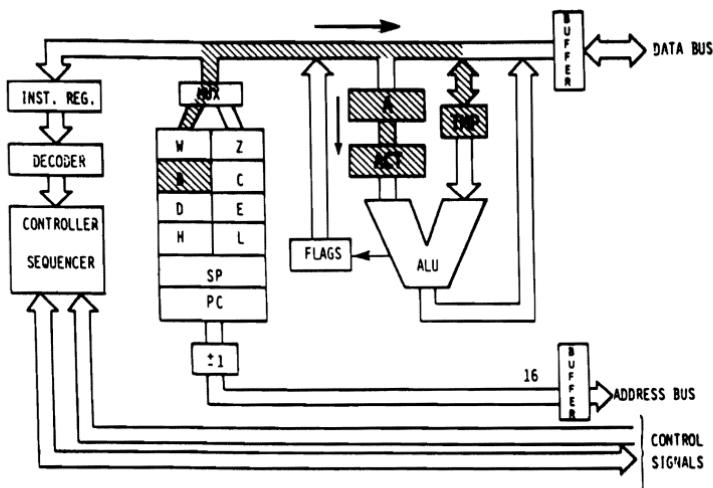


Fig. 2.23: Two Transfers Occur Simultaneously

Two transfers will be executed simultaneously. First, the contents of the specified source register (here B) are transferred into TMP, i.e., to the right input of the ALU (see Fig. 2.23). At the same time, the contents of the accumulator are transferred to the temporary accumulator (ACT). By inspecting Fig. 2.23, you will ascertain that those transfers can occur in parallel. They use different paths within the system. The

transfer from B to TMP uses the internal data bus. The transfer from ACT uses a short internal path independent of this data bus. In order to gain time, both transfers are done simultaneously. At this point, both the left and the right input of the ALU are correctly conditioned. The left input of the ALU is now conditioned by the accumulator contents, and the right input of the ALU is conditioned by the contents of register B. We are ready to perform the addition. We would normally expect to see the addition take place during state T5 of M1. However, this state is simply not used. The addition is not performed! We will enter machine cycle M2. During state T1, nothing happens! It is only in state T2 of M2 that the addition takes place (refer to ADD r in Figure 2.27):

T2 of M2: (ACT) + (TMP) ▶ A

The contents of ACT are added to the contents of TMP, and the result is finally deposited in the accumulator. See Figure 2.24. The operation is now complete.

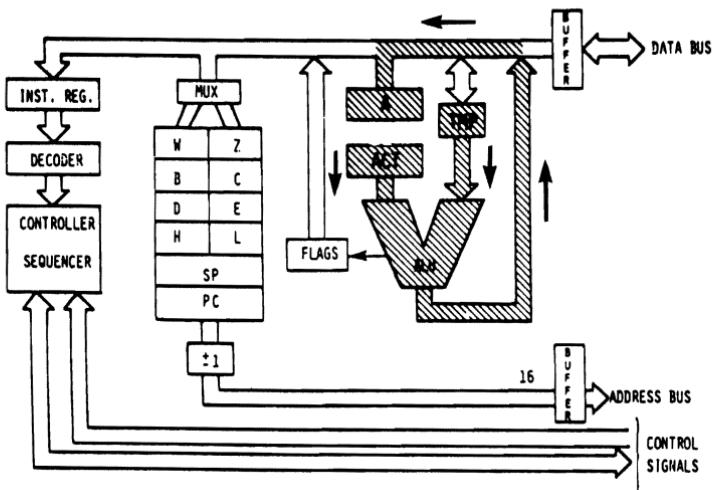


Fig. 2.24: End of ADD r

Question: Why was the completion of the addition deferred until state T2 of machine cycle M2, rather than taking place during state T5 of M1? (This is a difficult question, which requires an understanding of CPU design. However, the technique involved is fundamental to clock-synchronous CPU design. Try to see what happens.)

Answer: This is a standard design “trick” used in most CPU’s. It is called “fetch/execute overlap.” The basic idea is the following: looking back at Figure 2.23 it can be seen that the actual execution of the addition will only require the use of the ALU and of the data bus. In particular, it will not access the register RAM (register block). We (or the control unit) know that the next three states which will be executed after completion of any instruction will be T1, T2, T3 of machine cycle M1 of the next instruction. Looking back at the execution of these three states, it can be seen that their execution will only require access to the program counter (PC) and use of the address bus. Access to the program counter will require access to the register RAM. (This explains why the same trick could not be used in the instruction LD r,r’.) It is therefore possible to use simultaneously the shaded area in Figure 2.17 and the shaded area in Figure 2.24.

The data bus is used during state T1 of M1 to carry status information out. It cannot be used for the addition that we wish to perform. For that reason, it becomes necessary to wait until state T2 before the addition can be effectively carried out. This is what occurred in the chart: the addition is completed during state T2 of M2. The mechanism has now been explained. The advantage of this approach should now be clear. Let us assume that we had implemented a straightforward scheme, and performed the addition during state T5 of machine cycle

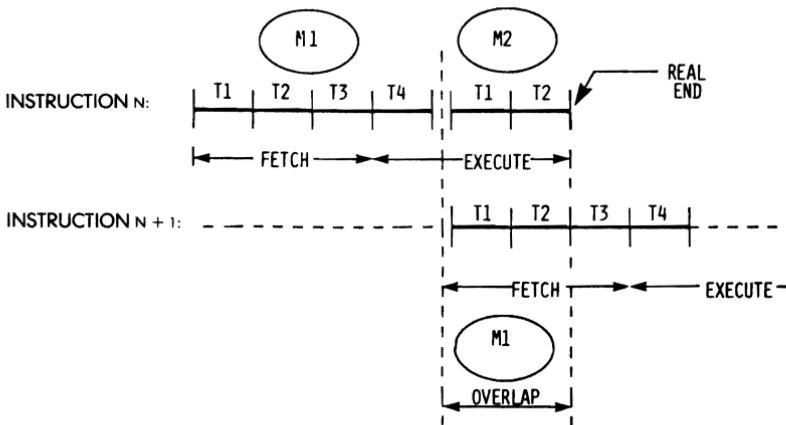


Fig. 2.25: FETCH-EXECUTE Overlap during T1-T2

M1. The duration of the ADD instruction would have been $5 \times 500 = 2500$ ns. With the overlap approach which has been implemented, once state T4 has been executed, the next instruction is initiated. In a manner

that is invisible to this next instruction, the “clever” control unit will use state T2 to carry out the end of the addition. On the chart T2 is shown as part of M2. Conceptually, M2 will be the second machine cycle of the addition. In fact, this M2 will be overlapped, i.e., be identical to machine cycle M1 of the next instruction. For the programmer, the delay introduced by ADD will be only four states, i.e., $4 \times 500 = 2000$ ns, instead of 2500 ns using the “straightforward” approach. The speed improvement is 500 ns, or 20%!

The overlap technique is illustrated on Figure 2.25. It is used whenever possible to increase the apparent execution speed of the microprocessor. Naturally, it is not possible to overlap in all cases. Required buses or facilities must be available without conflict. The control unit “knows” whether an overlap is possible.

NOTES:

1. The first memory cycle (M1) is always an instruction fetch; the first (or only) byte, containing the op code, is fetched during this cycle.
2. If the READY input from memory is not high during T2 of each memory cycle, the processor will enter a wait state (TW) until READY is sampled as high.
3. States T4 and T5 are present, as required, for operations which are completely internal to the CPU. The contents of the internal bus during T4 and T5 are available at the data bus; this is designed for testing purposes only. An “X” denotes that the state is present, but is only used for such internal operations as instruction decoding.
4. Only register pairs rp = B (registers B and C) or rp = D (registers D and E) may be specified.
5. These states are skipped.
6. Memory read sub-cycles; an instruction or data word will be read.
7. Memory write sub-cycle.
8. The READY signal is not required during the second and third sub-cycles (M2 and M3). The HOLD signal is accepted during M2 and M3. The SYNC signal is not generated during M2 and M3. During the execution of DAD, M2 and M3 are required for an internal register-pair add; memory is not referenced.
9. The results of these arithmetic, logical or rotate instructions are not moved into the accumulator (A) until state T2 of the next instruction cycle. That is, A is loaded while the next instruction is being fetched; this overlapping of operations allows for faster processing.
10. If the value of the least significant 4-bits of the accumulator is greater than 9 or, if the auxiliary carry bit is set, 6 is added to the accumulator. If the value of the most significant 4-bits of the accumulator is now greater than 9, or if the carry bit is set, 6 is added to the most significant 4-bits of the accumulator.
11. This represents the first sub-cycle (the instruction fetch) of the next instruction cycle.

12. If the condition was met, the contents of the register pair WZ are output on the address lines (A_{0-15}) instead of the contents of the program counter (PC).

13. If the condition was not met, sub-cycles M4 and M5 are skipped; the processor instead proceeds immediately to the instruction fetch (M1) of the next instruction cycle.

14. If the condition was not met, sub-cycles M2 and M3 are skipped; the processor instead proceeds immediately to the instruction fetch (M1) of the next instruction cycle.

15. Stack read sub-cycle.

16. Stack write sub-cycle.

17. CONDITION	CCC
NZ — not zero ($Z = 0$)	000
Z — zero ($Z = 1$)	001
NC — no carry ($CY = 0$)	010
C — carry ($CY = 1$)	011
PO — parity odd ($P = 0$)	100
PE — parity even ($P = 1$)	101
P — plus ($S = 0$)	110
M — minus ($S = 1$)	111

18. I/O sub-cycle: the I/O port's 8-bit select code is duplicated on address lines 0-7 (A_{0-7}) and 8-15 (A_{8-15}).

19. Output sub-cycle.

20. The processor will remain idle in the halt state until an interrupt, a reset or a hold is accepted. When a hold request is accepted, the CPU enters the hold mode; after the hold mode is terminated, the processor returns to the halt state. After a reset is accepted, the processor begins execution at memory location zero. After an interrupt is accepted, the processor executes the instruction forced onto the data bus (usually a restart instruction).

SSS or DDD	Value	rp	Value
A	111	B	00
B	000	D	01
C	001	H	10
D	010	SP	11
E	011		
H	100		
L	101		

Fig. 2.26: Intel Abbreviations

PROGRAMMING THE Z80

MNEMONIC	OP CODE	M1[1]					M2			
	D ₇ D ₆ D ₅ D ₄	D ₃ D ₂ D ₁ D ₀	T1	T2[2]	T3	T4	T5	T1	T2[2]	T3
MOV r1,r2	0 1 D D	D S S S	PC OUT STATUS	PC = PC + 1	INST-TMP/R	(SSS)→TMP	(TMP)→ODD			
MOV r, M	0 1 D D	D 1 1 0		A		X(3)		HL OUT STATUS[6]		DATA → DDD
MOV M,r	0 1 1 1	0 S S S				(SSS)→TMP		HL OUT STATUS[7]		(TMP)→DATA BUS
SPHL	1 1 1 1	1 0 0 1			(HL)→SP					
MVI r,data	0 0 D D	D 1 1 0				X	PC OUT STATUS[6]		B2 → DDDD	
MVI M,data	0 0 1 1	0 1 1 0				X		A	B2 → TMP	
LXI rp,data	0 0 R P	0 0 0 1				X			PC = PC + 1	B2 → r1
LDA addr	0 0 1 1	1 0 1 0				X			PC = PC + 1	B2 → Z
STA addr	0 0 1 1	0 0 1 0				X			PC = PC + 1	B2 → Z
LHLD addr	0 0 1 0	1 0 1 0				X			PC = PC + 1	B2 → Z
SHLD addr	0 0 1 0	0 0 1 0				X	PC OUT STATUS[6]	PC = PC + 1	B2 → Z	
LDAX rp[4]	U 0 R P	1 0 1 0				X	19 OUT STATUS[6]			DATA → A
STAX rp[4]	0 0 R P	0 0 1 0				X	19 OUT STATUS[7]			(A) → DATA BUS
XCHG	1 1 1 0	1 0 1 1			(HL)→IDE					
ADD r	1 0 0 0	0 S S S			(SSS)→TMP (A)→ACT		[8]			(ACT)+(TMP)→A
ADD M	1 0 0 0	0 1 1 0			(A)→ACT		HL OUT STATUS[6]			DATA → TMP
ADI data	1 1 0 0	0 1 1 0			(A)→ACT		PC OUT STATUS[6]	PC = PC + 1	B2 → TMP	
ADC r	1 0 0 0	1 S S S			(SSS)→TMP (A)→ACT		[8]			(ACT)+(TMP)CY→A
ADC M	1 0 0 0	1 1 1 0			(A)→ACT		HL OUT STATUS[6]			DATA → TMP
ACI data	1 1 0 0	1 1 1 0			(A)→ACT		PC OUT STATUS[6]	PC = PC + 1	B2 → TMP	
SUB r	1 0 0 1	0 S S S			(SSS)→TMP (A)→ACT		[8]			(ACT)-(TMP)→A
SUB M	1 0 0 1	0 1 1 0			(A)→ACT		HL OUT STATUS[6]			DATA → TMP
SUI data	1 1 0 1	0 1 1 0			(A)→ACT		PC OUT STATUS[6]	PC = PC + 1	B2 → TMP	
SSB r	1 0 0 1	1 S S S			(SSS)→TMP (A)→ACT		[8]			(ACT)-(TMP)CY→A
SSB M	1 0 0 1	1 1 1 0			(A)→ACT		HL OUT STATUS[6]			DATA → TMP
SBI data	1 1 0 1	1 1 1 0			(A)→ACT		PC OUT STATUS[6]	PC = PC + 1	B2 → TMP	
INR r	0 0 D D	D 1 0 0			(DDO)→TMP (TMP)+1→ALU	ALU→ODD				
INR M	0 0 1 1	0 1 0 0			X		HL OUT STATUS[6]			DATA → TMP (TMP)+1→ALU
DCR r	0 0 D D	D 1 0 1			(DDO)→TMP (TMP)+1→ALU	ALU→ODD				
DCR M	0 0 1 1	0 1 0 1			X		HL OUT STATUS[6]			DATA → TMP (TMP)-1→ALU
INX rp	0 0 R P	0 0 1 1			(RP)+1→RP					
DCX rp	0 0 R P	1 0 1 1			(RP)-1→RP					
DAD rp[8]	0 0 R P	1 0 0 1			X		[8]→ACT	(LI=TMP (ACT)+(TMP)→ALU		ALU→L CY
DAA	0 0 1 0	0 1 1 1			DAA→A, FLAGS[10]					
ANA r	1 0 1 0	0 S S S			(SSS)→TMP (A)→ACT		[8]			(ACT)+(TMP)→A
ANA M	1 0 1 0	0 1 1 0	PC OUT STATUS	PC = PC + 1	INST-TMP/R	(A)→ACT	HL OUT STATUS[6]			DATA → TMP

Fig. 2.27: Intel Instruction Formats

M3			M4			M8				
T1	T2[8]	T3	T1	T2[8]	T3	T1	T2[8]	T3	T4	T5
HL OUT STATUS[7]	(TMP) → DATA BUS									
PC OUT STATUS[8]	PC = PC + 1	B3 → R								
	PC = PC + 1	B3 → W	W2 OUT STATUS[8]	DATA → A						
	PC = PC + 1	B3 → W	W2 OUT STATUS[7]	(A) → DATA BUS						
	PC = PC + 1	B3 → W	W2 OUT STATUS[8]	DATA → L	W2 OUT STATUS[8]	DATA → H				
PC OUT STATUS[8]	PC = PC + 1	B3 → W	W2 OUT STATUS[7]	(L) → DATA BUS	W2 OUT STATUS[7]	(H) → DATA BUS				
[R]	(ACT) + (TMP) → A									
[R]	(ACT) + (TMP) → A									
[R]	(ACT) + (TMP) + CY → A									
[R]	(ACT) + (TMP) + CY → A									
[R]	(ACT) - (TMP) → A									
[R]	(ACT) - (TMP) → A									
[R]	(ACT) - (TMP) - CY → A									
[R]	(ACT) - (TMP) - CY → A									
HL OUT STATUS[7]	ALU → DATA BUS									
HL OUT STATUS[7]	ALU → DATA BUS									
(H) → ACT	(H) → TMP	(ACT) + (TMP) + CY → ALU	ALU - H, CY							
[R]	(ACT) + (TMP) → A									

© Intel. Reproduced by permission.

Fig. 2.27: Intel Instruction Formats (continued)

PROGRAMMING THE Z80

Mnemonic	Op Code		M1[1]					M2		
	D7 D6 D5 D4	D3 D2 D1 D0	T1	T2[2]	T3	T4	T5	T1	T2[2]	T3
ANI data	1 1 1 0	0 1 1 0	PC OUT STATUS	PC + PC + 1	INST-TMP/R	(A)→ACT		PC OUT STATUS[8]	PC + PC + 1	B2→TMP
XRA r	1 0 1 0	1 S S S				(A)→ACT (SSS)→TMP		[8]	(ACT)+(TMP)→A	
XRA M	1 0 1 0	1 1 1 0				(A)→ACT		HL OUT STATUS[8]	DATA→TMP	
XRI data	1 1 1 0	1 1 1 0				(A)→ACT		PC OUT STATUS[8]	PC + PC + 1	B2→TMP
ORA r	1 0 1 1	0 S S S				(A)→ACT (SSS)→TMP		[8]	(ACT)+(TMP)→A	
ORA M	1 0 1 1	0 1 1 0				(A)→ACT		HL OUT STATUS[8]	DATA→TMP	
ORI data	1 1 1 1	0 1 1 0				(A)→ACT		PC OUT STATUS[8]	PC + PC + 1	B2→TMP
CMP r	1 0 1 1	1 S S S				(A)→ACT (SSS)→TMP		[8]	(ACT)+(TMP), FLAGS	
CMP M	1 0 1 1	1 1 1 0				(A)→ACT		HL OUT STATUS[8]	DATA→TMP	
CPI data	1 1 1 1	1 1 1 0				(A)→ACT		PC OUT STATUS[8]	PC + PC + 1	B2→TMP
RLC	0 0 0 0	0 1 1 1				(A)→ALU ROTATE		[8]	ALU→A, CY	
RRC	0 0 0 0	1 1 1 1				(A)→ALU ROTATE		[8]	ALU→A, CY	
RAL	0 0 0 1	0 1 1 1				(A), CY→ALU ROTATE		[8]	ALU→A, CY	
RAR	0 0 0 1	1 1 1 1				(A), CY→ALU ROTATE		[8]	ALU→A, CY	
CMA	0 0 1 0	1 1 1 1				(A)→A				
CMC	0 0 1 1	1 1 1 1				CY→CY				
STC	0 0 1 1	0 1 1 1				1→CY				
JMP addr	1 1 0 0	0 0 1 1				X		PC OUT STATUS[8]	PC + PC + 1	B2→Z
J cond addr[17]	1 1 C C	C 0 1 0				JUDGE CONDITION		PC OUT STATUS[8]	PC + PC + 1	B2→Z
CALL addr	1 1 0 0	1 1 0 1				SP = SP - 1		PC OUT STATUS[8]	PC + PC + 1	B2→Z
C cond addr[17]	1 1 C C	C 1 0 0				JUDGE CONDITION IF TRUE, SP = SP - 1		PC OUT STATUS[8]	PC + PC + 1	B2→Z
RET	1 1 0 0	1 0 0 1				X		SP OUT STATUS[15]	SP = SP - 1	DATA→Z
R cond addr[17]	1 1 C C	C 0 0 0				INST-TMP/R	JUDGE CONDITION[14]	SP OUT STATUS[15]	SP = SP + 1	DATA→Z
RST n	1 1 N N	N 1 1 1				8-W INST-TMP/R	SP = SP - 1	SP OUT STATUS[16]	SP = SP - 1	(PCH)→DATA BUS
PCHL	1 1 1 0	1 0 0 1				INST-TMP/R	(HL)	PC OUT STATUS[8]	PC + PC + 1	B2→Z
PUSH rp	1 1 R P	0 1 0 1				X	SP = SP - 1	SP OUT STATUS[16]	SP = SP - 1	DATA BUS
PUSH PSW	1 1 1 1	0 1 0 1					SP = SP - 1	SP OUT STATUS[16]	SP = SP - 1	(A)→DATA BUS
POP rp	1 1 R P	0 0 0 1				X		SP OUT STATUS[15]	SP = SP + 1	DATA→P
POP PSW	1 1 1 1	0 0 0 1				X		SP OUT STATUS[15]	SP = SP + 1	DATA→FLAGS
XTHL	1 1 1 0	0 0 1 1				X		SP OUT STATUS[15]	SP = SP + 1	DATA→Z
I/W port	1 1 0 1	1 0 1 1				X		PC OUT STATUS[8]	PC + PC + 1	B2→Z, W
OUT port	1 1 0 1	0 0 1 1				X		PC OUT STATUS[8]	PC + PC + 1	B2→Z, W
EI	1 1 1 1	1 0 1 1				SET INT E/F				
DI	1 1 1 1	0 0 1 1				RESET INT E/F				
HLT	0 1 1 1	0 1 1 0				X		PC OUT STATUS	HALT MODE[2]	
NOP	0 0 0 0	0 0 0 0	PC OUT STATUS	PC + PC + 1	INST-TMP/R	X				

Fig. 2.27¹: Intel Instruction Formats (continued)

M3			M4			M5					
T1	T2[8]	T3	T1	T2[8]	T3	T1	T2[8]	T3	T4	T5	
[B]	(ACT1)+(TMP)→A										
[B]	(ACT1)+(TMP)→A										
[B]	(ACT1)+(TMP)→A										
[B]	(ACT1)+(TMP)→A										
[B]	(ACT1)+(TMP)→A										
[B]	(ACT1)+(TMP)→A										
[B]	(ACT1)+(TMP); FLAGS										
[B]	(ACT1)+(TMP), FLAGS										
PC OUT STATUS[8]	PC = PC + 1	B3 → W									
PC OUT STATUS[8]	PC = PC + 1	B3 → W									
PC OUT STATUS[8]	PC = PC + 1	B3 → W	→ OUT STATUS[16] SP = SP - 1	[PCH] → DATA BUS [PL] → DATA BUS	SP OUT STATUS[16] SP = SP - 1	[PCL] → DATA BUS					
PC OUT STATUS[8]	PC = PC + 1	B3 → W[13]	SP OUT STATUS[16] SP = SP - 1	[PCH] → DATA BUS [PL] → DATA BUS	SP OUT STATUS[16] SP = SP - 1	[PCL] → DATA BUS					
SP OUT STATUS[16]	SP = SP + 1	DATA → W									
SP OUT STATUS[16]	SP = SP + 1	DATA → W									
SP OUT STATUS[16]	ITMP = 0NNNN000 (PCL)	Z → DATA BUS									
SP OUT STATUS[16]											
SP OUT STATUS[16]			(H) → DATA BUS								
SP OUT STATUS[16]			FLAGS → DATA BUS								
SP OUT STATUS[16]			SP = SP + 1	DATA → H							
SP OUT STATUS[16]			SP = SP + 1	DATA → A							
SP OUT STATUS[16]			DATA → W	SP OUT STATUS[16] (H) → DATA BUS	SP OUT STATUS[16]	(L) → DATA BUS					
WZ OUT STATUS[8]			DATA → A								
WZ OUT STATUS[8]			(A) → DATA BUS								

Fig. 2.27¹: Intel Instruction Formats (continued)

Question: *Would it be possible to go further using this scheme, and to also use state T3 of M2 if we have to execute a longer instruction?*

In order to clarify the internal sequencing mechanism, it is suggested that you examine Figure 2.27, which shows the detailed instruction execution for the 8080. The Z80 includes all 8080 instructions, and more. The information presented in Figure 2.27 is not available for the Z80. It is shown here for its educational value in understanding the internal operation of this microprocessor. The equivalence between Z80 and 8080 instructions is shown in Appendices F and G.

A more complex instruction will now be examined:

ADD A, (HL)

The opcode for this instruction is 10000110. This instruction means "add to the accumulator the contents of memory location (HL)." The memory location is specified through a rather strange system. It is the memory location whose address is contained in registers H and L. This instruction assumes that these two special registers (HL) have been loaded with contents prior to executing the instruction. The 16-bit contents of these registers will now specify the address in the memory where data resides. This data will be added to the accumulator, and the result will be left in the accumulator.

This instruction has a history. It has been supplied in order to provide compatibility between the early 8008, and its successor, the 8080. The early 8008 was not equipped with a direct-memory addressing capability! The procedure used to access the contents of the memory was to load the two registers H and L, and then execute an instruction referencing H and L. ADD A, (HL) is just such an instruction. It must be stressed that the 8080 and the Z80 are not limited in the same way as the 8008 in memory-addressing capability. They do have direct-memory addressing. The facility for using the H and L registers becomes an added advantage, not a drawback, as was the case with the 8008.

Let us now follow the execution of this instruction (it is called ADD M for the 8080 and is the 16th instruction on Figure 2.27). States T1, T2, and T3 of M1 will be used, as usual, to *fetch* the instruction. During state T4, the contents of the accumulator are transferred to its buffer register, ACT, and the left input of the ALU is conditioned.

Memory must be accessed in order to provide the second byte of data which will be added to the accumulator. The address of this byte of

data is contained in H and L. The contents of H and L will therefore have to be transferred onto the address bus, where they will be gated to the memory. Let us do it.

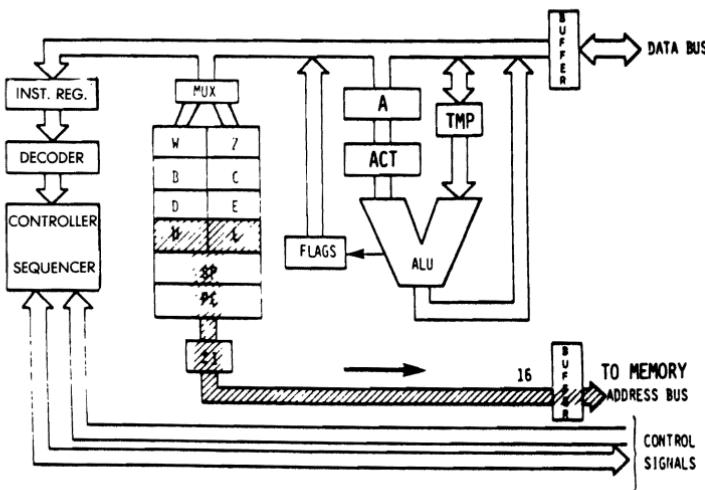


Fig. 2.28: Transfer Contents of HL to Address Bus

During machine cycle M2, we read: HL OUT. H and L are deposited on the address bus, in the same way PC used to be deposited there in previous instructions. As a remark, it has already been indicated that during state T1 *status* is output on the data bus, but no use of this will be made here. From a simplified standpoint, it will require two states: one for the memory to read its data, and one for the data to become available and transferred onto the right input of the ALU, TMP.

Both inputs of the ALU are now conditioned. The situation is analogous to the one we were in with the previous instruction ADDA, r: both inputs of the ALU are conditioned. We simply have to ADD as before. A fetch/execute overlap technique will be used, and, instead of executing the addition within state T4 of M2, final execution is postponed until state T2 of M3. It can be seen in Figure 2.27 that during T2 we indeed have: ACT + TMP → A. The addition is finally performed, the contents of ACT are added to TMP, and the result deposited into the accumulator A.

Question: *What is the apparent execution time (to the programmer) for this instruction? Using a 2.5 Mhz clock, is it 3.6 us? 2.8 us?*

Another more complex instruction will now be examined which is a direct-memory addressing instruction using two invisible W and Z registers:

LD A,(nn)

The opcode is 00111010. The 8080 equivalent is LDA addr. As usual, states T1, T2, T3 of M1 will be used to fetch the instruction from the memory. T4 is used, but no visible result can be described. During state T4, the instruction is in fact decoded. The control unit then finds out that it has to fetch the next two bytes of this instruction in order to obtain the address from which the accumulator will be loaded. The effect of this instruction is to load the accumulator from the memory contents whose address is specified in bytes 2 and 3 of the instruction. Note that state T4 is necessary to *decode* the instruction. It could be considered a waste of time since only part of the state is necessary to do the decoding. It is. However, this is the philosophy of *clock-synchronous logic*. Because *microinstructions* are used internally to perform the decoding and execution, this is the penalty that has to be paid in return for the advantages of microprogramming. The structure of this instruction appears in Figure 2.29.

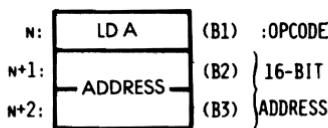
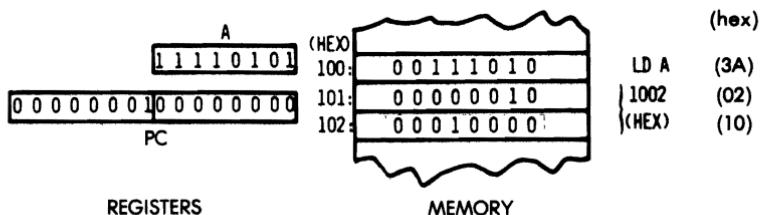
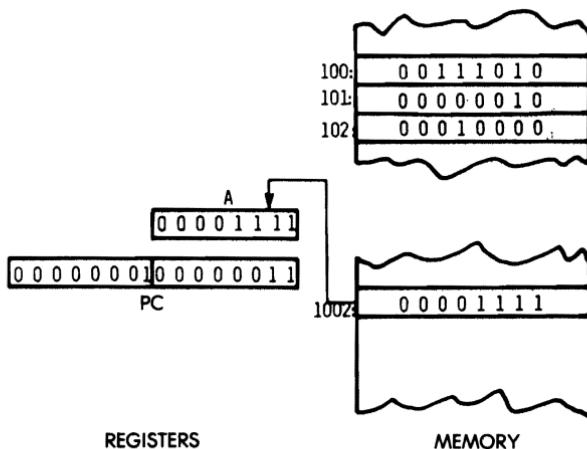


Fig. 2.29: LD A, (ADDRESS) Is a 3-Word Instruction

The next two bytes of instruction will now be fetched. They will specify an address (see Figure 2.30).

**Fig. 2.30: Before Execution of LD A****Fig. 2.31: After Execution of LD A**

The effect of the instruction is shown in Figures 2.30 and 2.31 above.

Two special registers are available to the control unit within the Z80 (but not to the programmer). They are "W" and "Z", and are shown in Figure 2.28.

Second Machine Cycle M2: As usual, the first 2 states, T1 and T2, are used to fetch the contents of memory location PC. During T2, the program counter, PC, is incremented. Sometime by the end of T2, data becomes available from the memory, and appears on the data bus. By the end of T3, the word which has been fetched from memory address PC (B2, second byte of the instruction) is available on the data bus. It must now be stored in a temporary register. It is deposited into Z: B2 \rightarrow Z (see Figure 2.32).

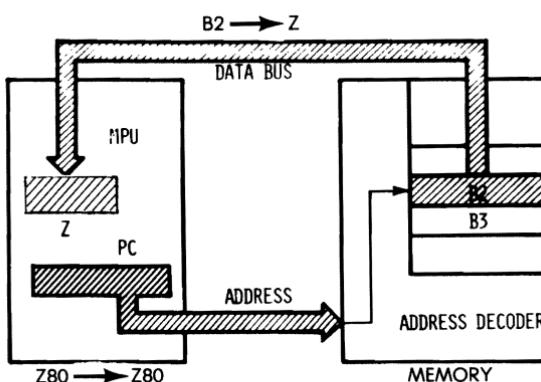


Fig. 2.32: Second Byte of Instruction Goes into Z

Machine Cycle M3: Again, PC is deposited on the address bus, incremented, and finally the third byte, B3, is read from the memory and deposited into register W of the microprocessor. At this point, i.e., by the end of state T3 of M3, registers W and Z inside the microprocessor contain B2 and B3, i.e., the complete 16-bit address which was originally contained in the two words following the instruction in the memory. Execution can now be completed. W and Z contain an address. This address will have to be sent to the memory, in order to extract the data. This is done in the next memory cycle:

Machine Cycle M4: This time, W and Z are output on the address bus. The 16-bit address is sent to the memory, and by the end of state T2, data corresponding to the contents of the specified memory location becomes available. It is finally deposited in A at the end of state T3. This terminates execution of this instruction.

This illustrates the use of an *immediate instruction*. This instruction required three bytes in order to store a two-byte *explicit address*. This instruction also required four memory cycles, as it needed to go to the memory three times in order to extract the three bytes of this three-word instruction, plus one more memory access in order to fetch the data specified by the address. It is a long instruction. However, it is also a basic one for loading the accumulator with specified contents residing at a known memory location. It can be noted that this instruction requires the use of W and Z registers.

Question: Could this instruction have used other registers than W, Z within the system?

Answer: No. If this instruction had used other registers, for example the H and L registers, it would have modified their contents. After execution of this instruction, the contents of H and L would have been lost. It is always assumed in a program that an instruction will not modify any registers other than those it is explicitly using. An instruction loading the accumulator should not destroy the contents of any other register. For this reason, it becomes necessary to supply the extra two registers, W and Z, for the internal use of the control unit.

Question: Would it be possible to use PC instead of W and Z?

Answer: Positively not. This would be suicidal. The reader should analyze this.

One more type of instruction will be studied now: a *branch or jump* instruction, which modifies the sequence in which instructions are executed within the program. So far, we have assumed that instructions were executed sequentially. Instructions exist which allow the programmer to jump out of sequence to another instruction within the program, or in practical terms, to jump to another area of the memory containing the program, or to another address. One such instruction is:

JP nn

This instruction appears on Line 18 of Figure 2.27¹ as "JMP addr." Its execution will be described by following the horizontal line of the Table. This is again a three-word instruction. The first word is the opcode, and contains 11000011. The next two words contain the

16-bit address, to which the jump will be made. Conceptually, the effect of this instruction is to replace the contents of the program counter with the 16 bits following the “JUMP” opcode. In practice, a somewhat different approach will be implemented, for reasons of efficiency.

As before, the first three states of M1 correspond to the instruction-fetch. During state T4 the instruction is decoded and no other event is recorded (X). The next two machine cycles are used to fetch bytes B2 and B3 of the instruction. During M2, B2 is fetched and deposited into internal register Z. The next two steps will be implemented by the processor during the next instruction-fetch, as was the case already with the addition. They will be executed instead of the usual steps for T1 and T2 of the next instruction. Let us look at them.

The next two steps will be: WZ OUT and $(WZ) + 1 \blacktriangleright PC$. In other words, the contents of WZ will be used instead of the contents of PC during the next instruction-fetch. The control unit will have recorded the fact that a jump was being executed and will execute the beginning of the next instruction differently.

The effect of these two extra states is the following:

The address placed on the address bus of the system will be the address contained in W and Z. In other words, the next instruction will be fetched from the address that was contained in W and Z. This is effectively a *jump*. In addition, the contents of WZ will be incremented by 1 and deposited in the program counter, so that the next instruction will be fetched correctly by using PC as usual. The effect is therefore correct.

Question: *Why have we not loaded the contents of PC directly? Why use the intermediate W and Z registers?*

Answer: It is not possible to use PC. If we had loaded the lower part of PC (PCL) with B2, instead of using Z, we would have destroyed PC! It would then have become impossible to fetch B3.

Question: *Would it be possible to use just Z, instead of W and Z?*

Answer: Yes, but it would be slower. We could have loaded Z with B2, then fetched B3, and deposited it into the high order half of PC (PCH). However, it would then have become necessary to transfer Z into PCL, before using the contents of PC. This would slow down the process. For this reason, both W and Z should be used. Further, and in order to save time, W and Z are not transferred into PC. They are directly gated to the address bus in order to fetch the next instruction.

Understanding this point is crucial to the understanding of efficient execution of instructions within the microprocessor.

Question: (*For the alert and informed reader only*). What happens in the case of an interrupt at the end of M3? (If instruction execution is suspended at this point, the program counter points to the instruction following the jump, and the jump address, contained in W and Z, will be lost.)

The answer is left as an interesting exercise for the alert reader.

The detailed descriptions we have presented for the execution of typical instructions should clarify the role of the registers and of the internal buses. A second reading of the preceding section may help in gaining a detailed understanding of the internal operation of the Z80.

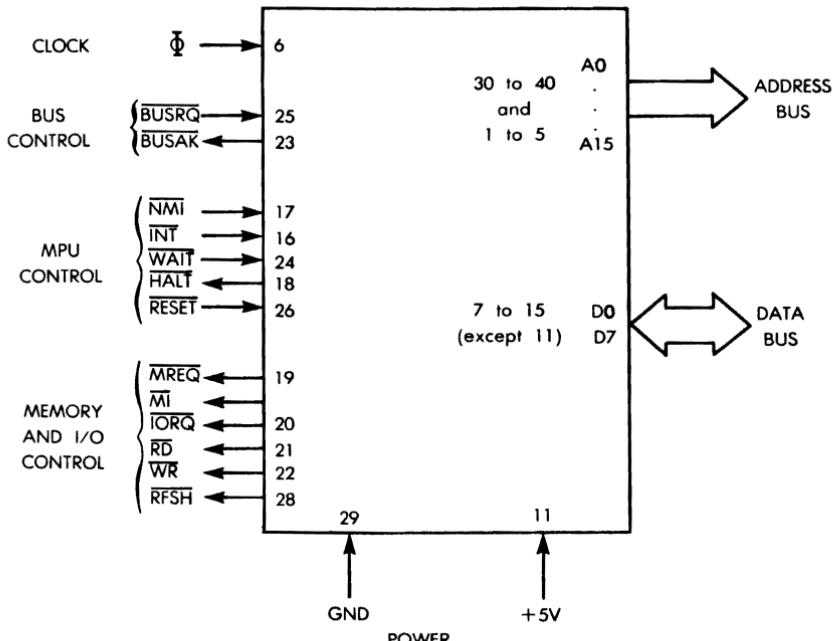


Fig. 2.33: Z80 MPU Pinout

The Z80 Chip

For completeness, the signals of the Z80 microprocessor chip will be examined here. It is not indispensable to understand the functions of

PROGRAMMING THE Z80

the Z80 signals in order to be able to program it. The reader who is not interested in the details of hardware may therefore skip this section. The pinout of the Z80 appears on Fig. 2.33. On the right side of the illustration, the address bus and the data bus perform their usual role, as described at the beginning of this chapter. We will describe here the function of the signals on the control bus. They are shown on the left of Figure 2.33.

The control signals have been partitioned in four groups. They will be described, going from the top of Figure 2.33 towards the bottom.

The clock input is Φ . The Z80 requires an external 330-ohm pull-up resistor. It is connected to the Φ input and to 5 volts. However, at 4 MHz, an external clock driver is required.

The two bus-control signals, BUSRQ and BUSAK, are used to disconnect the Z80 from its busses. They are mainly used by the DMA, but could also be used by another processor in the system. BUSRQ is the bus-request signal. It is issued to the Z80. In response, the Z80 will place its address bus, data bus, and tristate output control signals in the high-impedance state, at the end of the current machine cycle. BUSAK is the acknowledge signal issued by the Z80 once the busses have been placed in the high-impedance state.

Six Z80 control signals are related to its internal status or to its sequencing:

INT and NMI are the two interrupt signals. INT is the usual interrupt request. Interrupts will be described in Chapter 6. A number of input/output devices may be connected to the INT interrupt line. Whenever an interrupt request is present on this line, and when the internal interrupt enable flip-flop (IFF) is enabled, the Z80 will accept the interrupt (provided the BUSRQ is not active). It will then generate an acknowledge signal: IORQ (issued during the M1 state). The rest of the sequence of events is described in Chapter 6.

NMI is the non-maskable interrupt. It is always accepted by the Z80, and it forces the Z80 to jump to location 0066 hexadecimal. It too is described in Chapter 6. (It also assumes that BUSRQ is not active.)

WAIT is a signal used to synchronize the Z80 with slow memory or input/output devices. When active, this signal indicates that the memory or the device is not yet ready for the data transfer. The Z80 CPU will then enter a special wait state until the WAIT signal becomes inactive. It will then resume normal sequencing.

HALT is the acknowledge signal supplied by the Z80 after it has ex-

ecuted the HALT instruction. In this state, the Z80 waits for an external interrupt and keeps executing NOPs to continually refresh memory.

RESET is the signal which usually initializes the MPU. It sets the program counter, register I and R to "0". It disables the interrupt enable flip-flop and sets the interrupt mode to "0". It is normally used after power is applied to the board.

Memory and I/O Control

Six memory and I/O control signals are generated by the Z80. They are: MREQ is the memory request signal. It indicates that the address present on the address bus is valid. A read or write operation can then be performed on the memory.

M1 is machine cycle 1. This cycle corresponds to the fetch cycle of an instruction.

IORQ is the input/output request. It indicates that the I/O address present on bits 0-7 of the address bus is valid. An I/O read or write operation can then be carried out. IORQ is also generated together with M1 when the Z80 acknowledges an interrupt. This information may be used by external chips to place the interrupt response vector on the data bus. (Normal I/O operations never occur during the M1 state. The combination IORQ plus M1 indicates an interrupt-acknowledge situation.)

RD is the read signal.* It indicates the Z80 is ready to read the contents of the data bus into an internal register. It can be used by any external chip, whether memory or I/O, to deposit data onto the data bus.

WR is the write signal.* It indicates that the data bus holds valid data, ready to be written into the specified device.

RFSH is the refresh signal. When RFSH is active, the lower seven bits of the address bus contain a refresh address for dynamic memories. The MREQ signal is then used to perform the refresh by reading the memory.

HARDWARE SUMMARY

This completes our description of the internal organization of the Z80. The exact hardware details of the Z80 are not important here. However, the role of each of the registers is important and should be fully understood before proceeding to the next chapters. The actual instructions available on the Z80 will now be introduced, and basic programming techniques for the Z80 will be presented.

*used in conjunction with MREQ or IOREQ,

3

BASIC PROGRAMMING TECHNIQUES

INTRODUCTION

The purpose of this chapter is to present the basic techniques necessary in order to write a program using the Z80. This chapter will introduce new concepts such as register management, loops, and subroutines. It will focus on programming techniques using only the *internal* Z80 resources, i.e., the registers. Actual programs will be developed, such as arithmetic programs. These programs will serve to illustrate the various concepts presented so far and will use actual instructions. Thus, it will be seen how instructions may be used to manipulate the information between the memory and the MPU, as well as to manipulate information within the MPU itself. The next chapter will then discuss in complete detail the instructions available on the Z80. Chapter 5 will present Addressing Techniques, and Chapter 6 will present the techniques available for manipulating information *outside* the Z80: the Input/Output Techniques.

In this chapter, we will essentially learn by “doing.” By examining programs of increasing complexity, we will learn the role of the various instructions, of the registers, and we will apply the concepts developed so far. However, one important concept will not be presented here; it is the concept of addressing techniques. Because of its apparent complexity, it will be presented separately in Chapter 5.

Let us immediately start writing some programs for the Z80. We will start with arithmetic programs. The “programmer’s model” of the Z80 registers is shown in Figure 3.0.

BASIC PROGRAMMING TECHNIQUES

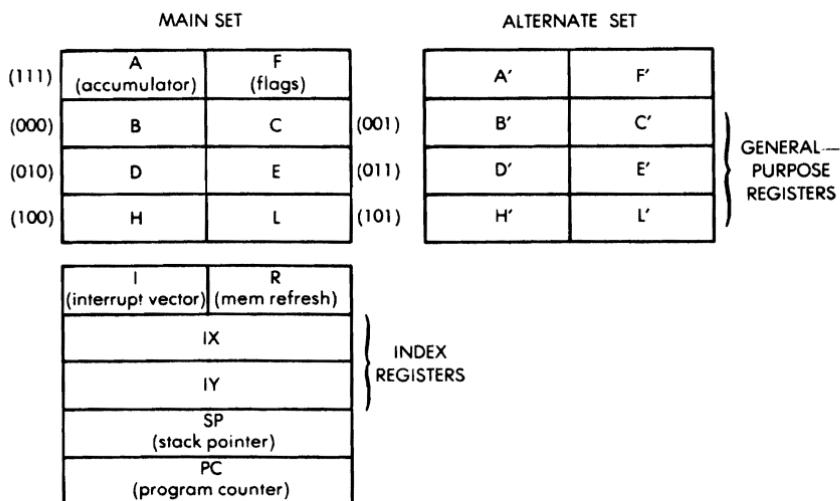


Fig. 3.0: The Z80 Registers

ARITHMETIC PROGRAMS

Arithmetic programs include addition, subtraction, multiplication, and division. The programs presented here will operate on integers. These integers may be positive binary integers or may be expressed in two's complement notation, in which case the left-most bit is the sign bit (see Chapter 1 for a description of the two's complement notation).

8-Bit Addition

We will add two 8-bit operands called OP1 and OP2, respectively stored at memory address ADR1, and ADR2. The sum will be called RES and will be stored at memory address ADR3. This is illustrated in Figure 3.1. The program which will perform this addition is the following:

Instructions	Comments
LD A, (ADR1)	LOAD OP1 INTO A
LD HL, ADR2	LOAD ADDRESS OF OP2 INTO HL
ADD A, (HL)	ADD OP2 TO OP1
LD (ADR 3), A	SAVE RESULT RES AT ADR3

PROGRAMMING THE Z80

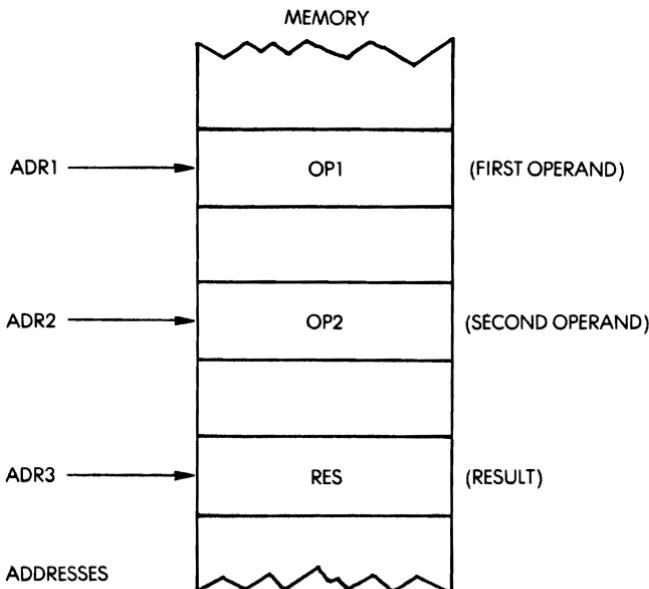


Fig. 3.1: Eight-Bit Addition $RES = OP1 + OP2$

This is our first program. The instructions are listed on the left and comments appear on the right. Let us now examine the program. It is a four-instruction program. Each line is called an *instruction* and is expressed here in symbolic form. Each such instruction will be translated by the *assembler* program into one, two, three or four binary bytes. We will not concern ourselves here with the translation and will only look at the symbolic representation.

The first line specifies loading the contents of ADR1 into the accumulator A. Referring to Figure 3.1, the contents of ADR1 are the first operand, "OP1". This first instruction therefore results in transferring OP1 from the memory into the accumulator. This is shown in Figure 3.2. "ADR1" is a symbolic representation for the actual 16-bit address in the memory. Somewhere else in the program, the ADR1 symbol will be defined. It could, for example, be defined as being equal to the address "100".

This *load* instruction will result in a *read operation* from address 100 (see Figure 3.2), the contents of which will be transferred along the data

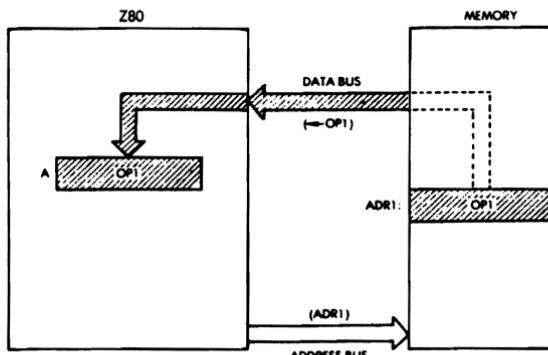


Fig. 3.2: LD A, (ADR1): OPR is Loaded from Memory

bus and deposited inside the accumulator. You will recall from the previous chapter that arithmetic and logical operations operate on the accumulator as one of the source operands. (Refer to the previous chapter for more details.) Since we wish to add the two values OP1 and OP2 together, we must first load OP1 into the accumulator. Then, we will be able to add the contents of the accumulator, i.e., add OP1 to OP2. The right-most field of this instruction is called a *comment* field. It is ignored by the assembler program at translation time, but is provided for program readability. In order to understand what the program does, it is of paramount importance to use good comments. This is called *documenting* a program.

Here the comment is self-explanatory: the value of OP1, which is located at address ADR1, is loaded into the accumulator A.

The result of this first instruction is illustrated by Figure 3.2. The second instruction of our program is:

LD HL, ADR2

It specifies: "Load ADR2 into registers H and L." In order to read the second operand, OP2, from the memory, we must first place its address into a register pair of the Z80, such as H and L. Then, we can add the contents of the memory location whose address is in H and L to the accumulator.

ADD A, (HL)

Referring to Figure 3.1, the contents of memory location ADR2 are OP2, our second operand. The contents of the accumulator are now OP1, our first operand. As a result of the execution of this instruction, OP2 will be fetched from the memory and added to OP1. This is illustrated in Figure 3.3.

PROGRAMMING THE Z80

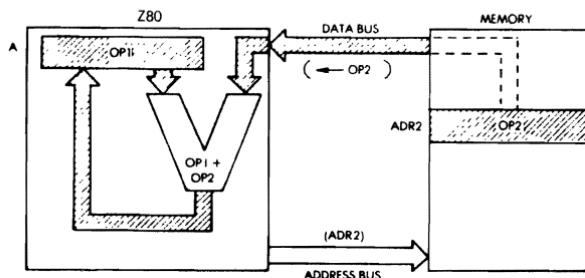


Fig. 3.3: ADD A, (HL)

The sum will be deposited in the accumulator. The reader will remember that, in the case of the Z80, the results of the arithmetic operation are deposited back into the accumulator. In other processors, it may be possible to deposit these results in other registers, or back into the memory.

The sum of OP1 and OP2 is now contained in the accumulator. To complete our program, we simply have to transfer the contents of the accumulator into memory location ADR3, in order to store the results at the specified location. This is performed by the fourth instruction of our program:

LD (ADR3), A

This instruction loads the contents of A into the specified address ADR3. The effect of this final instruction is illustrated by Figure 3.4.

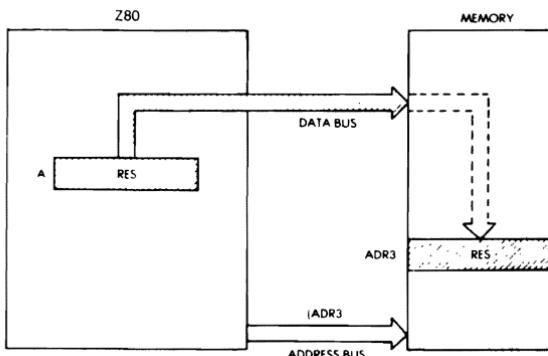


Fig. 3.4: LD (ADR3), A (Save Accumulator in Memory)

Before execution of the ADD operation, the accumulator contained OP1 (see Figure 3.3). After the addition, a new result has been written into the accumulator. It is “OP1 + OP2”. Recall that the contents of any register within the microprocessor, as well as any memory location, remain the same after a read operation has been performed on this register. In other words, reading the contents of a register or memory location does not change its contents. It is only, and exclusively, a *write* operation into this register location that will change its contents. In this example, the contents of memory locations ADR1 and ADR2 remain unchanged throughout the program. However, after the ADD instruction, the contents of the accumulator will have been modified, because the output of the ALU has been written into the accumulator. The previous contents of A are then lost.

Actual numerical addresses may be used instead of ADR1, ADR2, and ADR3. In order to keep symbolic addresses, it will be necessary to use so-called “pseudo-instructions” which specify the value of these symbolic addresses, so that the assembly program may, during translation, substitute the actual physical addresses. Such pseudo-instructions could be, for example:

```
ADR1 = 100H
ADR2 = 120H
ADR3 = 200H
```

Exercise 3.1: Now close this book. Refer only to the list of instructions at the end of the book. Write a program which will add two numbers stored at memory locations LOC1 and LOC2. Deposit the results at memory location LOC3. Then, compare your program to the one above.

16-Bit Addition

An 8-bit addition will only allow the addition of 8-bit numbers, i.e., numbers between 0 and 255, if absolute binary is used. For most practical applications it is necessary to add numbers having 16 bits or more, i.e., to use *multiple precision*. We will here present examples of arithmetic on 16-bit numbers. They can be readily extended to 24, 32 bits or more (always multiples of 8 bits). We will assume that the first operand is stored at memory locations ADR1 and ADR1-1. Since OP1 is a 16-bit number this time, it will require two 8-bit memory locations. Similarly,

PROGRAMMING THE Z80

OP2 will be stored at ADR2 and ADR2-1. The result is to be deposited at memory addresses ADR3 and ADR3-1. This is illustrated in Figure 3.5. H indicates the high half (bits 8 through 15), while L indicates the low half (bits 0 through 7).

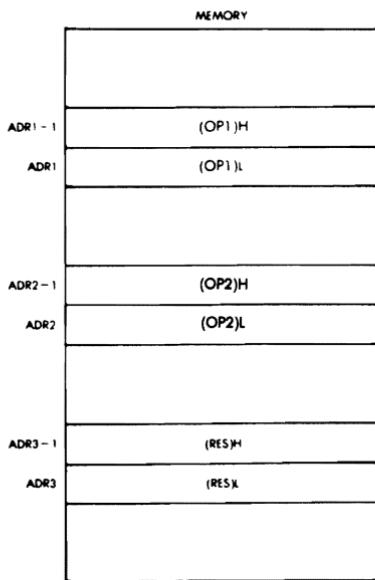


Fig. 3.5: 16-Bit Addition—The Operands

The logic of the program is exactly like the previous one. First, the lower half of the two operands will be added, since the microprocessor can only add on 8 bits at a time. Any carry generated by the addition of these low order bytes will automatically be stored in the internal carry bit ("C"). Then, the high order half of the two operands will be added together along with any carry, and the result will be saved in the memory. The program appears below:

LD A, (ADR1)	LOAD LOW HALF OF OP1
LD HL, ADR2	ADDRESS OF LOW HALF OF OP2
ADD A, (HL),	ADD OP1 AND OP2 LOW
LD (ADR3), A	STORE RESULT, LOW
LD A, (ADR1-1)	LOAD HIGH HALF OF OP1
DEC HL	ADDRESS OF HIGH HALF OF OP2
ADC A, (HL)	(OP1 + OP2) HIGH + CARRY
LD (ADR3-1), A	STORE RESULT, HIGH

The first four instructions of this program are identical to the ones used for the 8-bit addition in the previous section. They result in adding the least significant halves (bits 0-7) of OP1 and OP2. The sum, called "RES" is stored at memory location ADR3 (see Figure 3.5).

Automatically, whenever an addition is performed, any resulting carry (whether "0" or "1") is saved in the carry bit C of the flags register (register F). If the two numbers do generate a carry, then the C bit will be equal to "1" (it will be set). If the two 8-bit numbers do not generate any carry, the value of the carry bit will be "0".

The next four instructions of the program are essentially like those used in the previous 8-bit addition program. This time they add together the most significant half (or high half, i.e., bits 8-15) of OP1 and OP2, plus any carry, and store the result at address ADR3-1.

After execution of this 8-instruction program, the 16-bit result is stored at memory locations ADR3 and ADR3-1, as specified. Note, however, that there is one difference between the second half of this program and the first half. The "*ADD*" instruction which has been used is not the same as in the first half. In the first half of this program (the 3rd instruction), we had used the "*ADD*" instruction. This instruction adds the two operands, regardless of the carry. In the second half, we use the "*ADC*" instruction, which adds the two operands together, plus any carry that may have been generated. This is necessary in order to obtain the correct result. The addition initially performed on the low operands may result in a carry. Such a possible carry must be taken into account in the second half of the addition.

The question which comes naturally then is: what if the addition of the high half of the operands also results in a carry? There are two possibilities: the first one is to assume that this is an error. This program is then designed to work for results of only up to 16 bits, but not 17. The other one is to include additional instructions to test explicitly for the possibility of a carry at the end of this program. This is a choice which the programmer must make, the first of many choices.

Note: we have assumed here that the high part of the operand is stored "on top of" the lower part, i.e., at the lower memory address. This need not necessarily be the case. In fact, addresses are stored by the Z80 in the reverse manner: the low part is first saved in the memory, and the high part is saved in the next memory location. In order to use a common convention for both addresses and data, it is recommended that data also be kept with the low part on top of the high part. This is illustrated in Figure 3.6.

PROGRAMMING THE Z80

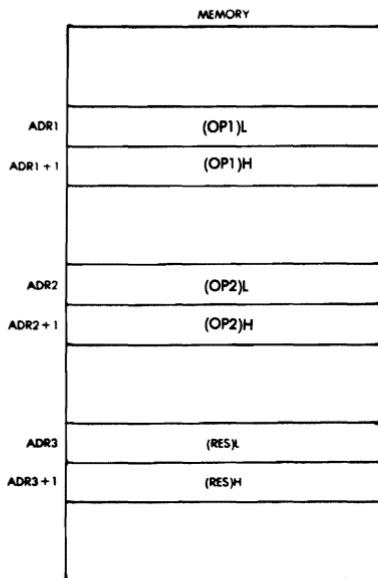


Fig. 3.6: Storing Operands in Reverse Order

When operating on multibyte operand, it is important to keep in mind two essential conventions:

- the order in which data is stored in the memory.
 - where data pointers are pointing: low byte or high byte.
- Exercises 3.2 and 3.3 are designed to clarify this point.

Exercise 3.2: Rewrite the 16-bit addition program above with the memory layout indicated in Figure 3.6.

Exercise 3.3: Assume now that ADR1 does not point to the lower half of OP1 (as in Figures 3.5 or 3.6), but points to the higher part of OP1. This is illustrated in Figure 3.7. Again, write the corresponding program.

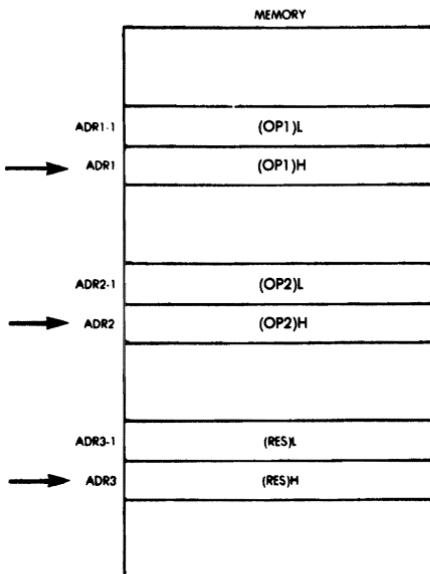


Fig. 3.7: Pointing to the High Byte

It is the programmer, i.e., you, who must decide how to store 16-bit numbers (i.e., low part or high part first) and also whether your address references point to the lower or to the higher half of such numbers. This is another choice which you will learn to make when designing algorithms or data structures.

The programs presented above are traditional programs, using the accumulator. We will now present an alternative program for the 16-bit addition that does not use the accumulator, but instead uses some of the special 16-bit instructions available on the Z80. Operands will be assumed to be stored as indicated in Figure 3.5. The program is:

LD HL, (ADR1)	LOAD HL WITH OP1
LD BC, (ADR2)	LOAD BC WITH OP2
ADD HL, BC	ADD 16 BITS
LD (ADR3), HL	STORE RES INTO ADR3

Note how much shorter this program is, compared to our previous version. It is more "elegant." *In a limited manner, the Z80 allows registers H and L to be used as a 16-bit accumulator.*

Exercise 3.4: Using the 16-bit instructions which have just been introduced, write an addition program for 32-bit operands, assuming that operands are stored as shown in Figure 3.8. (The answer appears below.)

Answer :

```
LD HL, (ADR1)
LD BC, (ADR2)
ADD HL, BC
LD (ADR3)
LD HL, (ADR1 + 2)
LD BC, (ADR2 + 2)
ADC HL, BC
LD (ADR3 + 2)
```

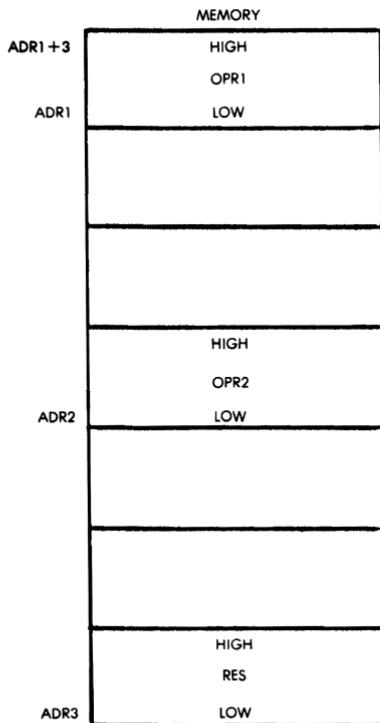


Fig. 3.8: A 32-Bit Addition

Now that we have learned to perform a binary addition, let us turn to subtraction.

Subtracting 16-Bit Numbers

Doing an 8-bit subtract would be too simple. Let us keep it as an exercise and directly perform a 16-bit subtract. As usual, our two numbers, OP1 and OP2, are stored at addresses ADR1 and ADR2. The memory layout will be assumed to be that of Figure 3.6. In order to subtract, we will use a subtract operation (SBC) instead of an add operation (ADD).

Exercise 3.5: Now write a subtraction program.

The program appears below. The data paths are shown in Figure 3.9.

LD HL, (ADR1)	OP1 INTO HL
LD DE, (ADR2)	OP2 INTO DE
AND A	CLEAR CARRY
SBC HL, DE	OP1 — OP2
LD (ADR3), HL	RES INTO ADR3

The program is essentially like the one developed for 16-bit addition. However, the Z80 instruction-set has two types of additions on double registers: ADD and ADC, but only one type of subtraction: SBC.

As a result, two changes can be noted.

PROGRAMMING THE Z80

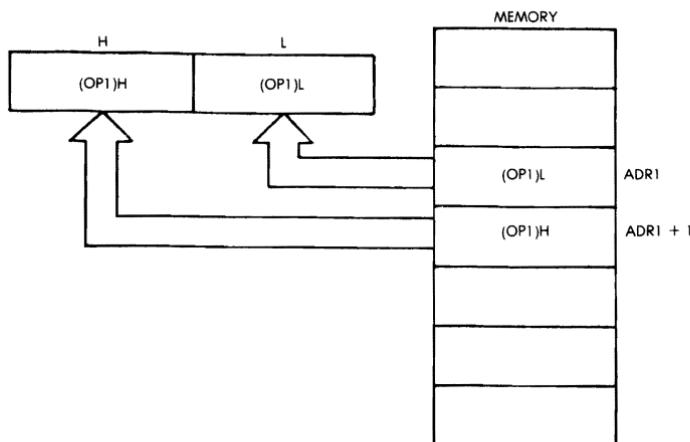


Fig. 3.9: 16-Bit Load — LD HL, (ADR1)

A first change is the use of SBC instead of ADD.

The other change is the “AND A” instruction, used to clear the carry flag prior to the subtraction. This instruction does not modify the value of A.

This precaution is necessary because the Z80 is equipped with two modes of addition, with and without carry on the H and L register, but with only one mode of subtraction, the SBC instruction of “subtract with carry” when operating on the HL register pair. Because SBC automatically takes into account the value of the carry bit, it must be set to 0 prior to starting the subtraction. This is the role of the “AND A” instruction.

Exercise 3.6: Rewrite the subtraction program without using the specialized 16-bit instruction.

Exercise 3.7: Write the subtract program for 8-bit operands.

It must be remembered that in the case of two’s complement arithmetic, the final value of the carry flag has no meaning. If an overflow condition has occurred as a result of the subtraction, then the overflow bit (bit V) of the flags register will have been set. It can then be tested.

The examples just presented are simple binary additions or subtractions. However, another type of arithmetic may be necessary; it is BCD arithmetic.

BCD ARITHMETIC

8-Bit BCD Addition

The concept of BCD arithmetic has been presented in Chapter 1. Let us recall its features. It is essentially used for business applications where it is imperative to retain every significant digit in a result. In the BCD notation, a 4-bit nibble is used to store one decimal digit (0 through 9). As a result, every 8-bit byte may store two BCD digits. (This is called *packed BCD*). Let us now add two bytes each containing two BCD digits.

In order to identify the problems, let us try some numeric examples first.

Let us add “01” and “02”:

“01” is represented by: 0000 0001

“02” is represented by: 0000 0010

The result is: 0000 0011

This is the BCD representation for “03”. (If you feel unsure of the BCD equivalent, refer to the conversion table at the end of the book.) Everything worked very simply in this case. Let us now try another example.

“08” is represented by 0000 1000

“03” is represented by 0000 0011

Exercise 3.8: Compute the sum of the two numbers above in the BCD representation. What do you obtain? (answer follows)

If you obtain “0000 1011”, you have computed the *binary* sum of 8 and 3. You have indeed obtained 11 in *binary*. Unfortunately, “1011” is an *illegal code in BCD*. You should obtain the *BCD* representation of “11”, i.e., 0001 0001!

The problem stems from the fact that the BCD representation uses only the first ten combinations of 4 digits in order to encode the decimal symbols 0 through 9. The remaining six possible combinations of 4 digits are unused, and the illegal “1011” is one such combination. In other words, whenever the sum of two BCD digits is greater than 9,

PROGRAMMING THE Z80

then one must add 6 to the result in order to skip over the 6 unused codes.

Add the binary representation of "6" to 1011:

$$\begin{array}{r} 1011 \quad (\text{illegal binary result}) \\ + 0110 \quad (+6) \\ \hline 0001\ 0001 \end{array}$$

The result is:

This is, indeed, "11" in the BCD notation! We now have the correct result.

This example illustrates one of the basic difficulties of the BCD mode. One must compensate for the six missing codes. A special instruction, "DAA", called "decimal adjust," must be used to adjust the result of the binary addition. (Add 6 if the result is greater than 9.)

The next problem is illustrated by the same example. In our example, the carry will be generated from the lower BCD digit (the right-most one) into the left-most one. This internal carry must be taken into account and added to the second BCD digit. The addition instruction takes care of this automatically. However, it is often convenient to detect this internal carry from bit 3 to bit 4 (the "half-carry"). The H flag is provided for this purpose.

As an example, here is a program to add the BCD numbers "11" and "22":

LD A, 11H	LOAD LITERAL BCD '11'
ADD A, 22H	ADD LITERAL BCD '22'
DAA	DECIMAL ADJUST RESULT
LD (ADR), A	STORE RESULT

In this program, we are using a new symbol "H". The "H" sign within the operand field of the instruction specifies that the data it follows is expressed in hexadecimal notation. The hexadecimal and the BCD representations for digits "0" through "9" are identical. Here we wish to add the literals (or constants) "11" and "22". The result is stored at the address ADR. When the operand is specified as part of the instruction, as it is in the above example, this is called *immediate addressing*. (The various addressing modes will be discussed in detail in Chapter 5.) Storing the result at a specified address, such as LD (ADR), A is called *absolute addressing* when ADR represents a 16-bit address.

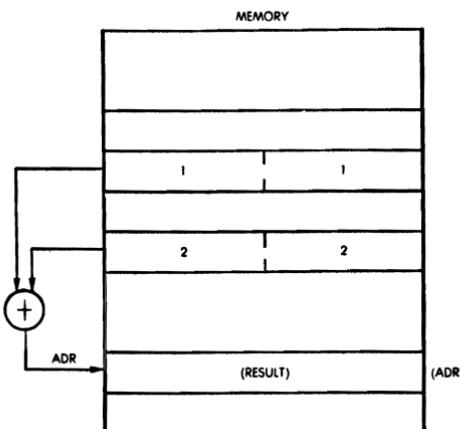


Fig. 3.10: Storing BCD Digits

This program is analogous to the 8-bit binary addition, but uses a new instruction: "DAA". Let us illustrate its role in an example. We will first add "11" and "22" in BCD:

$$\begin{array}{r}
 00010001 \quad (11) \\
 + 00100010 \quad (22) \\
 \hline
 = \underbrace{00110011}_{\begin{matrix} 3 & 3 \end{matrix}} \quad (33)
 \end{array}$$

The result is correct, using the rules of binary addition.

Let us now add "22" and "39", by using the rules of *binary* addition:

$$\begin{array}{r}
 00100010 \quad (22) \\
 + 00111001 \quad (39) \\
 \hline
 = \underbrace{01011011}_{\begin{matrix} 5 & ? \end{matrix}}
 \end{array}$$

"1011" is an *illegal BCD code*. This is because BCD uses only the first 10 binary codes, and "skips over" the next 6. We must do the same, i.e. add 6 to the result:

$$\begin{array}{r}
 01011011 \quad (\text{binary result}) \\
 + 0110 \quad (6) \\
 \hline
 = \underbrace{01100001}_{\begin{matrix} 6 & 1 \end{matrix}} \quad (61)
 \end{array}$$

This is the correct BCD result.

PROGRAMMING THE Z80

Exercise 3.9: Could we move the DAA instruction in the program after the instruction LD (ADR), A?

BCD Subtraction

BCD subtraction is, in appearance, complex. In order to perform a BCD subtraction, one must add the *ten's complement* of the number, just as one adds the two's complement of a number to perform a binary subtract. The ten's complement is obtained by computing the complement to 9, then adding "1". This requires typically three to four operations on a standard microprocessor. However, the Z80 is equipped with a powerful DAA instruction which simplifies the program.

The DAA instruction automatically adjusts the value of the result in the accumulator, depending on the value of the C, H and N flags before DAA, to the correct value. (See the next chapter for more details on DAA.)

16-Bit BCD Addition

16-bit addition is performed just as simply as in the binary case. The program for such an addition appears below:

LD A, (ADR1)	LOAD (OP1) L INTO A
LD HL, (ADR2)	LOAD ADR2 INTO HL
ADD A, (HL)	(OP1 + OP2) LOW
DAA	DECIMAL ADJUST
LD (ADR3), A	STORE (RESULT) LOW
LD A, (ADR1 + 1)	LD (OP1) H INTO A
INC HL	POINT TO ADR2 + 1
ADC A, (HL)	(OP1 + OP2) HIGH + CARRY
DAA	DECIMAL ADJUST
LD (ADR3 + 1), A	STORE (RESULT) HIGH

Packed BCD Subtract

Elementary BCD addition and subtraction have been described. However, in actual practice, BCD numbers include any number of bytes. As a simplified example of a packed BCD subtract, we will assume that the two numbers N1 and N2 include the same number of BCD bytes. The number of bytes is called COUNT. The register and

memory allocation is shown in Figure 3.11. The program appears below:

BCDPAK	LD B, COUNT	
	LD DE, N2	
	LD HL, N1	
	AND A	CLEAR CARRY
MINUS	LD A, (DE)	N2 BYTE
	SBC A, (HL)	$N2 - N1$
	DAA	
	LD (HL), A	STORE RESULT
	INC DE	
	INC HL	
	DJNZ MINUS	DEC B, LOOP UNTIL $B = 0$.

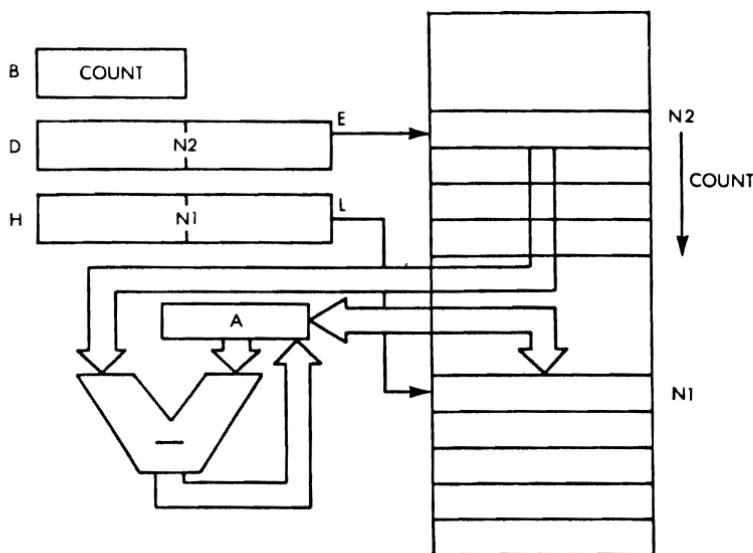


Fig. 3.11: Packed BCD Subtract: $N1 \leftarrow N2 - N1$

N1 and N2 represent the addresses where the BCD numbers are stored. These addresses will be loaded in register pairs DE and HL:

BCDPAK	LD B, COUNT	
	LD DE, N2	
	LD HL, N1	

PROGRAMMING THE Z80

Then, in anticipation of the first subtraction, the carry bit must be cleared. It has been pointed out that the carry bit can be cleared in a number of equivalent ways. Here, for example, we use:

AND A

The first byte of N2 is loaded into the accumulator, then the first byte of N1 is subtracted from it. The DAA instruction is then used, to obtain the correct BCD value:

```
MINUS LD A, (DE)
        SBC A, (HL)
        DAA
```

The result is then stored into N1:

```
LD (HL), A
```

Finally, the pointers to the current byte are incremented:

```
INC DE
INC HL
```

The counter is decremented and the subtraction loop is executed until it reaches the value “0”:

```
DJNZ MINUS
```

The DJNZ instruction is a special Z80 instruction which decrements register B and jumps if it is not zero, in a single instruction.

Exercise 3.10: Compare the program above to the one for the 16-bit binary addition. What is the difference?

Exercise 3.11: Can you exchange the roles of DE and HL? (Hint: Be careful with SBC.)

Exercise 3.12: Write the subtraction program for a 16-bit BCD.

BCD Flags

In BCD mode, the carry flag set as the result of an addition indicates the fact that the result is larger than 99. This is not like the two's complement situation, since BCD digits are represented in true binary. Conversely, the presence of the carry flag after a subtraction indicates a borrow.

Instruction Types

We have now used two types of microprocessor instructions. We

have used LD, which loads the accumulator from the memory address, or stores its contents at the specified address. This is a *data transfer* instruction.

Next, we have used *arithmetic* instructions, such as ADD, SUB, ADC and SBC. They perform addition and subtraction operations. More ALU instructions will be introduced soon in this chapter.

Still other types of instructions are available within the microprocessor which we have not used yet. They are in particular “jump” instructions, which will modify the order in which the program is being executed. This new type of instruction will be introduced in our next example. Note that jump instructions are often called “branch” for conditional situations, i.e. instances where there is a logical choice in the program. The “branch” derives its name from the analogy to a tree, and implies a fork in the representation of the program.

MULTIPLICATION

Let us now examine a more complex arithmetic problem: the multiplication of binary numbers. In order to introduce the algorithm for a binary multiplication, let us start by examining a usual decimal multiplication: We will multiply 12 by 23.

$$\begin{array}{r}
 12 \quad (\text{Multiplicand}) \\
 \times 23 \quad (\text{Multiplier}) \\
 \hline
 36 \quad (\text{Partial Product}) \\
 + 24 \\
 \hline
 = 276 \quad (\text{Final Result})
 \end{array}$$

The multiplication is performed by multiplying the right-most digit of the multiplier by the multiplicand, i.e., “3” \times “12”. The partial product is “36”. Then one multiplies the next digit of the multiplier, i.e., “2”, by “12”. “24” is then added to the partial product.

But there is one more operation: 24 is *offset to the left* by one position. We will say that 24 is *shifted left* by one position. Equivalently, we could have said that the partial product (36) had been *shifted one position to the right* before adding.

The two numbers, correctly shifted, are then added and the sum is 276. This is simple. The binary multiplication is performed in exactly the same way.

PROGRAMMING THE Z80

Let us look at an example. We will multiply 5×3 :

$$\begin{array}{r} (5) & 101 & (\text{MPD}) \\ (3) & \times & 011 & (\text{MPR}) \\ & & \hline & 101 & (\text{PP}) \\ & & 101 \\ & & 000 \\ \hline (15) & 01111 & (\text{RES}) \end{array}$$

In order to perform the multiplication, we operate exactly as we did above. The formal representation of this algorithm appears in Figure 3-12. It is a flowchart for the algorithm, our first flowchart. Let us examine it more closely.

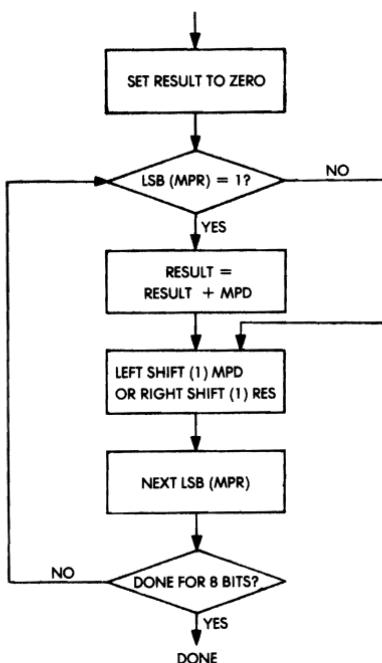


Fig. 3.12: The Basic Multiplication Algorithm—Flowchart

This flowchart is a symbolic representation of the algorithm we have just presented. Every rectangle represents an order to be carried out. It will be translated into one or more program instructions. Every

diamond-shaped symbol represents a test being performed. This will be a *branching point* in the program. If the test succeeds, we will branch to a specified location. If the test does not succeed, we will branch to another location. The concept of branching will be explained later, in the program itself. The reader should now examine this flowchart and ascertain that it does indeed exactly represent the algorithm which has been presented. Note that there is an arrow coming out of the last diamond at the bottom of the flowchart, back to the first diamond on top. This is because the same portion of the flowchart will be executed eight times, once for every bit of the multiplier. Such a situation, where execution will restart at the same point, is called a *program loop* for obvious reasons.

Exercise 3.13: Multiply “4” by “7” in binary, using the flowchart, and verify that you obtain “28”. If you do not, try again. It is only if you obtain the correct result that you are ready to translate this flowchart into a program.

8-By-8 Multiplication

Let us now translate this flowchart into a program for the Z80. The complete program appears in Figure 3.13. We are going to study it in detail. As you will recall from Chapter 1, programming consists here of translating the flowchart of Figure 3.12 into the program of Figure 3.13. Each of the boxes in the flowchart will be translated by one or more instructions.

It is assumed that MPR and MPD already have a value.

MPY88	LD	BC, (MPRAD)	LOAD MULTIPLIER INTO C
	LD	B, 8	B IS BIT COUNTER
	LD	DE, (MPDAD)	LOAD MULTIPLICAND INTO E
	LD	D, 0	CLEAR D
	LD	HL, 0	SET RESULT TO 0
MULT	SRL	C	SHIFT MULTIPLIER BIT INTO CARRY
	JR	NC, NOADD	TEST CARRY
	ADD	HL, DE	ADD MPD TO RESULT
NOADD	SLA	E	SHIFT MPD LEFT
	RL	D	SAVE BIT IN D
	DEC	B	DECREMENT SHIFT COUNTER
	JP	NZ, MULT	DO IT AGAIN IF COUNTER ≠ 0
	LD	(RESAD), HL	STORE RESULT

Fig. 3.13: 8 × 8 Multiplication Program

PROGRAMMING THE Z80

The first box of the flowchart is an *initialization box*. It is necessary to set a number of registers or memory locations to “0”, as this program will require their use. The registers which will be used by the multiplication program appear in Figure 3.14.

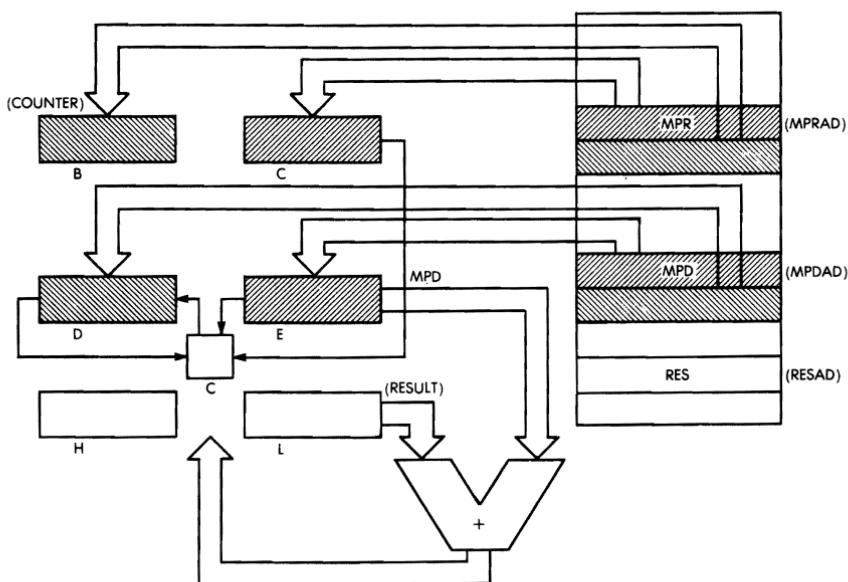


Fig 3.14: 8×8 Multiplication—The Registers

Three register pairs of the Z80 are used for the multiplication program. The 8-bit multiplier is assumed to reside at memory address MPRAD. The multiplicand MPD is assumed to reside at memory address MPDAD. The multiplier and the multiplicand respectively will be loaded into registers C and E (see Figure 3.14). Register B will be used as a counter.

Registers D and E will hold the multiplicand as it is shifted left one bit at a time.

Note that, even though only C and E need to be loaded initially, a 16-bit load must be used, so that B and D will also be loaded from memory, and will have to be reset respectively to “8” and to “0”.

Finally, the results of an 8-bit by 8-bit multiplication may require up to 16 bits. This is because $2^8 \times 2^8 = 2^{16}$. Two registers must therefore be reserved for the result. They are registers H and L, as indicated on Figure 3.14.

The first step is to load registers B, C, and E with the appropriate contents, and to initialize the result (the partial product) to the value "0" as specified by the flowchart of Figure 3.12. This is accomplished by the following instructions:

```
MPY88 LD BC, (MPRAD)
        LD B, 8
        LD DE, (MPDAD)
        LD D, 0
        LD HL, 0
```

The first three instructions respectively load MPR into the register pair BC, the value "8" into register B, and MPD into the register pair DE. Since MPR and MPD are 8-bit words, they are, in fact, loaded into registers C and E respectively, while the next words in the memory after MPR and MPD get loaded into B and D. This is shown in Figure 3.15 and 3.16. The next instruction will zero the contents of D.

In this multiplication program, the multiplicand will be shifted left before being added to the result (remember that, optionally, it is possible to shift the result right instead, as indicated in the fourth box of the flowchart of Figure 3.12). The multiplicand MPD will be shifted into register D at each step. This register D must therefore be initialized to the value "0". This is accomplished by the fourth instruction. Finally, the fifth instruction sets the contents of registers H and L to 0 in a single instruction.

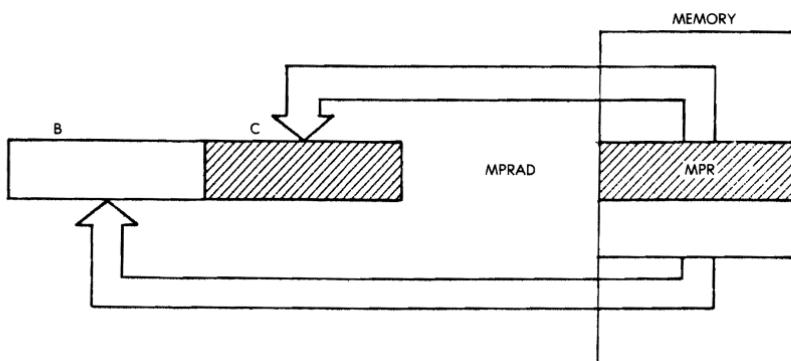
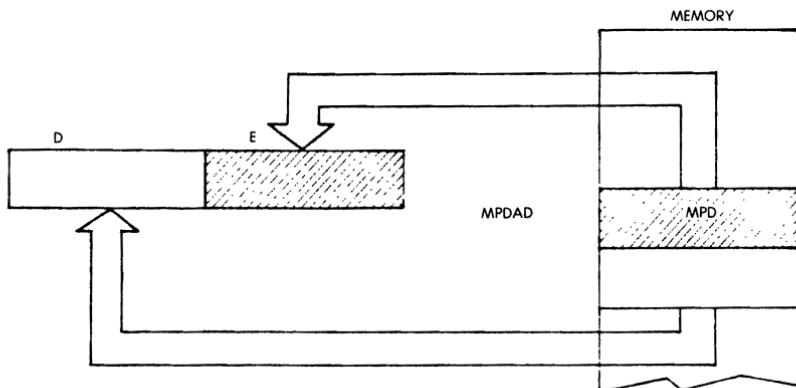


Fig. 3.15: LD BC, (MPRAD)

**Fig. 3.16: LD DE, (MPDAD)**

Referring back to the flowchart of Figure 3.12, the next step is to test the least significant bit (the right-most bit) of the multiplier MPR. If this bit is a "1", then the value of MPD must be added to the partial result, otherwise it will not be added. This is accomplished by the next three instructions:

```
MULT SRL C
JR NC, NOADD
ADD HL, DE
```

The first problem we must solve is how to test the least significant bit of the multiplier, contained in register C. We could here use the BIT instruction of the Z80, which allows testing any bit in any register. However, in this case, we would like to construct a program as simple as possible, using a loop. If we were using the BIT instruction here, we would first test bit 0, then later test bit 1, and so on until we reached bit 7. This would require a different instruction every time, and a simple loop could not be used. In order to shorten the length of the program, we must use a different instruction. Here we are using a *shift* instruction.

Note: There is a way to use the BIT instruction and a loop, but this would require the program to modify itself, a practice we will avoid.

SRL is a new type of operation within the arithmetic and logical unit. It stands for "shift right logical." A *logical shift to the right* is characterized by the fact that a "0" comes into bit position 7. This can be contrasted to an *arithmetic shift to the right*, where the bit coming into position 7 is identical to the previous value of bit 7. The different types of shift operations will be described in the next chapter. The effect of the SRL C instruction is illustrated in Figure 3.14 by an arrow coming out of register C and into the square used to designate the carry bit (also called "C"). At this point, the right-most bit of the MPR will be in the carry bit C, where it can be tested.

The next instruction, "JR NC, NOADD", is a *jump* operation. It means "jump on no carry" (NC) to the address (the label) NOADD. If the contents of the carry bit are "0" (no carry), then the program will jump to the address NOADD. If the contents of C are "1" (the carry bit is set), then no branch will occur, and the next sequential instruction will be executed, i.e., the instruction "ADD HL, DE" will be executed.

This instruction specifies that the contents of D and E be added to H and L, with the result in H and L. Since E contains the multiplicand MPD (see Figure 3.14), this adds the multiplicand to the partial result.

At this point, regardless of whether MPD has been added to the result or not, the multiplicand must be shifted left (this is the fourth box in the flowchart of Figure 3.12). This is accomplished by:

NOADD SLA E

SLA stands for "shift left arithmetic." It has just been explained above that there are two types of shift operations, a logical shift and an arithmetic shift. This is the arithmetic one. In the case of a left shift, an SLA specifies that the bit coming into the right part of the register (the least significant bit) be a "0" (just as in the case of an SRL before).

As an example, let us assume that the initial contents of register E were 00001001. After the SLA instruction, the contents of E will be 00010010. And the contents of the carry bit will be 0.

However, looking back at Figure 3.14, we really want to shift the most significant bit (called the MSB) of E directly into D (this is illustrated by the arrow on the illustration coming from E into D). However, there is no instruction which will shift a double register such as D and E in one operation. Once the contents of E have been shifted, the left-most bit has "fallen into" the carry bit. We must collect this bit from the carry bit and shift it into register D. This is accomplished by the next instruction:

RL D

PROGRAMMING THE Z80

RL is still another type of shift operation. It stands for “rotate left.” In a *rotation* operation, as opposed to a *shift* operation, this bit coming into the register is the contents of the carry bit C (see Figure 3.17). This is exactly what we want. The contents of the carry bit C are loaded into the right-most part of D, and we have effectively transferred the left-most bit of E.

This sequence of two instructions is illustrated in Figure 3.18. It can be seen that the bit marked by an X in the most significant position of E will first be transferred into the carry bit, then into the least significant position of D. Effectively, it will have been shifted from E into D.

At this point, referring back to the flowchart of Figure 3.12, we must point to the next bit of MPR and check for the eighth bit. This is accomplished by decrementing the byte counter, contained in register B (see Figure 3.14). The register is decremented by:

DEC B

This is a *decrement* instruction, which has the obvious effect.

Finally, we must check whether the counter has decremented to the value zero. This is accomplished by checking the value of the Z bit. The reader will recall that the Z (zero) flag indicates whether the previous arithmetic operation (such as a DEC operation) has produced a zero result. However, note that DEC HL, DEC BC, DEC DE, DEC IX, DEC SP do not affect the Z flag. If the counter is not “0”, the operation is not finished, and we must execute this program loop again. This is accomplished by the next instruction:

JP NZ MULT

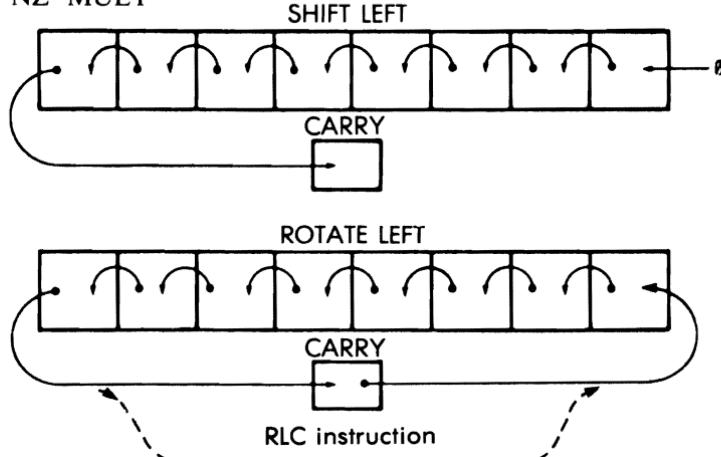


Fig. 3.17: Shift and Rotate

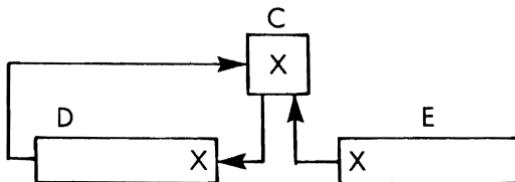


Fig. 3.18: Shifting from E into D

This is a jump instruction which specifies that whenever the Z bit is not set (NZ stands for non-zero), a jump occurs to location MULT. This is the *program loop*, which will be executed repeatedly until B decrements to the value 0. Whenever B decrements to the value 0, the Z bit will be set, and the JP NZ instruction will fail. This will result in the next sequential instruction being executed, namely:

LD (RESAD), HL

This instruction merely saves the contents of H and L, i.e., the result of the multiplication, at address RESAD, the address specified for the result. Note that this instruction will transfer the contents of both registers H and L into two consecutive memory locations, corresponding to addresses RESAD and RESAD + 1. It saves 16 bits at a time.

Exercise 3.14: Could you write the same multiplication program using the *BIT* instruction (described in the next chapter) instead of the *SRL C* instruction? What would be the disadvantage?

Let us now improve the program, if possible:

Exercise 3.15: Can JR be substituted for JP at the end of the program? If so, what is the advantage?

Exercise 3.16: Can you use DJNZ to shorten the end of the program?

Exercise 3.17: Examine the two instructions: LD D, 0 and LD HL, 0 at the beginning of the program. Can you substitute:

XOR A
LD D, A
LD H, A
LD L, A

If so, what is the impact on size (number of bytes) and speed?

Note that, in most cases, the program that we have just developed will be a subroutine and the final instruction in the subroutine will be RET (return). The subroutine mechanism will be explained later in this chapter.

Important Self-Test

This is the first significant program we have encountered so far. It includes many different types of instructions, including transfer instructions (LD), arithmetic operations (ADD), logical operations (SRL, SLA, RL), and jump operations (JR, JP). It also implements a program loop, in which the lower seven instructions, starting at address MULT, are executed repeatedly. In order to understand programming, it is essential to understand the operation of such a program in complete detail. The program is much longer than the previous simple arithmetic programs we have developed so far, and it should be studied in detail. An important exercise will now be proposed. The reader is strongly urged to do this exercise completely and correctly before proceeding. This will be the only real proof that the concepts presented so far have been understood. If a correct result is obtained, it will mean that you have really understood the mechanism by which instructions manipulate information in the microprocessor, transfer it between the memory and the registers, and process it. If you do not obtain the correct result, or if you do not do this exercise, it is likely that you will experience difficulties later in writing programs yourself. Learning to program requires personal practice. Please pause now, take a piece of paper, or use the illustration of Figure 3.19, and do the following exercise:

Exercise 3.18: Every time that a program is written, it should be verified by hand, in order to ascertain that its results will be correct. We are going to do just that: the goal of this exercise is to fill in the table of Figure 3.19 completely and accurately.

LABEL	INSTRUCTION	B	C	C (CARRY)	D	E	H	L

Fig. 3.19: Form for Multiplication Exercise

You may want to write directly on Figure 3.19 or make a copy of it. You must determine the contents of every relevant register in the Z80 after the execution of each instruction in the program, from beginning to end. All the registers used by the program of Figure 3.13 are shown in Figure 3.19. From left to right, they are registers B and C, the carry C, registers D and E, and, finally, registers H and L. On the left part of this illustration, fill in the label, if applicable, and then the instructions

PROGRAMMING THE Z80

being executed. On the right of the instruction, fill in the contents of each register after execution of the instruction. Whenever the contents of a register are not known (indefinite), you may use dashes to represent its contents. Let us start filling in this table together. You will then have to fill it out by yourself until the end. The first line appears below:

LABEL	INSTRUCTION	B	C	C	D	E	H	L
MPY88	LD BC,(0200)	-- 00	-- 03	-	--	--	--	--

Fig. 3.20: Multiplication: After One Instruction

We will assume here that we are multiplying “3” (MPR) by “5” (MPD).

The first instruction to be executed is “LD BC, (MPRAD)”. The contents of memory location MPRAD is loaded into registers B and C. It has been assumed that MPR is equal to 3, i.e., “00000011”. After execution of this instruction, the contents of register C have been set to “3”. Note that this instruction will also result in loading register B with whatever followed MPR in the memory. However, the next instruction in the program will take care of this by loading register B with “8”, as shown in Figure 3.21. Note that, at this point, the contents of D and E and H and L are still undefined, and this is indicated by dashes. The LD instruction does not condition the carry bit, so that the contents of the carry bit C are undefined. This is also indicated by a dash.

LABEL	INSTRUCTION	B	C	C	D	E	H	L
MPY88	LD BC,(0200)	-- 00	-- 03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--

Fig. 3.21: Multiplication: After Two Instructions

The situation after the execution of the first five instructions of the program (just before the MULT) is shown in Figure 3.22.

LABEL	INSTRUCTION	B	C	C	D	E	H	L
MPY88	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00

Fig. 3.22: Multiplication: After Five Instructions

The SRL instruction will perform a logical shift right, and the right-most bit of MPR will fall into the carry bit. You can see in Figure 3.23 that the contents of MPR after the shift is “0000 0001”. The carry bit C is now set to “1”. The other registers are unchanged by this operation. Please continue to fill out the chart by yourself.

A second iteration is shown at the end of this chapter in Fig. 3.41.

LABEL	INSTRUCTION	B	C	C	D	E	H	L
MPY88	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
NOADD	ADD HL,DE	08	01	0	00	05	00	05
	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05

Fig. 3.23: One Pass Through The Loop.

A complete listing showing the contents of all the Z80 registers and the flags is shown in Fig. 3.39 at the end of this chapter for the complete multiplication. A hex or decimal listing is shown in Fig. 3.40.

Programming Alternatives

The program that we have just developed could have been written in many other ways. As a general rule, every programmer can usually find ways to modify, and often improve, a program. For example, we have shifted the multiplicand left before adding. It would have been mathematically equivalent to shift the result one position to the right before adding it to the multiplicand. As a matter of fact, this is an interesting exercise!

Exercise 3.19: Write an 8×8 multiplication program using the same algorithm, but shifting the result one position to the right instead of shifting the multiplicand by one position to the left. Compare it to the previous program, and determine whether this different approach would be faster or slower than the preceding one. The speeds of the Z80 instructions are given in the next chapter.

Improved Multiplication Program

The program that we have just developed is a straightforward translation of the algorithm to code. However, *effective programming requires close attention to detail*, and the length of the program can often be reduced or its execution speed can be improved. We are now going to study alternatives designed to improve this basic program.

Step 1

A first possible improvement lies in the better utilization of the Z80 instruction set. The second-to-last instruction as well as the preceding one can be replaced by a single instruction:

DJNZ LOOP

This is a special Z80 “automated jump” which decrements the B register and branches to a specified location if it is not “0”. To be absolutely correct, the instruction is not completely identical to the previous pair

```
DEC B  
JP NZ, MULT
```

for it specifies a *displacement*, and one can only jump within the range of - 126 to + 129. However, we must here jump to a location which is only a few bytes away, and this improvement is legitimate. The resulting program is shown in Figure 3.24 below:

MPY88B	LD	DE, (MPDAD)	
	LD	BC, (MPRAD)	
	LD	B, 8	BIT COUNTER
	LD	HL, 0	
MULT	SRL	C	
	JR	NC, NOADD	
	ADD	HL, DE	
NOADD	SLA	E	
	RL	D	
	DJNZ	MULT	
	LD	(RESAD), HL	
	RET		

Fig. 3.24: Improved Multiply, Step 1

Step 2

In order to improve this multiplication program further, we will observe that three different shift operations are used in the initial program of Figure 3.13. The multiplier is shifted right, then the multiplicand MPD is shifted left, in two operations, by first shifting register E left, then rotating register D to the left. This is time-consuming. A standard programming “trick” used in the case of multiplication is based on the following observation: every time that the multiplier is shifted by one bit position, another bit position becomes available in the multiplier register. For example, assuming that the multiplier shifts right (in the previous example), a bit position becomes available on the left. Simultaneously, it can be observed that the first partial product (or “result”) will use, at most, 9 bits. If a single register had been allocated to the result in the beginning of the program, we could then use the bit position that has been vacated by the multiplier to store the ninth bit of the result.

After the next shift of the MPR, the size of the partial product will be increased by just one bit again. In other words, a single register can be reserved initially for the partial product, and the bit positions which are being freed by the multiplier can then be used as the MPR is being shifted. In order to improve the program, we are therefore going to

PROGRAMMING THE Z80

assign MPR and RES to a register pair. Ideally, they should be shifted together in a single operation. Unfortunately, the Z80 shifts only 8-bit registers at a time. Like most other 8-bit microprocessors, it has no instruction that allows shifting 16 bits at a time.

However, another trick can be used. The Z80 (like the 8080) is equipped with special 16-bit add instructions that we have already used. Provided that the multiplier and the result are stored in the register pair H and L, we can use the instruction:

ADD HL, HL

which adds the contents of H and L to itself. Adding a number to itself is doubling it. Doubling a number in the binary system is equivalent to a left shift. We have just obtained a 16-bit shift in a single instruction. Unfortunately, the shift occurs to the left when we would like it to occur to the right. This is not a problem.

Conceptually, the MPR can be shifted either left or right. We have used a right shift algorithm because this is the one which is used in ordinary addition. However, it does not necessarily need to be so. The addition operation is commutative, and the order can be reversed: shifting the MPR to the left is just as valid.

In order to take advantage of this simulated 16-bit shift, we will have to shift the MPR to the left. Therefore, the MPR will reside in register H and the result in register L. The resulting register configuration is shown in Figure 3.25.

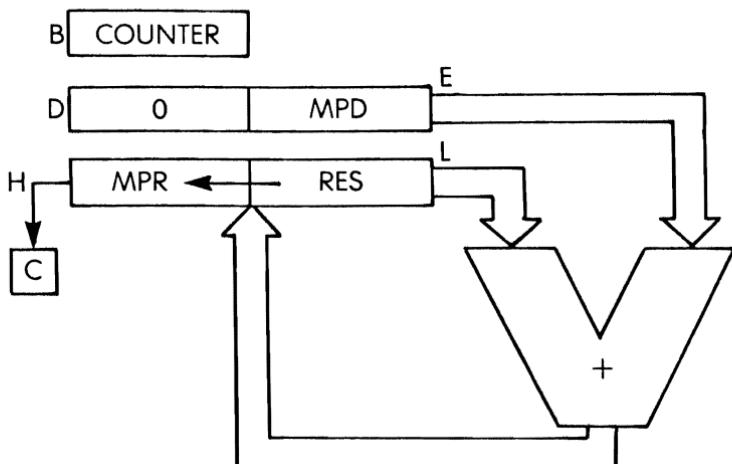


Fig. 3.25: Registers for Improved Multiply

The rest of the program is essentially identical to the previous one. The resulting program appears below:

MUL88C	LD	HL, (MPRAD-1)	
	LD	L, 0	
	LD	DE, (MPDAD)	
	LD	D, 0	
	LD	B, 8	COUNTER
MULT	ADD	HL, HL	SHIFT LEFT
	JR	NC, NOADD	
	ADD	HL, DE	
NOADD	DJNZ	MULT	
	LD	(RESAD), HL	
	RET		

Fig. 3.26: Improved Multiply, Step 2

When comparing this program to the previous one, it can be seen that the length of the multiplication loop (the number of instructions between MULT and the jump) has been reduced. This program has been written in fewer instructions and this will usually result in faster execution. This shows the advantage of selecting the correct registers to contain the information.

A straightforward design will generally result in a program that works. It will not result in a program that is *optimized*. It is therefore important to understand and use the available registers and instructions in the best possible way. These examples illustrate a rational approach to register selection and instruction selection for maximum efficiency.

Exercise 3.20: Compute the speed of a multiplication operation using this last program. Assume that a branch will occur in 50% of the cases. Look up the number of cycles required by every instruction in the index section. Assume a clock rate of 2 MHz (one cycle = 0.5 us).

Exercise 3.21: Note that here we have used the register pair D and E to contain the multiplicand. How would the above program be changed if we had used the register pair B and C instead? (Hint: this would require a modification at the end.)

Exercise 3.22: Why did we have to bother zeroing register D when loading MPD into E?

Finally, let us address a detail which may look irritating to the programmer who is not yet familiar with the Z80. The reader will have

noticed that, in order to load MPD into E from the memory, we had to load both registers D and E at the same time from a memory address. This is because, unless the address is contained in registers H and L, there is no way to fetch a single byte directly and load it into register E. This is a feature carried over from the early 8008, which had no direct addressing mode. The feature was carried forward into the 8080, with some improvements, and improved still further in the Z80, where it is possible to fetch 16 bits directly from a given memory address (but not 8 bits - except toward register A).

Now, having solved this possible mystery, let us execute a more complex multiplication.

A 16 X 16 Multiplication

In order to put our newly acquired skills to a test, we will multiply two 16-bit numbers. However, we will assume that the result requires only 16 bits, so that it can be contained in one of the register pairs.

The result, as in our first multiplication example, is contained in registers H and L (see Figure 3.27). The multiplicand MPD is contained in registers D and E.

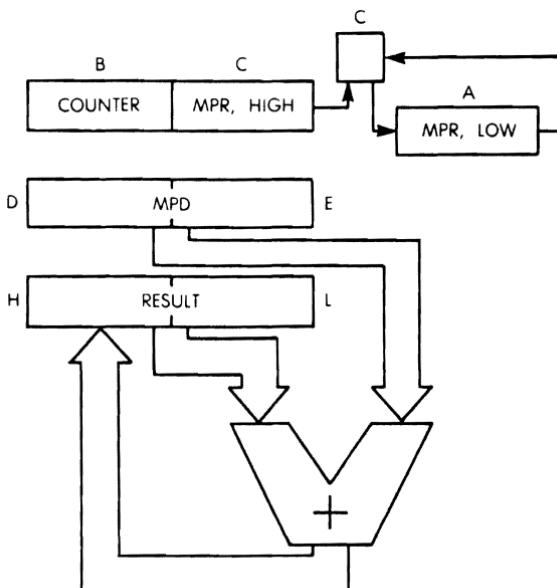


Fig. 3.27: 16 X 16 Multiply—The Registers

It would be tempting to deposit a multiplier into register B and C. However, if we want to take advantage of the DJNZ instruction, register B must be allocated to the counter. As a result, half of the multiplier will be in register C, and the other half in register A (see Figure 3.27). The multiplication program appears below:

MUL16	LD	A, (MPRAD + 1)	MPR, HIGH
	LD	C, A	
	LD	A, (MPRAD)	MPR, LOW
	LD	B, 16	COUNTER
	LD	DE, (MPDAD)	MPD
	LD	HL, 0	
MULT	SRL	C	RIGHT SHIFT MPR, HIGH
	RRA		ROTATE RIGHT MPR, LOW
	JR	NC, NOADD	TEST CARRY
	ADD	HL, DE	ADD MPD TO RESULT
NOADD	EX	DE, HL	
	ADD	HL, HL	DOUBLE – SHIFT MPD LEFT
	EX	DE, HL	
	DJNZ	MULT	
	RET		

Fig. 3.28: 16 X 16 Multiplication Program

The program is analogous to those we have developed before. The first six instructions (from label MUL16 to label MULT) perform the initialization of registers with the appropriate contents. One complication is introduced here by the fact that the two halves of MPR must be loaded in separate operations. It is assumed that MPRAD points to the low part of the MPR in the memory, followed in the next sequential memory location by the high part. (Note that the reverse convention can be used.) Once the high part of MPR has been read into A, it must be transferred into C:

```
LD    A, (MPRAD + 1)
LD    C, A
```

Finally, the low part of MPR can be read directly into the accumulator:

```
LD    A, (MPRAD)
```

PROGRAMMING THE Z80

The rest of the registers, B, D, E, H, and L are initialized as usual:

```
LD    B, 16  
LD    DE, (MPDAD)  
LD    HL, 0
```

A 16-bit shift must be performed on the multiplier. It requires two separate shift or rotate operations on registers C and A:

```
MULT  SRL   C  
      RRA
```

After the 16-bit shift, the right-most bit of the MPR, i.e., the LSB, is contained in the carry bit C where it can be tested:

```
JR    NC, NOADD
```

As usual, the multiplicand is not added to the result if the carry bit is “0”, and is added to the result if the carry bit is “1”:

```
ADD  HL, DE
```

Next, the multiplicand MPD must be shifted by one position to the left.

However, the Z80 does not have an instruction which will shift the contents of register D and E simultaneously to the left by one bit position, and it can also not add the contents of D and E to itself. The contents of D and E will therefore first be transferred into H and L, then doubled, and transferred back to D and E. This is accomplished by the next three instructions:

```
NOADD EX    DE, HL  
ADD   HL, HL  
EX    DE, HL
```

Finally, the counter B is decremented and a jump occurs to the beginning of the loop as long as it does not decrement to “0”:

```
DJNZ  MULT
```

As usual, it is possible to consider other register allocations which may (or may not) result in shorter codes:

Exercise 3.23: Load the multiplier into registers B and C. Place the counter in A. Write the corresponding multiplication program and discuss the advantages or disadvantages of this register allocation.

Exercise 3.24: Referring to the original 16-bit multiplication program of Figure 3.28, can you propose a way to shift the MPD, contained in registers D and E, without transferring it into registers H and L?

Exercise 3.25: Write a 16-by-16 multiplication program which detects the fact that the result has more than 16 bits. This is a simple improvement of our basic program.

Exercise 3.26: Write a 16-by-16 multiplication program with a 32-bit result. The suggested register allocation appears in Figure 3.29. Remember that the initial result after the first addition in the loop will require only 16 bits, and that the multiplier will free one bit for each subsequent iteration.

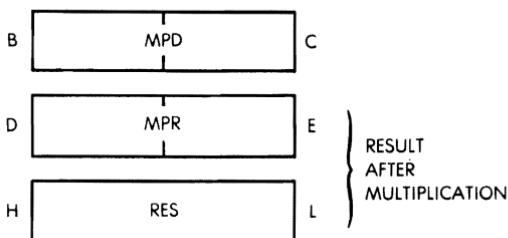


Fig. 3.29: 16×16 Multiply with 32-Bit Result

Let us now examine the last usual arithmetic operation, the division.

BINARY DIVISION

The algorithm for binary division is analogous to the one which has been used for the multiplication. The divisor is successively subtracted from the high order bits of the dividend. After each subtraction, the result is used instead of the initial dividend. The value of the quotient is simultaneously increased by 1 every time. Eventually, the result of the subtraction is negative. This is called an *overdraw*. One must then restore the partial result by adding the divisor back to it. Naturally, the quotient must be simultaneously decremented by 1. Quotient and dividend are then shifted by one bit position to the left and the algorithm is repeated. The flow-chart is shown in Figure 3.30.

The method just described is called the *restoring method*. A variation of this method which yields an improved speed of execution is called the *non-restoring method*.

PROGRAMMING THE Z80

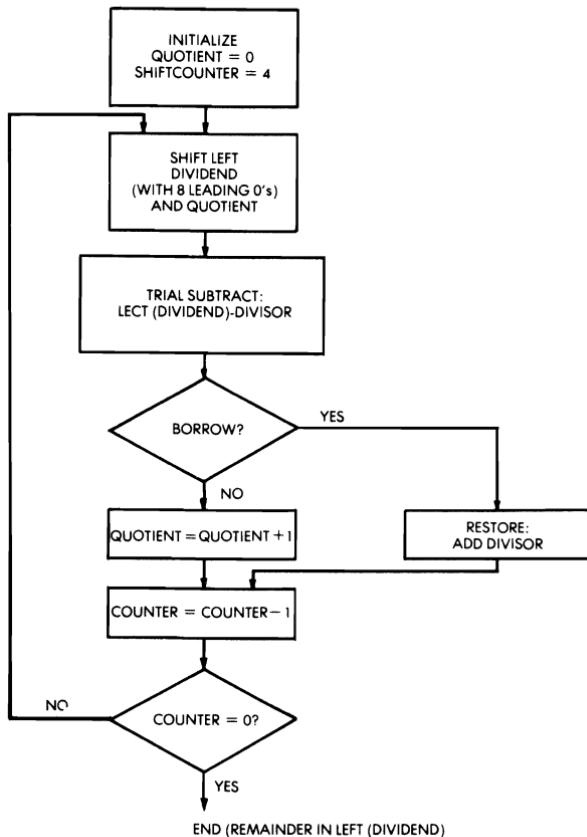


Fig. 3.30: 8-Bit Binary Division Flowchart

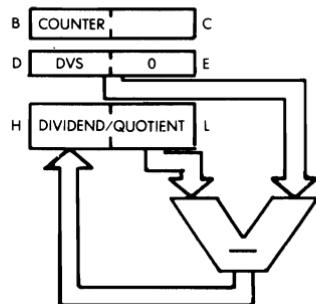


Fig. 3.31: 16/8 Division—The Registers

16-by-8 Division

As an example, let us here examine a 16-by-8 division, which will yield an 8-bit quotient and an 8-bit remainder dividend. The register allocation is shown in Figure 3.31.

The program appears below:

DIV168	LD	A, (DVSAD)	LOAD DIVISOR
	LD	D, A	INTO D
	LD	E, 0	
	LD	HL, (DVDAD)	LOAD 16-BIT DIVIDEND
	LD	B, 8	INITIALIZE COUNTER
DIV	XOR	A	CLEAR C BIT
	SBC	HL, DE	DIVIDEND - DIVISOR
	INC	HL	QUOTIENT = QUOTIENT + 1
	JP	P, NOADD	TEST IF REMAINDER POSITIVE
	ADD	HL, DE	RESTORE IF NECESSARY
	DEC	HL	QUOTIENT = QUOTIENT - 1
NOADD	ADD	HL, HL	SHIFT DIVIDEND LEFT
	DJNZ	DIV	LOOP UNTIL B = 0
		RET	

Fig. 3.32: 16/8 Division Program

The first five instructions in the program load the divisor and the dividend respectively into the appropriate registers. They also initialize the counter, in register B, to the value 8. Note again that register B is a preferred location for a counter if the specialized Z80 instruction DJNZ is to be used:

DIV168	LD	A, (DVSAD)
	LD	D, A
	LD	E, 0
	LD	HL, (DVDAD)
	LD	B, 8

Next, the divisor is subtracted from the dividend. Since an SBC instruction must be used (there is no 16-bit subtract without carry), the carry must be set to the value "0" before subtracting. This can be accomplished in a number of ways. The carry can be cleared by perform-

ing instructions such as:

```
XOR A  
AND A  
OR A
```

Here, an XOR is used:

```
DIV      XOR   A
```

The subtraction can then be performed:

```
SBC      HL, DE
```

It is anticipated that the subtraction will be successful, i.e., that the remainder will be positive. This is called the “trial subtract” step (refer to the flowchart of Figure 3.30). The quotient is therefore incremented by one. If the subtraction has in fact failed (i.e., if the remainder is negative), the quotient will have to be decremented by one later on:

```
INC      HL
```

The result of the subtraction is then tested:

```
JP      P, NOADD
```

If the remainder is positive or zero, the subtraction has been successful, and it is not necessary to store it. The program jumps to address NOADD. Otherwise, the current dividend must be restored to its previous value, by adding the divisor back to it, and the quotient must be decremented by one. This is performed by the next instructions:

```
ADD      HL, DE  
DEC      HL
```

Finally, the resulting dividend is shifted left, in anticipation of the next trial subtract operation. Finally, the B counter is decremented and tested for the value “0”. As long as B is not zero, this loop is executed:

```
NOADD  ADD  HL, HL  
DJNZ  DIV  
RET
```

Exercise 3.27: Verify the operation of this division program by hand, by filling out the table of Figure 3.33, as in Exercise 3.18 for the multiplication. Note that the contents of D need not be entered on the form of Figure 3.33, since they are never modified.

LABEL	INSTRUCTION	B	H	I

Fig. 3.33: Form for Division Program**8-Bit Division**

The following program uses a restoring method, and leaves a complemented quotient in A. It divides 8 bits by 8 bits (unsigned).

E IS DIVIDEND

C IS DIVISOR

A IS QUOTIENT

B IS REMAINDER

DIV88	XOR	A	CLEAR ACCUMULATOR
	LD	B, 8	LOOP COUNTER
LOOP88	RL	E	ROTATE CY INTO ACC-DIVIDEND
	RLA		CY WILL BE OFF
	SUB	C	TRIAL SUBTRACT DIVISOR
	JR	NC, \$ + 3	SUBTRACT OK
	ADD	A, C	RESTORE ACCUM, SET CY
	DJNZ	LOOP88	
	LD	B, A	PUT REMAINDER IN B
	LD	A, E	GET QUOTIENT
	RLA		SHIFT IN LAST RESULT BIT
	CPL		COMPLEMENT BITS
	RET		

Note: the “\$” symbol in the sixth instruction represents the value of the program counter.

Non Restoring Division

The following program performs a 16-bit by 15-bit integer division, using a non-restoring technique. IX points to the dividend, IY to the divisor (not zero). (see Figure 3.34.).

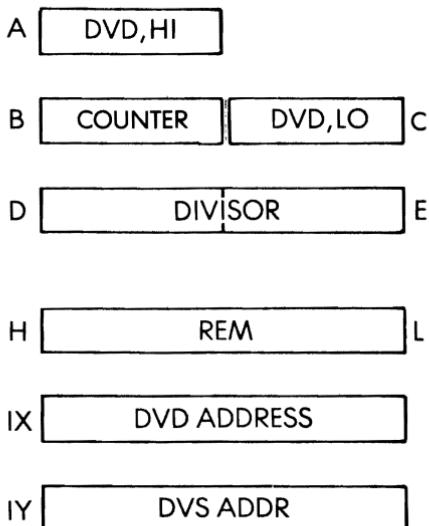


Fig. 3.34: Non-Restoring Division—The Registers

Register B is used as a counter, initially set to 16.

A and C contain the dividend.

D and E contain the divisor.

H and L contain the result.

The 16-bit dividend is shifted left by:

RL C

RLA

The remainder is shifted left by:

ADC HL, HL

The final quotient is left in B, C, with the remainder in HL. The program follows.

DIV16	LD	B, (IX + 1)	
	LD	C, (IX)	
	LD	D,(IY + 1)	
	LD	E, (IY)	
	LD	A, D	
	OR	E	(DIVISOR) HIGH OR (DIVISOR) LOW
	JR	Z, ERROR	CHECK FOR DIVISOR = ZERO
	LD	A, B	GET (DVD) HI
	LD	HL, 0	CLEAR RESULT
	LD	B, 16	COUNTER
TRIALSB	RL	C	ROTATE RESULT + ACC LEFT
	RLA		
	ADC	HL, HL	LEFT SHIFT. NEVER SETS CARRY.
	SBC	HL, DE	MINUS DIVISOR
NULL	CCF		RESULT BIT
	JR	NC, NGV	ACCUMULATOR NEGATIVE?
PTV	DJNZ	TRIALSB	COUNTER ZERO?
	JP	DONE	
RESTOR	RL	C	ROTATE RESULT + ACC LEFT
	RLA		
	ADC	HL, HL	AS ABOVE
	AND	A	
	ADC	HL, DE	RESTORE BY ADDING DVSR
	JR	C, PTV	RESULT POSITIVE
	JR	Z, NULL	RESULT ZERO
NGV	DJNZ	RESTOR	COUNTER ZERO?
DONE	RL	C	SHIFT IN RESULT BIT
	RLA		
	ADD	HL, DE	CORRECT REMAINDER
	LD	B, A	QUOTIENT IS IN B, C
	RET		

PROGRAMMING THE Z80

Exercise 3.28: Compare the previous program to the following one, using a restoring technique:

DIV16	LD	HL, 0	CLEAR ACCUMULATOR
	LD	B, 16	SET COUNTER
LOOP16	RL	C	ROT ACC-RESULT LEFT
	RLA		
	ADC	HL, HL	LEFT SHIFT
	SBC	HL, DE	TRIAL SUBTRACT DIVISOR
	JR	NC, \$ + 3	SUB WAS OK
	ADD	HL, DE	RESTORE ACCUM
	CCF		CALC RESULT BIT
	DJNZ	LOOP16	COUNTER NOT ZERO
	RL	C	SHIFT IN LAST RESULT BIT
	RLA		
	RET		

Note: The symbol “\$” means “current location” (eighth instruction).

LOGICAL OPERATIONS

The other class of instructions which can be executed by the ALU inside the microprocessor is the set of *logical instructions*. They include: AND, OR and exclusive OR (XOR). In addition, one can also include here the shift and rotate operations which have already been utilized, and the comparison instruction, called CP for the Z80. The individual use of AND, OR, XOR, will be described in Chapter 4 on the instruction set.

Let us now develop a brief program which will check whether a given memory location called LOC contains the value “0”, the value “1”, or something else.

The program will introduce the comparison instruction, and perform a series of logical tests. Depending on the result of the comparison, one program segment or another will be executed.

The program appears below:

	LD	A, (LOC)	READ CHARACTER IN LOC
	CP	00H	COMPARE TO ZERO
	JP	Z, ZERO	IS IT A 0?
	CP	01H	COMPARE TO ONE
	JP	Z, ONE	
NONEFOUND	...		
		...	
ZERO	...		
		...	
ONE	...		

The first instruction: “LD A, (LOC)” reads the contents of memory location LOC, and loads it into the accumulator. This is the character we want to test. It is compared to the value 0 by the following instruction:

CP 00H

This instruction compares the contents of the accumulator to the hexadeciml value “00”, i.e., the bit pattern “0000 0000”. This comparison instruction will set the Z bit in the flags register to the value “1”, if it succeeds. This bit can then be tested by the next instruction:

JP Z, ZERO

The jump instruction tests the value of the Z bit. If the comparison succeeds, the Z bit has been set to one, and the jump will succeed. The program will then jump to the address ZERO. If the test fails, then the next sequential instruction will be executed:

CP 01H

Similarly, the following jump instruction will branch to location ONE if the comparison succeeds. If none of the comparisons succeed, then the instruction at location NONEFOUND will be executed.

JP Z, ONE

NONEFOUND ...

PROGRAMMING THE Z80

This program was introduced to demonstrate the value of the comparison instruction followed by a jump. This combination will be used in many of the following programs.

Exercise 3.29: Refer to the definition of the LD A, (LOC) instruction in the next chapter. Examine the effect of this instruction on the flags, if any. Is the second instruction of this program necessary (CP 00H)?

Exercise 3.30: Write the program which will read the contents of memory location "24" and branch to an address called "STAR" if there was a "*" in memory location 24. The bit pattern for a "*" in binary notation will be assumed to be represented by "00101010".

INSTRUCTION SUMMARY

We have now studied most of the important instructions of the Z80 by using them. We have transferred values between the memory and the registers. We have performed arithmetic and logical operations on such data. We have tested it, and depending on the results of these tests, have executed various portions of the program. In particular, special "automated" Z80 instructions such as DJNZ have been used to shorten programs. Other automated instructions: LDDR, CPIR, INIR will be introduced throughout the remainder of this book.

Full use has been made of special Z80 features, such as 16-bit register instructions to simplify the programs, and the reader should be careful not to use these programs on an 8080: they have been optimized for the Z80.

We have also introduced a structure called a loop. Another important programming structure will be introduced now: the subroutine.

SUBROUTINES

In concept, a subroutine is simply a block of instructions which has been given a name by the programmer. From a practical standpoint, a subroutine must start with a special instruction called a *subroutine declaration*, which identifies it as such for the assembler. It is also terminated by another special instruction called a *return*. Let us first illustrate the use of a subroutine in a program in order to demonstrate its value. Then, we will examine how it is actually implemented.

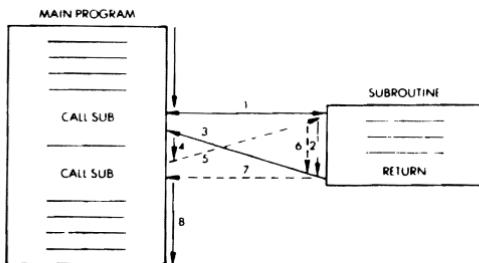


Fig. 3.35: Subroutine Calls

The use of a subroutine is illustrated in Figure 3.35. The main program appears on the left of the illustration. The subroutine is shown symbolically on the right. Let us examine the subroutine mechanism. The lines of the main program are executed successively until a new instruction "CALL SUB" is met. This special instruction is the *subroutine call* and results in a transfer to the subroutine. This means that the next instruction to be executed after the CALL SUB is the first instruction within the subroutine. This is illustrated by arrow 1 on the illustration.

Then, the subprogram within the subroutine executes just like any other program. We will assume that the subroutine does not contain any other calls. The last instruction of this subroutine is a RETURN. This is a special instruction which will cause a return to the main program. The next instruction to be executed after the RETURN is the one following the CALL SUB in the main program. This is illustrated by arrow 3 on the illustration. Program execution continues then, as illustrated by arrow 4.

In the body of the main program a second CALL SUB appears. A new transfer occurs, shown by arrow 5. This means that the body of the subroutine is again executed following the CALL SUB instruction.

Whenever the RETURN within the subroutine is encountered, a return occurs to the instruction following the CALL SUB in question. This is illustrated by arrow 7. Following the return to the main program, program execution proceeds normally, as illustrated by arrow 8.

The effect of the two special instructions CALL SUB and RETURN should now be clear. What is the value of the subroutine mechanism?

The essential value of the subroutine is that it can be called from any number of points in the main program, and used repeatedly *without*

rewriting it. A first advantage is that this approach saves memory space, since there is no need to rewrite the subroutine every time. A second advantage is that the programmer can design a specific subroutine only once and then use it repeatedly. This is a significant simplification in program design.

Exercise 3.31: What is the main disadvantage of a subroutine? (Answer follows.)

The disadvantage of the subroutine should be clear just by examining the flow of execution between the main program and the subroutine. A subroutine results in a *slower execution*, since extra instructions must be executed: the CALL SUB and the RETURN.

Implementation of the Subroutine Mechanism

We will examine here how the two special instructions, CALL SUB and RETURN, are implemented internally within the processor. The effect of the CALL SUB instruction is to cause the next instruction to be fetched at a new address. You will remember (or else read Chapter 1 again) that the address of the next instruction to be executed in a computer is contained in the program counter (PC). This means that the effect of the CALL SUB is to substitute new contents in register PC. Its effect is to load the start address of the subroutine in the program counter. *Is that really sufficient?*

To answer this question, let us consider the other instruction which has to be implemented: the RETURN. The RETURN must cause, as its name indicates, a return to the instruction that follows the CALL SUB. This is possible only if the address of this instruction has been preserved somewhere. This address happens to be the value of the program counter at the time that the CALL SUB was encountered. This is because the program counter is automatically incremented every time it is used (read Chapter 1 again). This is precisely the address that we want to preserve, so that we can later perform the RETURN.

The next problem is: where can we save this return address? This address must be saved in a location where it is guaranteed that it will not be erased.

However, let us now consider the following situation, illustrated by Figure 3.36. In this example, subroutine 1 contains a call to SUB2. Our mechanism should work in this case as well. Naturally, there might even be more than two subroutines, say N “nested” calls. Whenever a new

CALL is encountered, the mechanism must therefore again store the program counter. This implies that we need at least $2N$ memory locations for this mechanism. Additionally, we will need to return from SUB2 first and SUB1 next. In other words, we need a structure which can preserve the chronological ordering in which addresses have been saved.

The structure has a name and has already been introduced. It is *the stack*. Figure 3.38 shows the actual contents of the stack during successive subroutine calls. Let us look at the main program first. At address 100, the first call is encountered: CALL SUB1. We will assume that, in this microprocessor, the subroutine call uses 3 bytes (RST is an exception). The next sequential address is therefore not "101", but "103". The CALL instruction uses addresses "100", "101", "102". Because the control unit of the Z80 "knows" that it is a 3-byte instruction, the value of the program counter, when the call has been completely decoded, will be "103". The effect of the call will be to load the value "280" in the program counter. "280" is the starting address of SUB1.

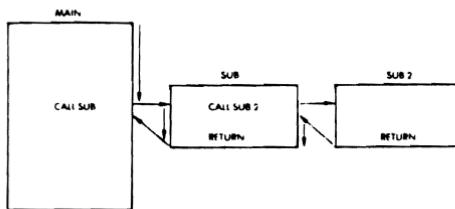


Fig. 3.36: Nested Calls

We are now ready to demonstrate the effect of the RETURN instruction and the correct operation of our stack mechanism. Execution proceeds within SUB2 until the RETURN instruction is encountered at time 3. The effect of the RETURN instruction is simply to pop the top of the stack into the program counter. In other words, the program counter is restored to its value prior to the entry into the subroutine. The top of the stack in our example is "303". Figure 3.38 shows that, at time 3, value "303" has been removed from the stack and has been put back into the program counter. As a result, instruction execution proceeds from address "303". At time 4, the RETURN of SUB1 is encountered. The value on top of the stack is "103". It is popped and is installed in the program counter. As a result, program execution will proceed from location "103" on within the main program. This is, indeed,

PROGRAMMING THE Z80

the effect that we wanted. Figure 3.38 shows that at time 4 the stack is again empty. The mechanism works.

The subroutine call mechanism works up to the maximum dimension of the stack. This is why early microprocessors which had a 4- or 8-register stack were essentially limited to 4 or 8 levels of subroutine calls.

Note that, on Figures 3.36 and 3.37, the subroutines have been shown to the right of the main program. This is only for the clarity of the diagram. In reality, the subroutines are typed by the user as regular instructions of the program. On a sheet of paper, when producing the listing of the complete program, the subroutines may be at the beginning of the text, in its middle, or at the end. This is why they are preceded by a subroutine declaration: they must be identified. The special instructions tell the assembler that what follows should be treated as a subroutine. Such assembler *directives* will be discussed in Chapter 10.

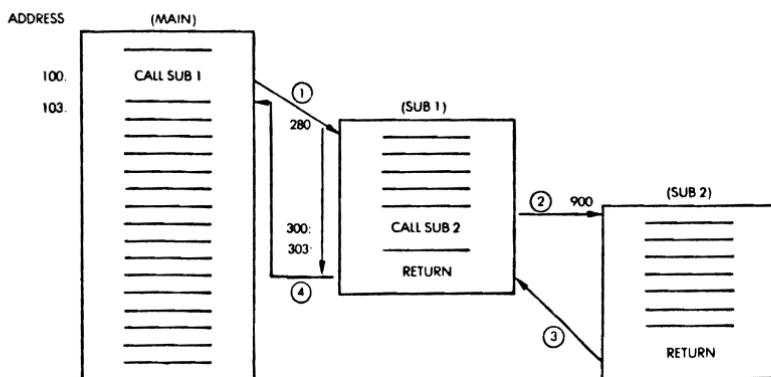


Fig. 3.37: The Subroutine Calls

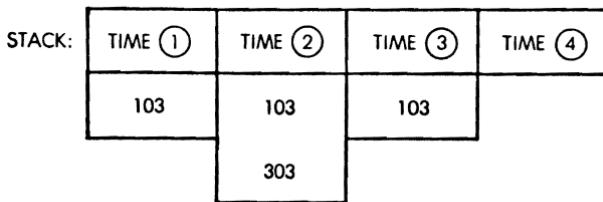


Fig. 3.38: Stack vs. Time

Z80 Subroutines

The basic concepts relating to subroutines have now been presented. It has been shown that the stack is required in order to implement this mechanism. The Z80 is equipped with a 16-bit stack-pointer register. The stack can therefore reside anywhere within the memory and may have up to 64K (1K = 1024) bytes, assuming they are available for that purpose. In practice, the start address for the stack, as well as its maximum dimension, will be defined by the programmer before writing his program. A memory area will then be reserved for the stack.

The subroutine-call instruction, in the case of the Z80, is called CALL, and comes in two versions; the direct or unconditional call, such as CALL ADDRESS, is the one we have already described. In addition, the Z80 is equipped with a conditional call instruction which will call a subroutine if a condition is met. For example: CALL NZ, SUB1 will result in a call to subroutine 1 if the Z flag is zero at the time of the test. This is a powerful facility, since many subroutine calls are conditional, i.e., occur only if some specific condition is met.

CALL CC, NN is executed only if the condition specified by "CC" is true. CC is a set of three bits (bits 3, 4, and 5 of the opcode) which may specify up to eight conditions. They correspond respectively to the four flags "Z", "C", "P/V", "S" being either zero or non-zero.

Similarly, two types of return instructions are provided: RET and RET CC.

RET is the basic return instruction. It occupies one byte, and causes the top two bytes of the stack to be re-installed in the program counter. It is unconditional.

RET CC has the same effect except that it is executed only if the conditions specified by CC are true. The condition bits are the same as for the CALL instruction just described.

Additionally, two specialized types of return are available which are used to terminate interrupt routines: RETI, RETN. They are described in the section on the Z80 instructions as well as in the section on interrupts.

Finally, one more specialized instruction is provided which is analogous to a subroutine call, but allows the program to branch to only one of eight starting locations located in page zero. This is the RST P instruction. This is a one-byte instruction which automatically preserves the program counter in the stack, and causes a branch to the address specified by the three-bit P field. The P field corresponds to bits 3, 4 and 5 of the instruction, multiplied by eight.

In other words, if bits 3, 4, 5 are “000”, the jump will occur to location 00H. If these bits are “001”, the branch will occur to 08H, etc. up to 111, which will cause a branch to location 38H. The RST instruction is very efficient in terms of speed since it is a single-byte instruction. However, it can jump to only eight locations, in page 0. Additionally, these addresses in page 0 are only eight bytes apart. This instruction is a carry-over from the 8080 and was extensively used for interrupts. This will be described in the interrupt section. However, this instruction may be used for any other purpose by the programmer, and should be considered as a possible specialized subroutine call.

Subroutine Examples

Most of the programs that we have developed and are going to develop would usually be written as subroutines. For example, the multiplication program is likely to be used by many areas of the program. In order to facilitate and clarify program development, it is therefore convenient to define a subroutine whose name would be, for example, MULT. At the end of this subroutine we would simply add the instruction RET.

Exercise 3.32: If MULT is used as a subroutine, would it “damage” any internal flags or registers?

Recursion

Recursion is a word used to indicate that a subroutine is calling itself. If you have understood the implementation mechanism, you should now be able to answer the following question:

Exercise 3.33: Is it legal to let a subroutine call itself? (In other words, will everything work even if a subroutine calls itself?) If you are not sure, draw the stack and fill it with the successive addresses. Then, look at the registers and memory (see Exercise 3.18) and determine if a problem exists.

Interrupts will be discussed in the input/output chapter (Chapter 6). All returns except returns from interrupts are one-byte instructions; all calls are 3-byte instructions (except RST).

Exercise 3.34: Look at the execution times of the CALL and the RET instructions in the next chapter. Why is the return from a subroutine so much faster than the CALL? (Hint: if the answer is not obvious, look again at the stack implementation of the subroutine mechanism, and analyze the internal operations that must be performed.)

Subroutine Parameters

When calling a subroutine, one normally expects the subroutine to work on some data. For example, in the case of multiplication, one wants to transmit two numbers to the subroutine which will perform the multiplication. We saw in the case of the multiplication routine that this subroutine expected to find the multiplier and the multiplicand in given memory locations. This illustrates one method of passing parameters: through memory. Two other techniques are used, so that we have three ways of passing parameters.

- 1—through registers
- 2—through memory
- 3—through the stack

Registers can be used to pass parameters. This is an advantageous solution, provided that registers are available, since one does not need to use a fixed memory location: the subroutine remains memory-independent. If a fixed memory location is used, any other user of the subroutine must be very careful that he uses the same convention and that the memory location is indeed available (look at Exercise 3.19 above). This is why, in many cases, a block of memory locations is reserved simply to pass parameters among various subroutines.

Using memory has the advantage of greater flexibility (more data), but results in poorer performance and also in tying the subroutine to a given memory area.

Depositing parameters in *the stack* has the same advantage as using registers: it is memory-independent. The subroutine simply knows that it is supposed to receive, say, two parameters which are stored on top of the stack. Naturally, it has disadvantages: it clutters the stack with data and, therefore, reduces the number of possible levels of subroutine calls. It also significantly complicates the use of the stack, and may require multiple stacks.

The choice is up to the programmer. In general, one wishes to remain independent from actual memory locations as long as possible.

If registers are not available, a possible solution is the stack. However, if a large quantity of information should be passed to a subroutine, this information may have to reside directly in the memory. An elegant way around the problem of passing a block of data is simply to transmit a pointer to the information. A *pointer* is the address of the beginning of the block. A pointer can be transmitted in a register, or in the stack (two-stack locations can be used to store a 16-bit address), or in a given memory location(s).

PROGRAMMING THE Z80

Finally, if neither of the two solutions is applicable, then an agreement may be made with the subroutine that the data will be at some fixed memory location (the “mail-box”).

Exercise 3.35: *Which of the three methods above is best for recursion?*

Subroutine Library

There is a strong advantage to structuring portions of a program into identifiable subroutines: they can be debugged independently and can have a mnemonic name. Provided that they will be used in other areas of the program, they become shareable, and one can thus build a library of useful subroutines. However, there is no general panacea in computer programming. Using subroutines systematically for any group of instructions that can be grouped by function may also result in poor efficiency. The alert programmer will have to weigh the advantages against the disadvantages.

SUMMARY

This chapter has presented the way information is manipulated inside the Z80 by instructions. Increasingly complex algorithms have been introduced and translated into programs. The main types of instructions have been used and explained.

Important structures such as loops, stacks and subroutines, have been defined.

You should now have acquired a basic understanding of programming, and of the major techniques used in standard applications. Let us study the instructions available.

BASIC PROGRAMMING TECHNIQUES

```

A=00 BC=0000 DE=0000 HL=0000 S=0300 P=0100 0100' LD BC,(0200)
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (0200')
A=00 BC=0003 DE=0000 HL=0000 S=0300 P=0104 0104' LD B,08
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
A=00 BC=0803 DE=0000 HL=0000 S=0300 P=0106 0106' LD DE,(0202)
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (0202')
A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010A 010A' LD B,00
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010C 010C' LD HL,0000
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (0000')
A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010F 010F' SRL C
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
C A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0111 0111' JR NC,0114
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (0114')
C A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0113 0113' ADD HL,DE
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
A=00 BC=0801 DE=0005 HL=0005 S=0300 P=0114 0114' SLA E
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
V A=00 BC=0801 DE=000A HL=0005 S=0300 P=0116 0116' RL D
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V A=00 BC=0801 DE=000A HL=0005 S=0300 P=0118 0118' DEC B
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
N A=00 BC=0701 DE=000A HL=0005 S=0300 P=0119 0119' JP NZ,010F
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (010F')
N A=00 BC=0701 DE=000A HL=0005 S=0300 P=010F 010F' SRL C
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V C A=00 BC=0700 DE=000A HL=0005 S=0300 P=0111 0111' JR NC,0114
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (0114')
Z V C A=00 BC=0700 DE=000A HL=0005 S=0300 P=0113 0113' ADD HL,DE
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V A=00 BC=0700 DE=000A HL=000F S=0300 P=0114 0114' SLA E
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
V A=00 BC=0700 DE=0014 HL=000F S=0300 P=0116 0116' RL D
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V A=00 BC=0700 DE=0014 HL=000F S=0300 P=0118 0118' DEC B
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
N A=00 BC=0600 DE=0014 HL=000F S=0300 P=0119 0119' JP NZ,010F
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (010F')
N A=00 BC=0600 DE=0014 HL=000F S=0300 P=010F 010F' SRL C
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V A=00 BC=0600 DE=0014 HL=000F S=0300 P=0111 0111' JR NC,0114
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (0114')
Z V A=00 BC=0600 DE=0014 HL=000F S=0300 P=0114 0114' SLA E
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
V A=00 BC=0600 DE=0028 HL=000F S=0300 P=0116 0116' RL D
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V A=00 BC=0600 DE=0028 HL=000F S=0300 P=0118 0118' DEC B
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 .
N A=00 BC=0500 DE=0028 HL=000F S=0300 P=0119 0119' JP NZ,010F
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (010F')
N A=00 BC=0500 DE=0028 HL=000F S=0300 P=010F 010F' SRL C
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V A=00 BC=0500 DE=0028 HL=000F S=0300 P=0111 0111' JR NC,0114
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (0114')
Z V A=00 BC=0500 DE=0028 HL=000F S=0300 P=0114 0114' SLA E
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
V A=00 BC=0500 DE=0050 HL=000F S=0300 P=0116 0116' RL D
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
Z V A=00 BC=0500 DE=0050 HL=000F S=0300 P=0118 0118' DEC B
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
N A=00 BC=0400 DE=0050 HL=000F S=0300 P=0119 0119' JP NZ,010F
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00 (010F')
N A=00 BC=0400 DE=0050 HL=000F S=0300 P=010F 010F' SRL C
A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00

```

Fig. 3.39: Multiplication: A Complete Trace

PROGRAMMING THE Z80

Z V	A=00 BC=0400 DE=0050	HL=000F S=0300 P=0111 0111' JR	NC,0114
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	(0114')
Z V	A=00 BC=0400 DE=0050	HL=000F S=0300 P=0114 0114' SLA	E
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
S V	A=00 BC=0400 DE=00A0	HL=000F S=0300 P=0116 0116' RL	D
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
Z V	A=00 BC=0400 DE=00A0	HL=000F S=0300 P=0118 0118' DEC	B
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
N	A=00 BC=0300 DE=00A0	HL=000F S=0300 P=0119 0119' JP	NZ,010F
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	(010F')
N	A=00 BC=0300 DE=00A0	HL=000F S=0300 P=010F 010F' SRL	C
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
Z V	A=00 BC=0300 DE=00A0	HL=000F S=0300 P=0111 0111' JR	NC,0114
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	(0114')
Z V	A=00 BC=0300 DE=00A0	HL=000F S=0300 P=0114 0114' SLA	E
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
C	A=00 BC=0300 DE=0040	HL=000F S=0300 P=0116 0116' RL	D
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
	A=00 BC=0300 DE=0140	HL=000F S=0300 P=0118 0118' DEC	B
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
N	A=00 BC=0200 DE=0140	HL=000F S=0300 P=0119 0119' JP	NZ,010F
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	(010F')
N	A=00 BC=0200 DE=0140	HL=000F S=0300 P=010F 010F' SRL	C
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
Z V	A=00 BC=0200 DE=0140	HL=000F S=0300 P=0111 0111' JR	NC,0114
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	(0114')
Z V	A=00 BC=0200 DE=0140	HL=000F S=0300 P=0114 0114' SLA	E
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
S	A=00 BC=0200 DE=0180	HL=000F S=0300 P=0116 0116' RL	D
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
	A=00 BC=0200 DE=0280	HL=000F S=0300 P=0118 0118' DEC	B
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
N	A=00 BC=0100 DE=0280	HL=000F S=0300 P=0119 0119' JP	NZ,010F
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	(010F')
N	A=00 BC=0100 DE=0280	HL=000F S=0300 P=010F 010F' SRL	C
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
Z V	A=00 BC=0100 DE=0280	HL=000F S=0300 P=0111 0111' JR	NC,0114
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	(0114')
Z V	A=00 BC=0100 DE=0280	HL=000F S=0300 P=0114 0114' SLA	E
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
Z V C	A=00 BC=0100 DE=0200	HL=000F S=0300 P=0116 0116' RL	D
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
V	A=00 BC=0100 DE=0500	HL=000F S=0300 P=0118 0118' DEC	B
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	
Z N	A=00 BC=0000 DE=0500	HL=000F S=0300 P=0119 0119' JP	NZ,010F
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	(010F')
Z N	A=00 BC=0000 DE=0500	HL=000F S=0300 P=011C 011C' LD	(0204),HL
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	(0204')
Z N	A=00 BC=0000 DE=0500	HL=000F S=0300 P=011F 011F' NOP	
	A'=00 B'=0000 D'=0000	H'=0000 X=0000 Y=0000 I=00	

Fig. 3.39: Multiplication: A Complete Trace (continued)

ANSWERS TO EXERCISE 3.18 (MULTIPLICATION):

```
CROMEMCO CDOS Z80 ASSEMBLER version 02.15 PAGE 0001
0000'      0001      ORG     0100H
             (0200)    0002  MPRAD  DL      0200H
             (0202)    0003  MPDAD  DL      0202H
             (0204)    0004  RESAD  DL      0204H
             0005  ;
0100 ED4B0002 0006 MP488 LD BC,(MPRAD) ;LOAD MULTIPLIER INTO C
0104 0608 0007 LD B,8   ;B IS BIT COUNTER
0106 ED5B0202 0008 LD DE,(MPDAD) ;LOAD MULTPLICAND INTO E
010A 1600 0009 LD D,0   ;CLEAR D
010C 210000 0010 LD HL,0  ;SET RESULT TO 0
010F CB39 0011 MULT  SRL C   ;SHIFT MULTIPLIER BT1 INTO CARRY
0111 3001 0012 JR NC,NOADD ;TEST CARRY
0113 19 0013 ADD HL,DE ;ADD MPD TO RESULT
0114 CB23 0014 NOADD SLA E   ;SHIFT MPD LEFT
0116 CB12 0015 RL D   ;SAVE BIT IN D
0118 05 0016 DEC B   ;DECREMENT SHIFT COUNTER
0119 C20F01 0017 JP NZ,MULT ;DO IT AGAIN IF COUNTER > 0
011C 220402 0018 LD (RESAD),HL ;STORE RESULT
011F (0000) 0019 END

Errors 0
```

Fig. 3.40: The Multiplication Program (Hex)

LABEL	INSTRUCTION	B	C	C (CARRY)	D	E	H	L
MP488	LD BC,(0200)	00	00	0	00	00	00	00
	LD B,08	08	03	0	00	00	00	00
	LD DE,(0202)	08	03	0	00	05	00	00
	LD D,00	08	03	0	00	05	00	00
	LD HL,0000	08	03	0	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	0	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
MULT	JP NZ,010F	07	01	0	00	0A	00	05
	SRL C	07	00	1	00	0A	00	05
	JR NC,0114	07	00	1	00	0A	00	05
NOADD	ADD HL,DE	07	00	0	00	0A	00	0F
	SLA E	07	00	0	00	14	00	0F
	RL D	07	00	0	00	14	00	0F
	DEC B	06	00	0	00	14	00	0F
	JP NZ,010F	06	00	0	00	14	00	0F

Fig. 3.41: Two Iterations Through the Loop

4

THE Z80 INSTRUCTION SET

INTRODUCTION

This chapter will first analyze the various classes of instructions which should be available in a general-purpose computer. It will then analyze one by one all of the instructions available for the Z80, and explain in detail their purpose and the manner in which they affect flags or can be used in conjunction with various addressing modes. A detailed discussion of addressing techniques will be presented in Chapter 5.

CLASSES OF INSTRUCTIONS

Instructions may be classified in many ways, and there is no standard. We will here distinguish five main categories of instructions:

- 1—data transfers
- 2—data processing
- 3—test and branch
- 4—input/output
- 5—control

Let us now examine each of these classes of instructions in turn.

Data Transfers

Data transfer instructions will transfer data between registers, or between a register and memory, or between a register and an input/output device. Specialized transfer instructions may exist for registers which play a specific role. For example, push and pop operations are provided for efficient stack operation. They will move a word of

data between the top of the stack and the accumulator in a single instruction, while automatically updating the stack-pointer register.

Data Processing

Data processing instructions fall into five general categories:

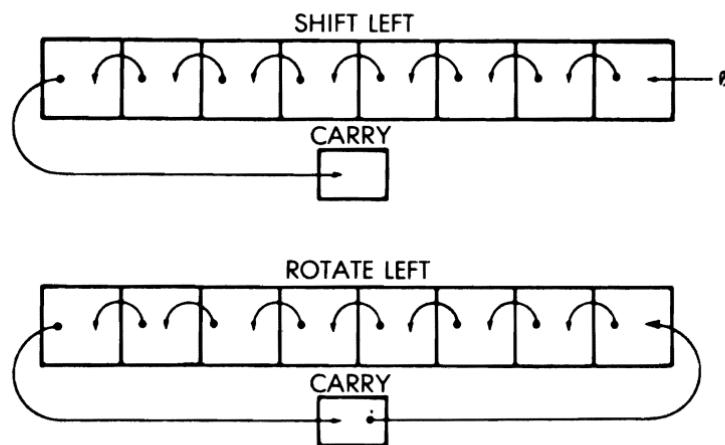
- 1—arithmetic operations (such as plus/minus)
- 2—bit manipulation (set and reset)
- 3—increment and decrement
- 4—logical operations (such as AND, OR, exclusive OR)
- 5—skew and shift operations (such as shift, rotate)

It should be noted that, for efficient data processing, it is desirable to have powerful arithmetic instructions, such as multiply and divide. Unfortunately, they are not available on most microprocessors. It is also desirable to have powerful shift and skew instructions, such as shift n bits, or a nibble exchange, where the right half and the left half of the byte are exchanged. These are also usually unavailable on most microprocessors.

Before examining the actual Z80 instructions, let us recall the difference between a *shift* and a *rotation*. The shift will move the contents of a register or a memory location by one bit location to the left or to the right. The bit falling out of the register will go into the carry bit. The bit coming in on the other side will be a “0” except in the case of an “arithmetic shift right,” where the MSB will be duplicated.

In the case of a rotation, the bit coming out still goes in the carry. However, the bit coming in is the previous value which was in the carry bit. This corresponds to a 9-bit rotation. It is often desirable to have a true 8-bit rotation where the bit coming in on one side is the one falling from the other side. This is not provided on most microprocessors but is available on the Z80 (see Figure 4.1).

Finally, when shifting a word to the right, it is convenient to have one more type of shift, called a sign extension or an “arithmetic shift right.” When doing operations on two’s complement numbers, particularly when implementing floating-point routines, it is often necessary to shift a negative number to the right. When shifting a two’s complement number to the right, the bit which must come in on the left side should be a “1” (the sign should get repeated as many times as needed by the successive shifts). This is the arithmetic shift right.

**Fig. 4.1: Shift and Rotate**

Test and Jump

The test instructions will test bits in the specified register for “0” or “1”, or combinations. At a minimum, it must be possible to test the flags register. It is, therefore, desirable to have as many flags as possible in this register. In addition, it is convenient to be able to test for combinations of such bits with a single instruction. Finally, it is desirable to be able to test *any bit position in any register*, and to test the value of a register compared to the value of any other register (greater than, less than, equal). Microprocessor test instructions are usually limited to testing single bits of the flags register. The Z80, however, offers better facilities than most.

The jump instructions that may be available generally fall into three categories:

- 1—the jump, which specifies a full 16-bit address
- 2—the relative jump, which often is restricted to an 8-bit displacement field
- 3—the call, which is used with subroutines

It is convenient to have two- or even three-way jumps, depending, for example, on whether the result of a comparison is "greater than," "less than," or "equal." It is also convenient to have skip operations, which will jump forward or backwards by a few instructions. However, a "skip" is equivalent to a "jump." Finally, in most loops, there is usually a decrement or increment operation at the end, followed by a test-and-branch. The availability of a single-instruction increment/decrement plus test-and-branch is, therefore, a significant advantage for efficient loop implementation. This is not available in most microprocessors. Only simple branches, combined with simple tests, are available. This, naturally, complicates programming and reduces efficiency. In the case of the Z80, a "decrement and jump" instruction is available. However, it only tests a specific register (B) for zero.

Input/Output

Input/output instructions are specialized instructions for the handling of input/output devices. In practice, a majority of the 8-bit microprocessors use *memory-mapped I/O*: input/output devices are connected to the address bus just like memory chips, and addressed as such. They appear to the programmer as memory locations. All memory-type operations normally require 3 bytes and are, therefore, slow. For efficient input/output handling in such an environment, it is desirable to have a short addressing mechanism available so that I/O devices whose handling speed is crucial may reside in page 0. However, if page 0 addressing is available, it is usually used for RAM memory, which prevents its effective use for input/output devices. The Z80, like the 8080, is equipped with specialized I/O instructions. As a result, in the case of the Z80, the designer may use either method: input/output devices may be addressed as memory devices, or else as input/output devices, using the I/O instructions.

They will be described later in this chapter.

Control Instructions

Control instructions supply synchronization signals and may suspend or interrupt a program. They can also function as a break or a simulated interrupt. (Interrupts will be described in Chapter 6 on Input/Output Techniques.)

THE Z80 INSTRUCTION SET

Introduction

The Z80 microprocessor was designed to be a replacement for the 8080, and to offer additional capabilities. As a result of this design philosophy, the Z80 offers all the instructions of the 8080, plus additional instructions. In view of the limited number of bits available in an 8-bit opcode, one may wonder how the designers of the Z80 succeeded in implementing many additional ones. They did so by using a few unused 8080 opcodes and by adding an additional byte to the opcode for indexed operations. This is why some of the Z80 instructions occupy up to five bytes in the memory.

It is important to remember that any program can be written in many different ways. A thorough knowledge and understanding of the instruction set is indispensable for achieving efficient programming. However, when learning how to program, it is not essential to write optimized programs. During a first reading of this chapter, it is therefore unimportant to remember all the various instructions. It is important to remember the categories of instructions and to study typical examples. Then, when writing programs, the reader should consult the Z80 instruction-set description, and select the instructions best suited to his needs. The various instructions of the Z80 will therefore be reviewed in this section with the intent of simplifying them and grouping them in logical categories. The reader interested in exploring the capabilities of the various instructions is referred to the individual descriptions of the instructions.

We will now examine the capabilities provided by the Z80 in terms of the five classes of instructions which have been defined at the beginning of this chapter.

Data Transfer Instructions

Data transfer instructions on the Z80 may be classified in four categories: 8-bit transfers, 16-bit transfers, stack operations, and block transfers. Let us examine them.

Eight-Bit Data Transfers

All eight-bit data transfers are accomplished by load instructions. The format is:

LD destination, source

For example, the accumulator A may be loaded from register B by using the instructions:

LD A,B

Direct transfers may be accomplished between any two of the working registers (ABCDEHL).

In order to load any of the working registers, except for the accumulator, from a memory location, the address of this memory location must first be loaded into the H-L register pair.

For example, in order to load register C from memory location 1234, register H and L will first have to be loaded with the value "1234". (A load instruction operating on 16 bits will be used. This is described in the following section.)

Then, the instruction LD C, (HL) will be used and will accomplish the desired result.

The accumulator is an exception. It can be loaded directly from any specified memory location. This is called the extended addressing mode. For example, in order to load the accumulator with the contents of memory location 1234, the following instruction will be used:

LD A, (1234H) (Note the use of "()" to denote "contents of.")

The instruction will be stored in the memory as follows:

address	PC :3A	(opcode)
	PC + 1:34	(low order half of the address)
	PC + 2:12	(high order half of the address)

Note that the address is stored in "reverse order" in the instruction itself:

3A	low addr	high addr
----	----------	-----------

All the working registers may also be loaded with any specified eight-bit value, or "literal," contained in the second byte of the instruction (this is called *immediate addressing*). An example is:

LD E, 12H

which loads register E with the value 12 hexadecimal.

In the memory, the instruction appears as:

PC: 1E	(opcode)
PC + 1: 12	(literal operand)

PROGRAMMING THE Z80

As a result of this instruction, the immediate operand, or literal value will be contained in register E.

The *indexed addressing* mode is also available for loading register contents, and will be fully described in the next chapter on addressing techniques. Other miscellaneous possibilities exist for loading specific registers, and a table listing all the possibilities is shown in Figure 4.2 (tables supplied by Zilog, Inc.). The grey areas show instructions common with the 8080A.

		SOURCE															
		IMPLIED		REGISTER				REG INDIRECT		INDEXED		EXT. ADDR.		IMME.			
REGISTER	I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX + d)	(IY + d)	(nn)	n	
	A	ED 57	ED 5F	DD 7E d	FD 46 d	FD 46 d											
	B			DD 56 d	FD 5E d	FD 5E d											
	C			DD 4E d	FD 56 d	FD 56 d											
	D			DD 56 d	FD 5E d	FD 5E d											
	E			DD 5E d	FD 66 d	FD 66 d											
	H			DD 66 d	FD 6E d	FD 6E d											
	L			DD 6E d													
DESTINATION	(HL)		DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d					DD 76 d			
	(BC)		FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d					FD 76 d			
	(DE)																
INDEXED	(IX+d)		DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d								
	(IY+d)		FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d								
EXT. ADDR	(nn)																
	IMPLIED		I		ED 47												
		R		ED 4F													

Fig. 4.2: Eight-Bit Load Group—‘LD’

16-Bit Data Transfers

Basically, any of the 16-bit register pairs, BC, DE, HL, SP, IX, IY, may be loaded with a literal 16-bit operand, or from a specified memory address (*extended addressing*), or from the top of the stack, i.e., from the address contained in SP. Conversely, the contents of these

register pairs may be stored in the same manner at a specified memory address or on top of the stack. Additionally, the SP register may be loaded from HL, IX, and IY. This facilitates creating multiple stacks. The register pair AF may also be pushed on top of the stack.

The table listing all the possibilities is shown in Figure 4.3. The stack push and pop operations are included as parts of the 16-bit data transfers. All stack operations transfer the contents of a register pair to or from the stack. Note that there are no single push and pop instructions for saving individual eight-bit registers.

		SOURCE									
		REGISTER							IMM. EXT.	EXT. ADDR.	REG. INDIR.
		AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
DESTINATION	REGISTER	AF									
	BC								91 n n	ED 4B n n	C1
	DE								11 n n	ED 5B n n	D1
	HL								21 n n	2A n n	E1
	SP				41 n n		DD F9	FD F9	31 n n	ED 7B n n	
	IX								DD 21 n n	DD 2A n n	DD E1
	IY								FD 21 n n	FD 2A n n	FD E1
PUSH INSTRUCTIONS	EXT. ADDR.	(nn)		ED 43 n n	ED 53 n n	— —	ED 73 n n	DD 22 n n	FD 22 n n		
REG. IND.	(SP)	— —	— —	— —	— —	— —		DD E5	FD E5		

NOTE: The Push & Pop Instructions adjust
the SP after every execution

↑
POP
INSTRUCTIONS

Fig. 4.3: 16-Bit Load Group—‘LD’, ‘PUSH’ and ‘POP’

A double-byte push or pop is always executed on a register pair: AF, BC, DE, HL, IX, IY (see the bottom row and right-most column in Figure: 4.3).

When operating on AF, BC, DE, HL, a single-byte is required for the instruction, resulting in good efficiency. For example, assume that the

stack pointer SP contains the value “0100”. The following instruction is executed:

PUSH AF

When pushing the contents of the register pair on the stack, the stack pointer SP is first decremented, then the contents of register A are deposited on top of the stack. Then the SP is decremented again, and the contents of F are deposited on the stack. At the end of the stack transfer, SP points to the top element of the stack, which in our example is the value of F.

It is important to remember that, in the case of the Z80, the SP points to the *top* of the stack and the SP is *decremented* whenever a register pair is pushed. Other conventions are often used in other processors, and this may be a source of confusion.

		IMPLIED ADDRESSING				
		AF'	BC', DE' & HL'	HL	IX	IY
IMPLIED	AF	08				
	BC, DE & HL		D9			
	DE			EB		
REG. INDIR.	(SP)			E3	DD E3	FD E3

Fig. 4.4: Exchanges ‘EX’ and ‘EXX’

Exchange Instructions

Additionally, a specialized mnemonic EX has been reserved for exchange operations. EX is not a simple data transfer, but a dual data transfer. It actually changes the contents of *two* specified locations. EX

may be used to exchange the top of the stack with HL, IX, IY and also to swap the contents of DE and HL and AF and AF' (remember that AF' stands for the other AF register pair available in the Z80).

Finally, a special EXX instruction is available to exchange the contents of BC, DE, HL with the contents of the corresponding registers in the second register bank of the Z80.

The possible exchanges are summarized in Figure 4.4.

		SOURCE	
		REG. INDIR.	(HL)
DESTINATION	REG. INDIR.	ED A0	'LDI' – Load (DE) ← (HL) Inc HL & DE, Dec BC
		ED B0	'LDIR,' – Load (DE) ← (HL) Inc HL & DE, Dec BC, Repeat until BC = 0
		ED A8	'LDD' – Load (DE) ← (HL) Dec HL & DE, Dec BC
		ED B8	'LDDR' – Load (DE) ← (HL) Dec HL & DE, Dec BC, Repeat until BC = 0

Reg HL points to source
 Reg DE points to destination
 Reg BC is byte counter

Fig. 4.5: Block Transfer Group

Block Transfer Instructions

Block transfer instructions are instructions which will result in the transfer of a block of data rather than a single or double byte. Block transfer instructions are more complex for the manufacturer to implement than most instructions and are usually not provided on microprocessors. They are convenient for programming, and may improve the

performance of a program, especially during input/output operation. Their use and advantages will be demonstrated throughout this book. Some automatic block transfer instructions are available in the case of the Z80. They use specific conventions.

All block transfer instructions require the use of three pairs of registers: BC, DE, HL:

BC is used as a 16-bit counter. This means that up to $2^{16} = 64K$ bytes may be moved automatically. HL is used as the source pointer. It may point anywhere in the memory. DE is used as the destination pointer and may point anywhere in the memory.

Four block transfer instructions are provided:

LDD, LDDR, LDI, LDIR

All of them decrement the counter register BC with each transfer. Two of them decrement the pointer registers DE and HL, LDD and LDDR, while the two others increment DE and HL, LDI and LDIR. For each of these two groups of instructions, the letter R at the end of the mnemonic indicates an automatic repeat. Let us examine these instructions.

LDI stands for "load and increment." It transfers one byte from the memory location pointed to by H and L to the destination in the memory pointed to by D and E. It also decrements BC. It will automatically increment H and L and D and E so that all register pairs are properly conditioned to perform the next byte transfer whenever required.

LDIR stands for "load increment and repeat," i.e., execute LDI repeatedly until the counter registers BC reach the value "0". It is used to move a continuous block of data automatically from one memory area to another.

LDD and LDDR operate in the same way except that the address pointer is *decremented* rather than incremented. The transfer therefore starts at the *highest* address in the block instead of the lowest. The effect of the four instructions is summarized in Figure 4.5.

Similar automated instructions are available for CP (compare) and are summarized in Figure 4.6.

Data Processing Instructions

Arithmetic

Two main arithmetic operations are provided: addition and subtraction. They have been used extensively in the previous chapter. There are two types of addition, with and without carry, ADC and ADD respec-

SEARCH LOCATION	
REG. INDIR.	
(HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR', Inc HL, Dec BC repeat until BC = 0 or find match
ED A9	'CPD' Dec HL & BC
ED B9	'CPDR' Dec HL & BC Repeat until BC = 0 or find match

HL points to location in memory
 to be compared with accumulator
 contents
 BC is byte counter

Fig. 4.6: Block Search Group

tively. Similarly, two types of subtraction are provided with and without carry. They are SBC and SUB.

Additionally, three special instructions are provided: DAA, CPL, and NEG. The Decimal Adjust Accumulator instruction DAA has been used to implement BCD operations. It is normally used for each BCD add or subtract. Two complementation instructions also are available. CPL will compute the one's complement of the accumulator, and NEG will negate the accumulator into its complement format (two's complement).

All the previous instructions operate on eight-bit data. 16-bit operations are more restricted. ADD, ADC, and SBC are available on specific registers, as described in Figure 4.8.

Finally, increment and decrement instructions are available which operate on all the registers, both in an eight-bit and a 16-bit format. They are listed in Figure 4.7 (eight-bit operations) and 4.8 (16-bit operations).

PROGRAMMING THE Z80

	REGISTER ADDRESSING								(HL)	(IX+d)	(IY+d)	IMMED.
	A	B	C	D	E	H	L	n				
'ADD'	07	00	01	02	03	04	05	06	DD 86 d	FD 86 d	CB n	
ADD w CARRY 'ADC'	0F	00	00	0A	00	0C	0D	0E	DD 8E d	FD 8E d	CE n	
SUBTRACT 'SUB'	07	00	01	02	03	04	05	06	DD 96 d	FD 96 d	DE n	
SUB w CARRY 'SBC'	0F	00	00	0A	00	0C	0D	0E	DD 9E d	FD 9E d	DE n	
'AND'	A7	A6	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	EE n	
'XOR'	AF	A8	A9	AA	A8	AC	AD	AE	DD AE d	FD AE d	EE n	
'OR'	07	00	01	02	03	04	05	06	DD B6 d	FD B6 d	FB n	
COMPARE 'CP'	BF	00	00	0A	00	0C	0D	0E	DD BE d	FD BE d	FE B	
INCREMENT 'INC'	3C	00	0C	10	1C	24	2C	20	DD 34 d	FD 34 d		
DECREMENT 'DEC'	3D	00	0D	15	1D	25	2D	35	DD 35 d	FD 35 d		

Fig. 4.7: Eight-Bit Arithmetic and Logic

Note that, in general, all arithmetic operations modify some of the flags. Their effect is fully described in the instruction descriptions later in this chapter. However, it is important to note that the INC and DEC instructions which operate on register pairs do not modify any of the flags. This detail is important to keep in mind. This means that if you increment or decrement one of the register pairs to the value "0", the Z-bit in the flags register F will not be set. The value of the register must be explicitly tested for the value "0" in the program.

Also, it is important to remember that the instructions ADC and SBC always affect all the flags. This does not mean that all the flags will necessarily be different after their execution. However, they might.

		SOURCE					
DESTINATION		BC	DE	HL	SP	IX	IY
	HL	09	19	29	39		
	IX	DD 09	DD 19		DD 39	DD 29	
	IY	FD 09	FD 19		FD 39		FD 29
ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
INCREMENT 'INC'		03	13	23	33	DD 23	FD 23
DECREMENT 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

Fig. 4.8: Sixteen-Bit Arithmetic and Logic

Logical

Three logical operations are provided: AND, OR (inclusive) and XOR (exclusive), plus a comparison instruction CP. They all operate exclusively on eight-bit data. Let us examine them in turn. (A table listing all the possibilities and operation codes for these instructions is part of Figure 4.7.)

AND

Each logical operation is characterized by a *truth table*, which expresses the logical value of the result in function of the inputs. The truth table for AND appears below:

PROGRAMMING THE Z80

$$\begin{array}{l}
 0 \text{ AND } 0 = 0 \\
 0 \text{ AND } 1 = 0 \\
 1 \text{ AND } 0 = 0 \\
 1 \text{ AND } 1 = 1
 \end{array}
 \quad \text{or} \quad$$

AND	0	1
0	0	0
1	0	1

The AND operation is characterized by the fact that the output is “1” only if both inputs are “1”. In other words, if one of the inputs is “0”, it is guaranteed that the result is “0”. This feature is used to zero a bit position in a word. This is called “masking.”

One of the important uses of the AND instruction is to clear or “mask out” one or more specified bit positions in a word. Assume for example that we want to zero the right-most four-bit positions in a word. This will be performed by the following program:

LD	A, WORD	WORD CONTAINS ‘10101010’
AND	11110000B	‘11110000’ IS MASK

Let us assume that WORD is equal to ‘10101010’. The result of this program is to leave the value ‘10100000’ in the accumulator. “B” is used to indicate a binary value.

Exercise 4.1: Write a three-line program which will zero bits 1 and 6 of WORD.

Exercise 4.2: What happens with a MASK = ‘11111111’?

OR

This instruction is the inclusive OR operation. It is characterized by the following truth table:

$$\begin{array}{l}
 0 \text{ OR } 0 = 0 \\
 0 \text{ OR } 1 = 1 \\
 1 \text{ OR } 0 = 1 \\
 1 \text{ OR } 1 = 1
 \end{array}
 \quad \text{or} \quad$$

OR	0	1
0	0	1
1	1	1

The logical OR is characterized by the fact that if one of the operands is “1”, then the result is always “1”. The obvious use of OR is to set any bit in a word to “1”.

Let us set the right-most four bits of WORD to 1’s. The program is:

LD	A, WORD
OR	00001111B

Let us assume that WORD did contain '10101010'. The final value of the accumulator will be '10101111'.

Exercise 4.3: What would happen if we were to use the instruction OR 10101111 B?

Exercise 4.4: What is the effect of ORing with "FF" hexadecimal?

XOR

XOR stands for "exclusive OR." The exclusive OR differs from the inclusive OR that we have just described in one respect: the result is "1" only if one, and only one, of the operands is equal to "1". If both operands are equal to "1", the normal OR would give a "1" result. The exclusive OR gives a "0" result. The truth table is:

$$\begin{array}{l} 0 \text{ XOR } 0 = 0 \\ 0 \text{ XOR } 1 = 1 \\ 1 \text{ XOR } 0 = 1 \\ 1 \text{ XOR } 1 = 0 \end{array}$$

or

XOR	0	1
0	0	1
1	1	0

The exclusive OR is used for comparisons. If any bit is different, the exclusive OR of two words will be non-zero. In addition, in the case of the Z80, the exclusive OR may be used to *complement* a word, since there is no complement instruction on anything but the accumulator. This is done by performing the XOR of a word with all ones. The program appears below:

```
LD A, WORD
XOR, 11111111 B
```

Let us assume that WORD contained "10101010". The final value of the register will be "01010101". You can verify that this is the complement of the original value.

XOR can be used to advantage as a "bit toggle."

Exercise 4.5: What is the effect of XOR using a register with "00" hexadecimal?

Skew Operations (Shift and Rotate)

Let us first differentiate between the shift and the rotate operations, which are illustrated in Figure 4.9. In a shift operation, the contents of

the register are shifted to the left or to the right by one bit position. The bit which falls out of the register goes into the carry bit C, and the bit which comes in is zero. This was explained in the previous section.

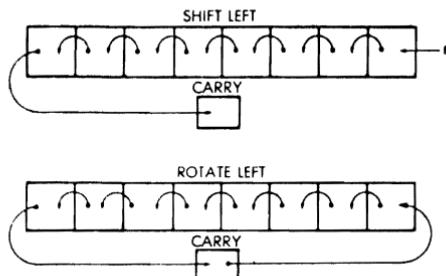


Fig. 4.9: Shift and Rotate

One exception exists: it is the *shift-right-arithmetic*. When performing operations on negative numbers in the two's complement format, the left-most bit is the sign bit. In the case of negative numbers it is “1”. When dividing a negative number by “2” by shifting it to the right, it should remain negative, i.e., the left-most bit should remain a “1”. This is performed automatically by the SRA instruction or Shift Right Arithmetic. In this arithmetic shift right, the bit which comes in on the left is identical to the sign bit. It is “0” if the left-most bit was a “0”, and “1” if the left-most bit was a “1”. This is illustrated on the right of Figure 4.10, which shows all the possible shift and rotate operations.

Rotations

A rotation differs from a shift by the fact that the bit coming into the register is the one which will fall from either the other end of the register or the carry bit. Two types of rotations are supplied in the case of the Z80: an eight-bit rotation and a nine-bit rotation.

The nine-bit rotation is illustrated in Figure 4.11. For example, in the case of a right rotation, the eight bits of the register are shifted right by one bit position. The bit which falls off the right part of the register goes, as usual, into the carry bit. At this time the bit which comes in on the left end of the register is the previous value of the carry bit (before it is overwritten with the bit falling out.) In mathematics this is called a nine-bit rotation since the eight bits of the register plus the ninth bit (the

THE Z80 INSTRUCTION SET

carry bit) are rotated to the right by one bit position. Conversely, the left rotation accomplishes the same result in the opposite direction.

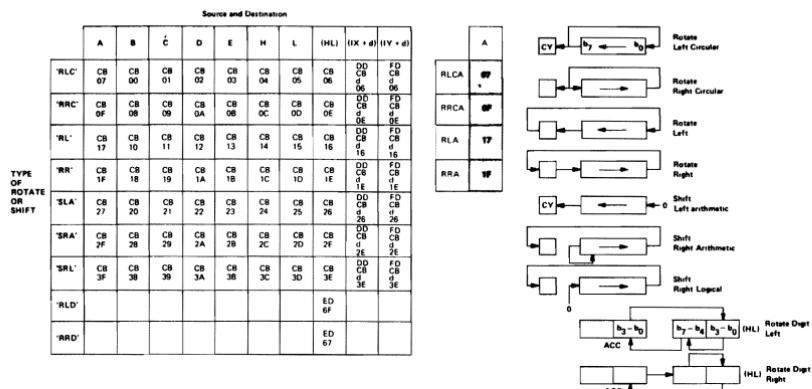


Fig. 4.10: Rotates and Shifts

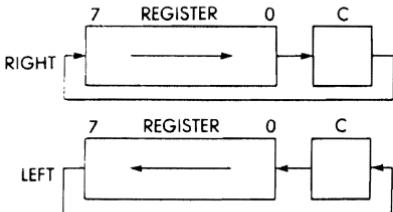


Fig. 4.11: Nine-Bit Rotation

The eight-bit rotation operates in a similar way. Bit 0 is copied into bit seven, or else bit seven is copied into bit 0, depending on the direction of the rotation. In addition, the bit coming out of the register is also copied in the carry bit. This is illustrated by Figure 4.12.

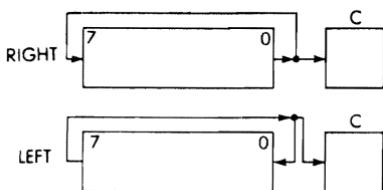


Fig. 4.12: Eight-Bit Rotation

Special Digit Instructions

Two special digit-rotate instructions are provided to facilitate BCD arithmetic. The result is a four-bit rotation between two digits contained in the memory location pointed to by the HL registers and one digit in the lower half of the accumulator. This is illustrated by Figure 4.13.

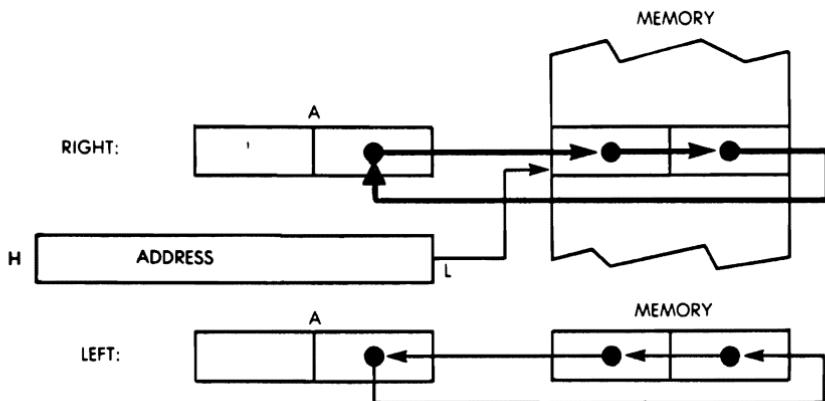


Fig. 4.13: Digit Rotate Instructions (Rotate Decimal)

Bit Manipulation

It has been shown above how the logical operations may be used to set or reset bits or groups of bits in the accumulator. However, it is convenient to set or reset any bit in any register or memory location with a single instruction. This facility requires a considerable number of opcodes and is therefore usually not provided on most microprocessors. However, the Z80 is equipped with extensive bit-manipulation facilities. They are shown in Figure 4.14. This table also includes the test instructions which will be described only in the next section.

Two special instructions are also available for operating on the carry flag. They are CCF (Complement Carry Flag) and SCF (Set Carry Flag). They are shown in Figure 4.15.

Test and Jump

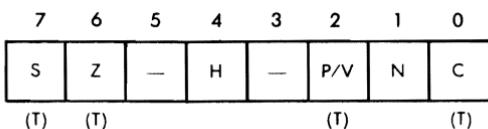
Since testing operations rely heavily on the use of the flags register, we will here describe in detail the role of each of the flags. The contents of the flags register appear in Figure 4.16.

THE Z80 INSTRUCTION SET

BIT	REGISTER ADDRESSING							(HL)	REG. INDIR.	INDEXED	
	A	B	C	D	E	H	L			(IX+d)	(IY+d)
TEST 'BIT'	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	CB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
RESET 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
SET BIT 'SET'	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

Fig. 4.14: Bit Manipulation Group

Decimal Adjust Acc, 'DAA'	27
Complement Acc, 'CPL'	2F
Negate Acc, 'NEG' (2's complement)	ED 44
Complement Carry Flag, 'CCF'	3F
Set Carry Flag, 'SCF'	37

Fig. 4.15: General-Purpose AF Operations**Fig. 4.16: The Flags Register**

C is the carry, N is add or subtract, P/V is parity or overflow, H is half carry, Z is zero, S is sign. Bits 3 and 5 of the flags register are not used (“—”). The two flags H and N are used for BCD arithmetic and cannot be tested. The other four flags (C, P/V, Z, S) can be tested in conjunction with conditional jump or call instructions.

The role of each flag will now be described.

Carry (C)

In the case of nearly all microprocessors, and of the Z80 in particular, the carry bit assumes a dual role. First, it is used to indicate whether an addition or subtraction operation has resulted in a carry (or borrow). Secondly, it is used as a ninth bit in the case of shift and rotate operations. Using a single bit to perform both roles facilitates some operations, such as a multiplication operation. This should be clear from the explanation of the multiplication which has been presented in the previous chapter.

When learning to use the carry bit, it is important to remember that all arithmetic operations will either set it or reset it, depending on the result of the instructions. Similarly, all shift and rotation operations use the carry bit and will either set it or reset it, depending on the value of the bit which comes out of the register.

In the case of logical instructions (AND, OR, XOR), the carry bit will always be reset. They may be used to zero the carry explicitly.

Instructions which affect the carry bit are: ADD A,s; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; ADD DD,ss; ADC HL,ss; SBC HL,ss; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; DAA; SCF; CCF.

Subtract (N)

This flag is normally not used by the programmer, and is used by the Z80 itself during BCD operations. The reader will remember from the previous chapter that, following a BCD add or subtract, a DAA (Decimal Adjust Accumulator) instruction is executed to obtain the valid BCD results. However, the "adjustment" operation is different after an addition and after a subtraction. The DAA therefore executes differently depending on the value of the N flag. The N flag is set to "0" after an addition and is set to a "1" after a subtraction.

The symbol used for this flag, "N", may be confusing to programmers who have used other processors, since it may be mistaken for the sign bit. It is an internal operation sign bit.

N is set to "0" by: ADD A,s; ADC A,s; ANDs; ORs; XORs; INCs; ADD DD,ss; ADC HL,ss; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; SCF; CCF; IN r, (C); LDI; LDD; LDIR; LDDR; LD A, I; LD A, R; BIT b, s.

N is set to "1" by: SUB s; SBC A,s; CP s; NEG; DEC m; SBC HL, ss; CPL; INI; IND; OUTI; OUTD; INIR; INDR; OTIR; OTDR; CPI; CPIR; CPD; CPDR.

Parity/Overflow (P/V)

The parity/overflow flag performs two different functions. Specific instructions will set or reset this flag depending on the parity of the result; parity is determined by counting the total number of ones in the result. If this number is odd, the parity bit will be set to "0" (odd parity). If it is even, the parity bit will be set to "1" (even parity). Parity is most frequently used on blocks of characters (usually in the ASCII format). The parity bit is an additional bit which is added to the seven-bit code representing the character, in order to verify the integrity of data which has been stored in a memory device. For example, if one bit in the code representing the character has been changed by accident, due

to a malfunction in the memory device (such as a disk or RAM memory), or during transmission, then the total number of ones in the seven-bit code will have been changed. By checking the parity bit, the discrepancy will be detected, and an error will be flagged. In particular, the flag is used with logical and rotate instructions. Also, naturally, during an input operation from an I/O device, the parity flag will indicate the parity of the data being read.

For the reader familiar with the Intel 8080, note that the parity flag in the 8080 is used exclusively as such. In the case of the Z80, it is used for several additional functions. This flag should therefore be handled with care when going from one of the microprocessors to the other.

In the case of the Z80, the second essential use of this flag is as an overflow flag (not available in the 8080). The overflow flag has been described in Chapter 1, when the two's complement notation was introduced. It detects the fact that, during an addition or subtraction, the sign of the result is "accidentally" changed due to the overflow of the result into the sign bit. (Recall that, using an eight-bit representation, the largest positive number is +127, and the smallest negative number is -128 in two's complement.)

Finally, this bit is also used, in the case of the Z80, for two unrelated functions.

During the block transfer instructions (LDD, LDDR, LDI, LDIR), and during the search instructions (CPD, CPDR, CPI, CPIR), this flag is used to detect whether the counter register B has attained the value "0". With decrementing instructions, this flag is reset to "0" if the byte counter register pair is "0". When incrementing, it is reset if BC - 1 = 0 at the beginning of the instruction, i.e., if BC will be decremented to "0" by the instruction.

Finally, when executing the two special instructions LD A,I and LD A,R, the P/V flag reflects the value of the interrupt enable flip-flop (IFF2). This feature can be used to preserve or test this value.

The P flag is affected by: AND s; OR s; XOR s; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; IN r,(C).

The V flag is affected by: ADD A,s; ADC A,s; SUB s; SBC A,s; CP s; NEG; INC s; DEC m; ADC HL,ss; SBC HL,ss.

It is also used by: LDIR; LDDR (set to "0"); LDI; LDD; CPI; CPIR; CPD; CPDR.

The Half-Carry Flag (H)

The half-carry flag indicates a possible carry from bit 3 into bit 4 during an arithmetic operation. In other words, it represents the carry from

the low-order nibble (group of 4 bits) into the high order one. Clearly, it is primarily used for BCD operations. In particular, it is used internally within the microprocessor by the Decimal Adjust Accumulator (DAA) instruction in order to adjust the result to its correct value.

This flag will be set during an addition when there is a carry from bit 3 to bit 4 and reset when there is no carry. Conversely, during a subtract operation, it will be set if there is a borrow from bit 4 to bit 3, and reset if there is no borrow.

The flag will be conditioned by addition, subtraction, increment, decrement, comparisons, and logical operations.

Instructions which affect the H bit are: ADD A,r ; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SR m; SRL m; RLD; RRD; DAA; CPL; SCF; IN r,(C) ; LDI; LLD; LDIR; LDDR; LD A; LD A,R; BIT b,r; CPI; CPIR; CPD; CPDR.

Note that the H bit is randomly affected by the 16-bit add and subtract instructions, and by block input and output instructions.

Zero (Z)

The Z flag is used to indicate whether the value of a byte which has been computed, or is being transferred, is zero. It is also used with comparison instructions to indicate a match, and for other miscellaneous functions.

In the case of an operation resulting in a zero result, or of a data transfer, the Z bit is set to "1" whenever the byte is zero. Z is reset to "0" otherwise.

In the case of comparison instructions, the Z bit is set to "1" whenever the comparison succeeds and to "0" otherwise.

Additionally, in the case of the Z80, it is used for three more functions: it is used with the BIT instruction to indicate the value of a bit being tested. It is set to "1" if the specified bit is "0" and reset otherwise.

With the special "block input-output instructions" (INI, IND, OUTI, OUTD), the Z flag is set if $D - 1 = 0$, and reset otherwise; it is set if the byte counter will decrement to "0" (INIR, INDR, OTIR, OTDR).

Finally, with the special instructions IN r,(C), the Z flag is set to "1" to indicate that the input byte has the value "0".

In summary, the following instructions condition the value of the Z bit: ADD A,s; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; ADC HL, ss; SBC HL,ss; RL m; RLC m;

RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; IN r,(C); INI; IND; OUTI; OUTD; INIR; INDR; OTIR; OTDR; CPI; CPIR; CPD; CPDR; LD A, I; LD A, R; BIT b,s; NEG s.

Usual instructions which do not affect the Z bit are: ADD DD,ss; RLA; RLCA; RRA; RRCA; CPL; SCF; CCF; LDI; LDD; LDIR; LDDR; INC DD; DEC DD.

Sign (S)

This flag reflects the value of the most significant bit of a result or of a byte being transferred (bit seven). In two's complement notation, the most significant bit is used to represent the sign. "0" indicates a positive number and a "1" indicates a negative number. As a result, bit seven is called the sign bit.

In the case of most microprocessors, the sign bit plays an important role when communicating with input/output devices. Most microprocessors are not equipped with a BIT instruction for testing the contents of any bits in a register or the memory. As a result, the sign bit is usually the most convenient bit to test. When examining the status of an input/output device, reading the status register will automatically condition the sign bit, which will be set to the value of bit seven of the status register. It can then be tested conveniently by the program. This is why the status register of most input/output chips connected to microprocessor systems have their most important indicator (usually ready/not ready) in bit position seven.

A special BIT instruction is provided in the case of the Z80. However, in order to test a memory location (which may be the address of an I/O status register), the address must first be loaded into registers IX, IY or HL. There is no bit instruction provided to test a specified memory address directly (i.e., no direct addressing mode for this instruction). The value of positioning an input/output ready flag in bit position seven, therefore, remains intact, even in the case of the Z80.

Finally, the sign flag is used by the special instruction IN, (C) to indicate the sign of the data being read.

Instructions which affect the sign bit are: ADD A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; ADC HL, ss; SBC HL, ss; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD ; RRD; DAA; IN r,(C); CPI; CPIR; CPD; CPDR; LD A,I;LD A,r; NEG, ADC A,s.

Summary of the Flags

The flag bits are used to automatically detect special conditions within the ALU of the microprocessor. They can be conveniently tested by specialized instructions, so that specific action can be taken in response to the condition detected. It is important to understand the role of the various indicators available, since most decisions taken within the program will be taken in function of these flag bits. All jumps executed within a program will jump to specified locations depending on the status of these flags. The only exception involves the interrupt mechanism, which will be described in the chapter on input/output and may cause jumping to specific locations whenever a hardware signal is received on specialized pins of the Z80.

At this point, it is only necessary to remember the main function of each of these bits. When programming, the reader can refer to the description of the instruction later in this chapter to verify the effect of every instruction of the various flags. Most flags can be ignored most of the time, and the reader who is not yet familiar with them should not feel intimidated by their apparent complexity. Their use will become clearer as we examine more application programs.

A summary of the six flags and the way they are set or reset by the various instructions is shown in Figure 4.17.

The Jump Instructions

A branch instruction is an instruction which causes a forced branching to a specified program address. It changes the normal flow of execution of the program from a sequential mode into one where a different segment of the program is suddenly executed. Jumps may be conditional or unconditional. An unconditional jump is one in which the branching occurs to a specific address, regardless of any other condition.

A conditional jump is one which occurs to a specific address only if one or more conditions are met. This is the type of jump instruction used to make decisions based upon data or computed results.

In order to explain the conditional jump instructions, it is necessary to understand the role of the flags register, since all branching decisions are based upon these flags. This was the purpose of the preceding section. We can now examine in more detail the jump instructions provided by the Z80.

Two main types of jump instructions are provided: jump instructions within the main program (they are called "jumps"), and the special

PROGRAMMING THE Z80

INSTRUCTION	C	Z	P V	S	N	H	COMMENTS
ADD A, s; ADC A, s SUB s; SBC A, s, CP s, NEG	t	t	V	t	0	t	8-bit add or add with carry 8-bit subtract, subtract with carry, compare and negate accumulator
AND s OR s; XOR s INC s DEC m	0	:	P	t	0	1	Logical operations And sets different flags
INC s DEC m	•	:	V	:	0	:	8-bit increment 8-bit decrement
ADD DD, ss ADC HL, ss SBC HL, ss	:	•	V	:	1	:	16-bit add 16-bit add with carry
RLA; RLCA, RRA, RRCA RL m; RLC m; RR m; RRC m SLA m; SRA m; SRL m	:	•	•	•	0	X	16-bit subtract with carry Rotate accumulator
RLD, RRD DAA CPL SCF CCF IN r, (C) INI; IND; OUTI; OUTD INIR; INDR; OTIR; OTDR LDI, LDD LDIR, LDOR	•	:	P	:	0	0	Rotate digit left and right Decimal adjust accumulator Complement accumulator Set carry Complement carry Input register indirect Block input and output Z = 0 if B ≠ 0 otherwise Z = 1
LDI, LDD LDIR, LDOR	•	X	:	X	0	0	Block transfer instructions P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI, CPIR, CPD, CPDR	•	:	:	t	1	X	Block search instructions Z = 1 if A = (HL), otherwise Z = 0 P/V = 1 if BC ≠ 0, otherwise P/V = 0
LD A, I; LD A, R	•	:	IFF	:	0	0	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s NEG	•	:	X	X	0	1	The complement of bit b of location is copied into the Z flag Negate accumulator

Courtesy of Zilog, Inc.

The following notation is used in this table:

SYMBOL	OPERATION
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract.
	H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
:	The flag is affected according to the result of the operation.
*	The flag is unchanged by the operation.
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care."
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8-bit location for all the addressing modes allowed for the particular instruction.
ss	Any 16-bit location for all the addressing modes allowed for that instruction.
ii	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range <0, 255>.
nn	16-bit value in range <0, 65535>.
m	Any 8-bit location for all the addressing modes allowed for the particular instruction.

Fig. 4.17: Summary of Flag Operation

type of branch instructions used to jump to a subroutine and to return from it ("call" and "return"). As a result of any jump instruction, the program counter PC will be reloaded with a new address, and the usual program execution will resume from this point on. The full power of the various jump instructions can be understood only in the context of the various addressing modes provided by the microprocessor. This part of the discussion will be deferred until the next chapter, where the addressing modes are discussed. We will only consider here the other aspects of these instructions.

Jumps may be unconditional (branching to a specified memory address) or else conditional. In the case of a conditional jump, one of four flag bits may be tested. They are the Z, C, P/V, and S flags. Each of them may be tested for the value "0" or "1".

The corresponding abbreviations are:

Z = zero ($Z = 1$)
 NZ = non zero ($Z = 0$)
 C = carry ($C = 1$)
 NC = no carry ($C = 0$)
 PO = odd parity
 PE = even parity
 P = positive ($S = 0$)
 M = minus ($S = 1$)

In addition, a special combination instruction is available in the Z80 which will decrement the B register and jump to a specified memory address as long as it is not zero. This is a powerful instruction used to terminate a loop, and it has already been used several times in the previous chapter: it is the DJNZ instruction.

Similarly, the CALL and the RET (return) instructions may be conditional or unconditional. They test the same flags as the branch instruction which we have already described.

The availability of conditional branches is a powerful resource in a computer and is generally not provided on other eight-bit microprocessors. It improves the efficiency of programs by implementing in a single instruction what requires two instructions otherwise.

Finally, two special return instructions have been provided in the case of interrupt routines. They are RETI and RETN. They will be described in the section of Chapter 6 on interrupts.

The addressing modes and the opcodes for the various branches available are shown in Figure 4.18.

PROGRAMMING THE Z80

CONDITION												
			UN-COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG	SIGN POS	REG B=0
JUMP 'JP'	IMMED. EXT.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVE	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'	REG. INDIR.	(HL)	E9									
JUMP 'JP'		(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
'CALL'	IMMED. EXT.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
DECREMENT B, JUMP IF NON ZERO 'DJNZ'	RELATIVE	PC+e										10 e-2
RETURN 'RET'	REGISTER INDIR.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REG. INDIR.	(SP) (SP+1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REG. INDIR.	(SP) (SP+1)	ED 45									

Fig. 4.18: Jump Instructions

A detailed discussion of the various addressing modes is presented in Chapter 5.

By examining Figure 4.18, it becomes apparent that many addressing modes are restricted. For example, the absolute jump JP nn can test four flags, while JR can only test two flags.

Note an important observation: JR tends to be used whenever possible as it is shorter than JP (one less byte) and facilitates program relocation. However, JR and JP are not interchangeable: JR cannot test the parity or the sign flags.

One more type of specialized branch is available; this is the *restart* or RST instruction. It is a one-byte instruction which allows jumping to any one of eight starting addresses at the low end of the memory. Its starting addresses are, in decimal, 0, 8, 16, 24, 32, 40, 48 and 56. It is a powerful instruction because it is implemented in a single byte. It provides a fast branch, and for this reason is used essentially to respond to interrupts. However, it is also available to the programmer for other uses. A summary of the opcodes for this instruction is shown in Figure 4.19.

CALL ADDRESS	OP CODE	
	0000 _H	CF
	0008 _H	CF
	0010 _H	D7
	0018 _H	DF
	0020 _H	E7
	0028 _H	EF
	0030 _H	F7
		'RST 56'

H Indicates a hexadecimal number.

Fig. 4.19: Restart Group

Input/Output Instructions

Input/output techniques will be described in detail in Chapter 6. Simply, input/output devices may be addressed in two ways: as memory locations, using any one of the instructions that have already

been described, or using specific input/output instructions. Usual memory addressing instructions use three bytes: one byte for the op-code and two bytes for the address. As a result, they are slow to execute, since they require three memory accesses. The main purpose of specialized input/output instructions is to provide shorter and, therefore faster, instructions. However, input/output instructions have two disadvantages.

First, they "waste" several of the precious few opcodes available (since usually only 8 bits are used to supply all opcodes necessary for a microprocessor). Secondly, they require the generation of one or more specialized input/output signals, and therefore "waste" one or more of the few pins available in the microprocessor. The number of pins is usually limited to 40. Because of these possible disadvantages, specific input/output instructions are not provided on most microprocessors. They are, however, provided on the original 8080 (the first powerful eight-bit general-purpose microprocessor introduced) and on the Z80, which we know is compatible with the 8080.

The advantage of input/output instructions is to execute faster by requiring only two bytes. However, a similar result can be obtained by supplying a special addressing mode called "page 0" addressing, where the address is limited to a field of eight bits. This solution is often chosen in other microprocessors.

The two basic input/output instructions are IN and OUT. They transfer either the contents of the specified I/O locations into any of the working registers or the contents of the register into the I/O device. They are naturally two bytes long. The first byte is reserved for the op-code, the second byte of the instruction forms the low part of the address. The accumulator is used to supply the upper part of the address. It is therefore possible to select one of the 64K devices. However, this requires that the accumulator be loaded with the appropriate contents every time, and this may slow the execution.

Additionally, the Z80 provides a register-indirect mode, plus four specialized block-transfer instructions for input and output.

In the *register-input* mode, whose format is IN r, (C), the register pair B and C is used as a pointer to the I/O device. The contents of B are placed on the high-order part of the address bus. The contents of the specified I/O device are then loaded into the register designated by r.

The same applies to the OUT instruction.

The four block-transfer instructions on input are: INI, INIR (repeated INI), IND and INDR (repeated IND). Similarly, on output,

they are: OUTI, OTIR, OUTD, and OTDR.

In this automated block transfer, the register pair H and L is used as a destination pointer. Register C is used as the I/O device selector (one out of 256 devices). In the case of the output instruction, H and L point to the source. Register B is used as a counter and can be incremented or decremented. The corresponding instructions on input are INI when incrementing and IND when decrementing.

INI is an automated single-byte transfer. Register C selects the input device. A byte is read from the device and is transferred to the memory address pointed to by H and L. H and L are then incremented by 1, and the counter B is decremented by 1.

INIR is the same instruction, automated. It is executed repeatedly until the counter decrements to "0". Thus, up to 256 bytes may be transferred automatically. Note that to achieve a total transfer of exactly 256, register B should be set to the value "0" prior to executing this instruction.

The opcodes for the input and output instructions are summarized in Figures 4.20 and 4.21.

Control Instructions

Control instructions are instructions which modify the operating mode of the CPU or manipulate its internal status information. Seven such instructions are provided.

The NOP instruction is a no-operation instruction which does nothing for one cycle. It is typically used either to introduce a deliberate delay (4 states = 2 microseconds with a 2MHz clock), or to fill the gaps created in a program during the debugging phase. In order to facilitate program debugging, the opcode for the NOP is traditionally all 0's. This is because, at execution time, the memory is often cleared, i.e., all 0's. Executing NOP's is guaranteed to cause no damage and will not stop the program execution.

The HALT instruction is used in conjunction with interrupts or a reset. It actually suspends the operation of the CPU. The CPU will then resume operation whenever either an interrupt or a reset signal is received. In this mode, the CPU keeps executing NOP's. A halt is often placed at the end of programs during the debugging phase, as there is usually nothing else to be done by the main program. The program must then be explicitly restarted.

Two specialized instructions are used to disable and enable the internal interrupt flag. They are EI and DI. Interrupts will be described in

PROGRAMMING THE Z80

				SOURCE							REG. IND.
				REGISTER							
		A	B	C	D	E	H	L	(HL)		
'OUT'	IMMED.	(n)	D3 n								
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69		
'OUTI' – OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3	BLOCK OUTPUT COMMANDS
'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0	REG. IND.	(C)								ED B3	
'OTUD' – OUTPUT Dec HL & B	REG. IND.	(C)								ED AB	
'OTDR' – OUTPUT, Dec HL & B, REPEAT IF B≠0	REG. IND.	(C)								ED BB	

Fig. 4.20: Output Group

INPUT DESTINATION		SOURCE PORT ADDRESS				
		REG.	IMMED.	REG. INDIR.		
		(n)	(I)			
INPUT 'IN'		A	DB n	ED 78		
		B		ED 40		
		C		ED 48		
		D		ED 50		
		E		ED 58		
		H		ED 60		
		L		ED 68		
'INI' - INPUT & Inc HL, Dec B		REG. INDIR		ED A2		
'INIR' - INP, Inc HL, Dec B, REPEAT IF B=0				ED B2		
'IND' - INPUT & Dec HL, Dec B				ED AA		
'INDR' - INPUT, Dec HL, Dec B, REPEAT IF B=0				ED BA		
BLOCK INPUT COMMANDS						

Fig. 4.21: Input Group

Chapter 6. The interrupt flag is used to authorize or not authorize the interruption of a program. To prevent interrupts from occurring during any specific portion of a program, the interrupt flip-flop (flag) may be disabled by this instruction. It will be used in Chapter 6. These instructions are shown in Figure 4.22.

'NOP'	
'HALT'	
DISABLE INT '(DI)'	
ENABLE INT '(EI)'	
SET INT MODE 0 'IM0'	ED 46
SET INT MODE 1 'IM1'	ED 56
SET INT MODE 2 'IM2'	ED 5E

8080A MODE

CALL TO LOCATION 0038_H

INDIRECT CALL USING REGISTER I AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER.

Fig. 4.22: Miscellaneous CPU Control

Finally, three interrupt modes are provided in the Z80. (Only one is available on the 8080). Interrupt mode 0 is the 8080 mode, interrupt 1 is a call to location 038H, and interrupt mode 2 is an indirect call which uses the contents of the special register I, plus 8 bits provided by the interrupting device as a pointer to the memory location whose contents are the address of the interrupt routine. These modes will be explained in Chapter 6.

which will also be explained in Chapter 6. They are the IRQ and the NMI pins.

SUMMARY

The five categories of instructions available on the Z80 have now been described. The details on individual instructions are supplied in the following section of the book. It is not necessary to understand the role of each instruction in order to start to program. The knowledge of a few essential instructions of each type is sufficient at the beginning. However, as you begin to write programs by yourself, you should learn about all the instructions of the Z80 if you want to write good programs. Naturally, at the beginning, efficiency is not important, and this is why most instructions can be ignored.

One important aspect has not yet been described. This is the set of addressing techniques implemented on the Z80 to facilitate the retrieval of data within the memory space. These addressing techniques will be studied in the next chapter.

THE Z80 INSTRUCTIONS: INDIVIDUAL DESCRIPTION**ABBREVIATIONS**

FLAG	ON	OFF
Carry	C (carry)	NC (no carry)
Sign	M (minus)	P (plus)
Zero	Z (zero)	NZ (non zero)
Parity	PE (even)	PO (odd)

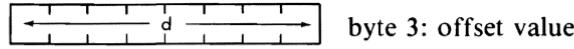
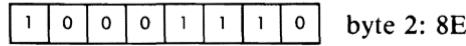
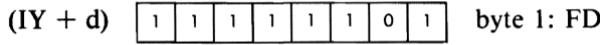
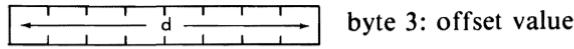
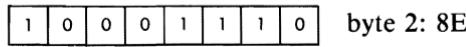
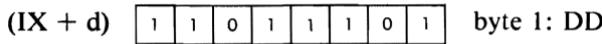
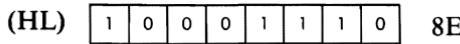
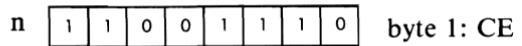
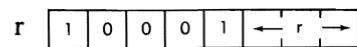
- changed functionally according to operation
 - flag is set to zero
 - 1 flag is set to one
 - ? flag is set randomly by operation
 - X special case, see accompanying note on that page
- bit positions 3 and 5 are always random

PROGRAMMING THE Z80

ADC A, s Add accumulator and specified operand with carry.

Function: $A \leftarrow A + s + C$

Format: s: may be r, n, (HL), (IX + d), or (IY + d)



r may be any one of:

A - 111 E - 011

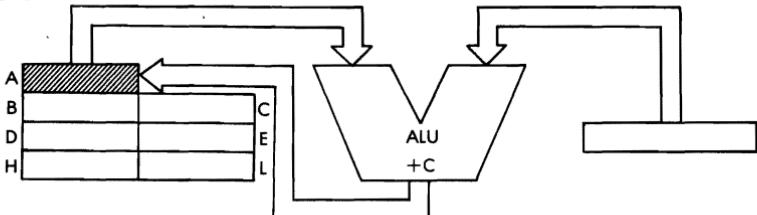
B - 000 H - 100

C - 001 L - 101

D - 010

Description:

The operand s and the carry flag C from the status register are added to the accumulator, and the result is stored in the accumulator. s is defined in the description of the similar ADD instructions.

Data Flow:**Timing:**

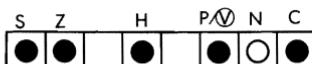
<i>s:</i>	<i>M cycles:</i>	<i>T states:</i>	<i>usec</i> @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Addressing Mode: r: implicit; n: immediate; (HL): indirect; (IX + d), (IY + d): indexed.

Byte Codes:

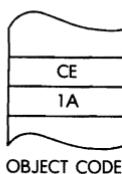
ADC A,r r:

A	B	C	D	E	H	L
8F	88	89	8A	8B	8C	8D

Flags:**Example:**

ADC A, 1A

Before:



After:

A [06 | 13] F A [41 | 10] F

PROGRAMMING THE Z80

ADC HL, ss Add with carry HL and register pair ss.

Function: $HL \leftarrow HL + ss + C$

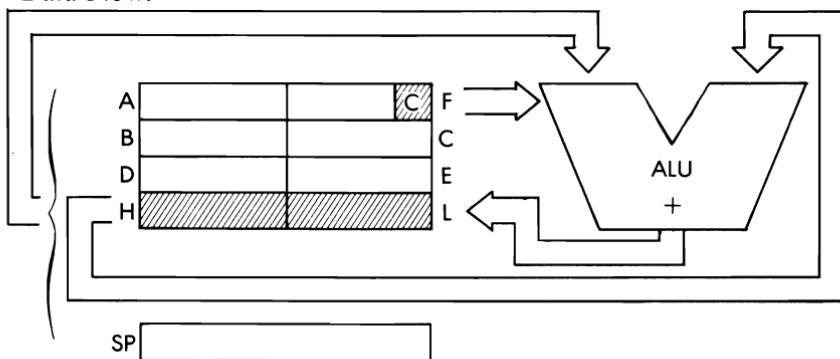
Format:

<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	1	1	1	0	1	1	0	1	byte 1: ED
1	1	1	0	1	1	0	1		
<table border="1"><tr><td>0</td><td>1</td><td>s</td><td>s</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>	0	1	s	s	1	0	1	0	byte 2
0	1	s	s	1	0	1	0		

Description: The contents of the HL register pair are added to the contents of the specified register pair, and then the contents of the carry flag are added. The final result is stored back in HL. ss may be any one of:

BC – 00 HL – 10
DE – 01 SP – 11

Data Flow:

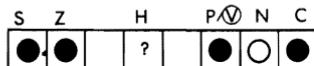


Timing: 4 M cycles; 15 T states: 75 usec @ 2 MHz

Addressing Mode: Implicit.

Byte Codes: ss: BC DE HL SP
 ED –

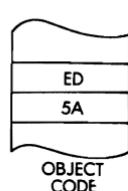
4A	5A	6A	7A
----	----	----	----

Flags:

H is set if there is a carry from bit 11.

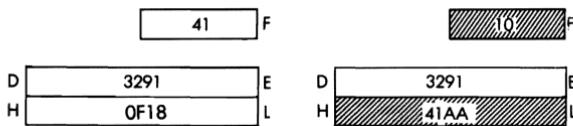
Example:

ADC HL, DE



Before:

After:



PROGRAMMING THE Z80

ADD A, (HL) Add accumulator with indirectly addressed memory location (HL).

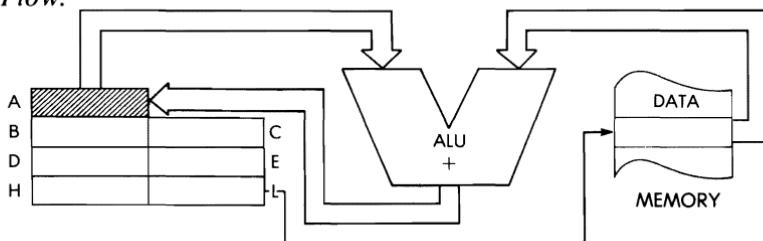
Function: $A \leftarrow A + (HL)$

Format:

1	0	0	0	0	1	1	0	86
---	---	---	---	---	---	---	---	----

Description: The contents of the accumulator are added to the contents of the memory location addressed by the HL register pair. The result is stored in the accumulator.

Data Flow:



Timing: 2 M cycles; 7 T states: 3.5 usec @ 2 MHz

Addressing Mode: Indirect.

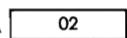
Flags:

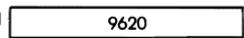
S	Z	H	P/V	N	C
●	●	●	●	○	●

Example:

ADD A, (HL)

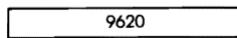
Before:

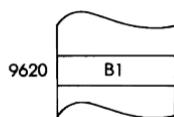
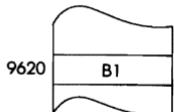
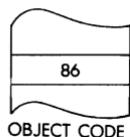
A  02

H  9620

After:

A  B3

H  9620



PROGRAMMING THE Z80

ADD A, (IX + d) Add accumulator with indexed addressed memory location (IX + d)

Function: $A \leftarrow A + (IX + d)$

Format:

1	1	0	1	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: DD

1	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

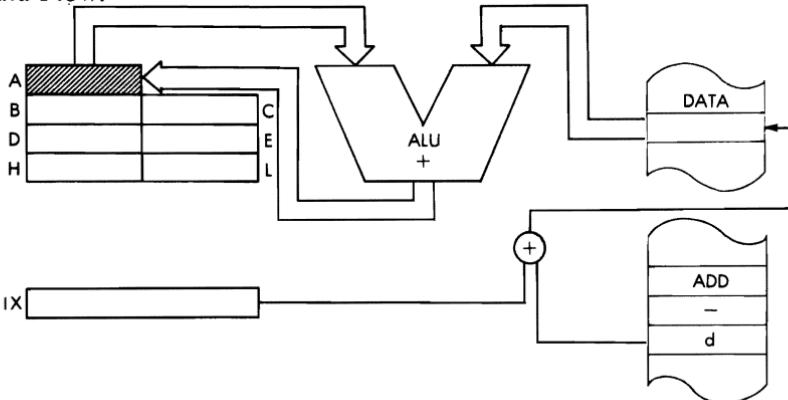
 byte 2: 86

d			
---	--	--	--

 byte 3: offset value

Description: The contents of the accumulator are added to the contents of the memory location addressed by the contents of the IX register plus the immediate offset value. The result is stored in the accumulator.

Data Flow:



Timing: 5 M cycles; 19 T states: 9.5 usec @ 2 MHz

Addressing Mode: Indexed.

Flags:

S	Z	H	P/V	N	C
●	●	●	●	○	●

THE Z80 INSTRUCTION SET

Example: ADD A, (IX + 3)

Before:

A

11

IX

0B61

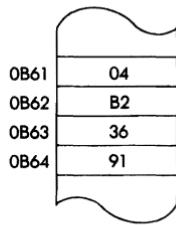
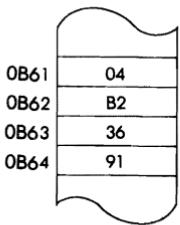
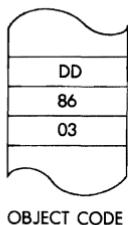
After:

A

A2

IX

0B61



PROGRAMMING THE Z80

ADD A, (IY + d) Add accumulator with indexed addressed memory location (IY + d)

Function: $A \leftarrow A + (IY + d)$

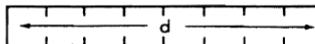
Format:

1	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

byte 1: FD

1	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---

byte 2: 86

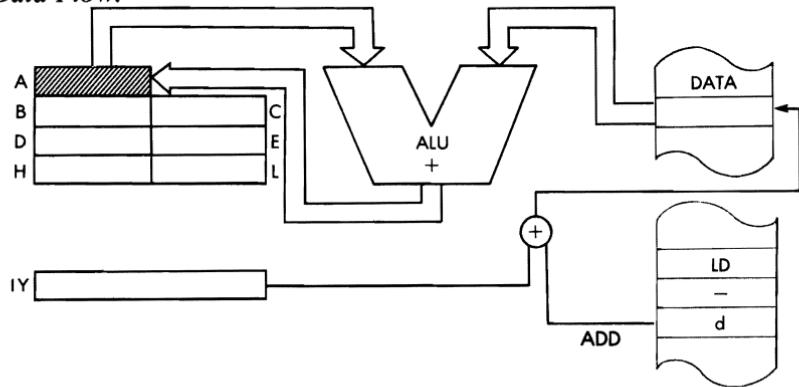


byte 3: offset value

Description:

The contents of the accumulator are added to the contents of the memory location addressed by the contents of the IY register plus the given offset value. The result is stored in the accumulator.

Data Flow:



Timing: 5 M cycles; 19 T states; 9.5 usec @ 2 MHz

Addressing Mode: Indexed.

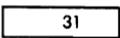
Flags:

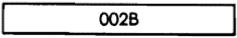
S	Z	H	P/V	N	C
●	●	●	●	○	●

THE Z80 INSTRUCTION SET

Example: ADD A, (IY + 1)

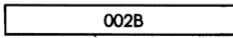
Before:

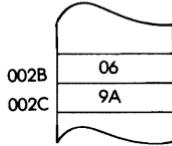
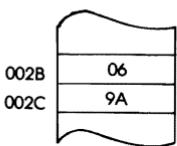
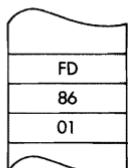
A  31

IX  002B

After:

A  CB

IX  002B



OBJECT
CODE

PROGRAMMING THE Z80

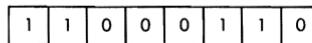
ADD A, n

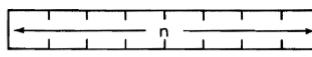
Add accumulator with immediate data n.

Function:

$$A \leftarrow A + n$$

Format:

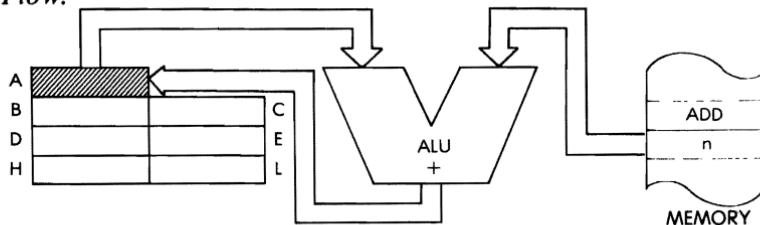
 byte 1: C6

 byte 2: immediate data

Description:

The contents of the accumulator are added to the contents of the memory location immediately following the op code. The result is stored in the accumulator.

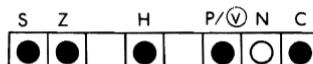
Data Flow:



Timing: 2 M cycles; 7 T states: 3.5 usec @ 2 MHz

Addressing Mode: Immediate.

Flags:

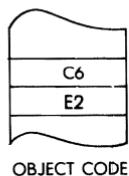


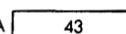
Example:

ADD A, E2

Before:

After:



A  43

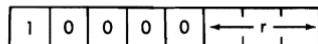
A  125

ADD A, r

Add accumulator with register r.

Function:

$$A \leftarrow A + r$$

Format:*Description:*

The contents of the accumulator are added with the contents of the specified register. The result is placed in the accumulator. r may be any one of:

A - 111

E - 011

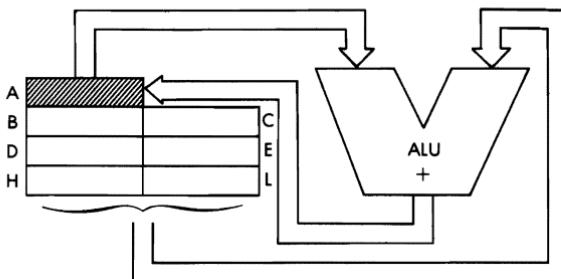
B - 000

H - 100

C - 001

L - 101

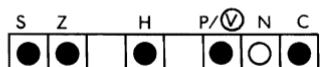
D - 010

Data Flow:*Timing:*

1 M cycle; 4 T states: 2 usec @ 2 MHz.

Addressing Mode: Implicit.*Byte Codes:*

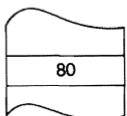
r:	A	B	C	D	E	H	L
	87	80	81	82	83	84	85

Flags:

PROGRAMMING THE Z80

Example: ADD A, B

Before:



OBJECT CODE

A	3D
B	02

After:

A	3F
B	02

ADD HL, ss Add HL and register pair ss.

Function: $HL \leftarrow HL + ss$.

Format:

0	0	s	s	1	0	0	1
---	---	---	---	---	---	---	---

Description: The contents of the specified register pair are added to the contents of the HL register pair and the result is stored in HL. ss may be any one of:

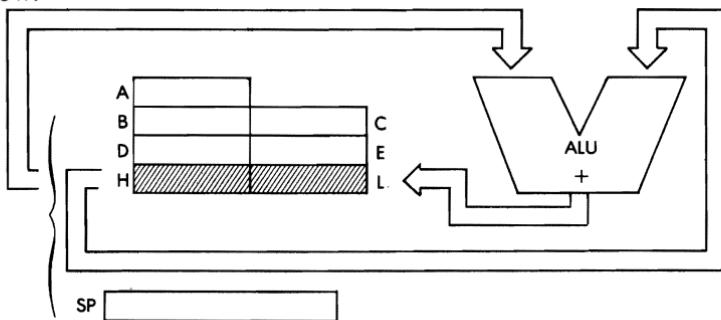
BC – 00

HL – 10

DE – 01

SP – 11

Data Flow:



Timing: 3 M cycles; 11 T states: 5.5 usec @ 2 MHz

Addressing Mode: Implicit.

Byte Codes:

ss:	BC	DE	HL	SP
	09	19	29	39

Flags:

S	Z	H	P/V	N	C
		?		○	●

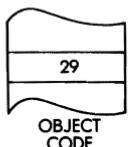
C is set by carry from bit 15, reset otherwise.

H is set by a carry from bit 11

PROGRAMMING THE Z80

Example: ADD HL, HL

Before:



H L

After:

H L

ADD IX, rr Add IX with register pair rr.

Function: $IX \leftarrow IX + rr$

Format:

1	1	0	1	1	1	0	1
byte 1: DD							

0	0	r	r	1	0	0	1
byte 2							

Description: The contents of the IX register are added to the contents of the specified register pair and the result is stored back in IX. rr may be anyone of:

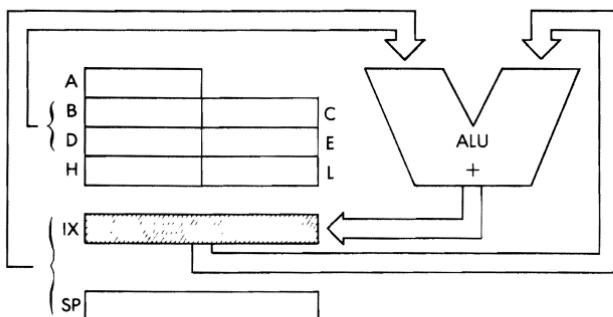
BC – 00

IX – 10

DE – 01

SP – 11

Data Flow:



Timing: 4 M cycles; 15 T states: 7.5 usec @ 2 MHz

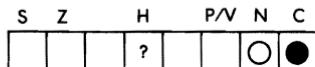
Addressing Mode: Implicit.

Byte Codes: rr: BC DE IX SP
DD-

09	19	29	39
----	----	----	----

PROGRAMMING THE Z80

Flags:



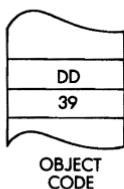
H is set by carry out of bit 11.
C is set by carry from bit 15.

Example:

ADD IX, SP

Before:

After:



ADD IY, rr Add IY and register pair rr.

Function: $IY \leftarrow IY + rr$

Format:

1	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: FD

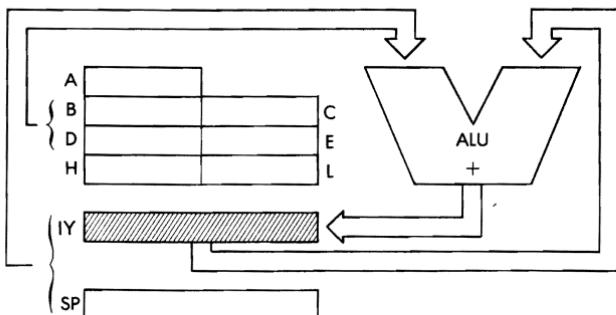
0	0	r	r	1	0	0	1
---	---	---	---	---	---	---	---

 byte 2

Description: The contents of the IY register are added to the contents of the specified register pair and the result is stored back in IY. rr may be any one of:

BC - 00	IY - 10
DE - 01	SP - 11

Data Flow:



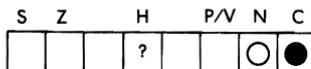
Timing: 4 M cycles; 15 T states: 7.5 usec @ 2 MHz

Addressing Mode: Implicit.

Byte Codes: rr: BC DE IY SP
 FD- [09 19 29 39]

PROGRAMMING THE Z80

Flags:



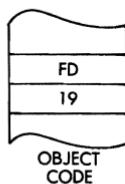
?



H is set by carry out of bit 11.
C is set by carry out of bit 15.

Example:

ADD IY, DE



Before:

D [6122] E

IY [3051]

After:

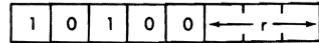
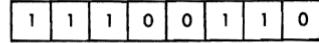
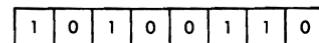
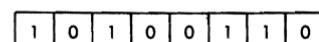
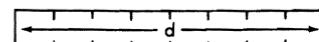
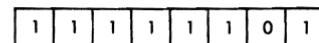
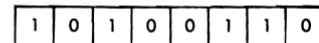
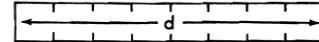
D [6122] E

IY [9173]

AND s

Logical AND accumulator with operand s.

Function: $A \leftarrow A \wedge s$ *Format:* *s: may be r, n, (HL), (IX + d), or (IY + d)*

r		
n		byte 1: E6
(HL)		A6
(IX + d)		byte 1: DD
		byte 2: A6
		byte 3: offset value
(IY + d)		byte 1: FD
		byte 2: A6
		byte 3: offset value

r may be any one of:

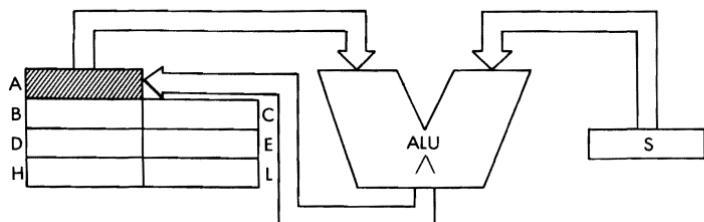
- | | |
|---------|---------|
| A - 111 | E - 011 |
| B - 000 | H - 100 |
| C - 001 | L - 101 |
| D - 010 | |

Description:

The accumulator and the specified operand are logically 'and'ed and the result is stored in the accumulator. s is defined in the description of the similar ADD instructions.

PROGRAMMING THE Z80

Data Flow:



Timing:

<i>s:</i>	<i>M cycles:</i>	<i>T states:</i>	<i>usec @ 2 MHz:</i>
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Addressing Mode: r: implicit; n: immediate; (HL): indirect; (IX + d), (IY + d): indexed.

Byte Codes:

AND r *r:* A B C D E H L

A7	A0	A1	A2	A3	A4	A5
----	----	----	----	----	----	----

Flags:

S Z H \oplus V N C

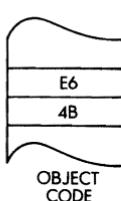
●	●		1	●	○	○
---	---	--	---	---	---	---

Example:

AND 4B

Before:

After:



A

36

A

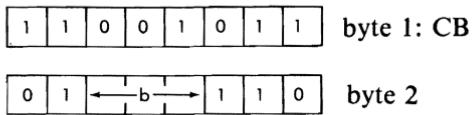
02

BIT b, (HL)

Test bit b of indirectly addressed memory location (HL)

Function: $Z \leftarrow \overline{(\text{HL})_b}$

Format:

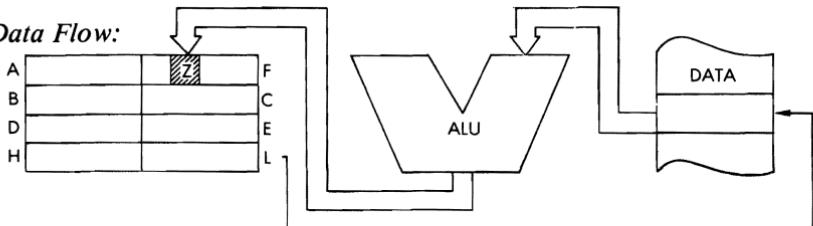


Description:

The specified bit of the memory location addressed by the contents of the HL register pair is tested and the Z flag is set according to the result. b may be any one of:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

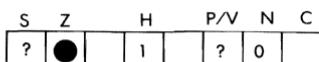
Data Flow:



Timing: 3 M cycles; 12 T states; 6 usec @ 2 MHz

Addressing Mode: Indirect.

Flags:



PROGRAMMING THE Z80

Byte Codes:

b:	0	1	2	3	4	5	6	7
CB-	46	4E	56	5E	66	6E	76	7E

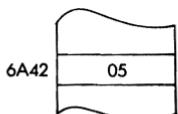
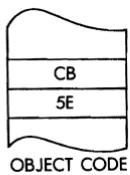
Example:

BIT 3, (HL)

Before:



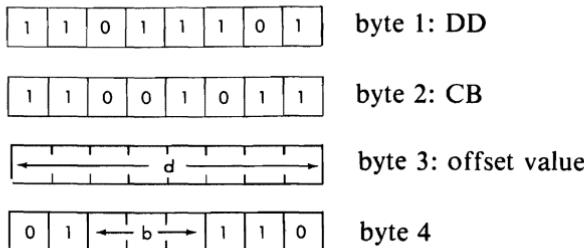
After:



BIT b, (IX + d) Test bit b of indexed addressed memory location $(IX + d)_b$

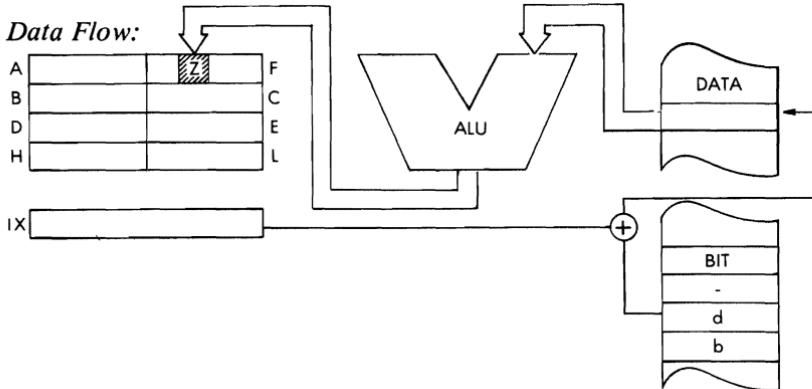
Function: $Z \leftarrow \overline{(IX + d)_b}$

Format:



Description: The specified bit of the memory location addressed by the contents of the IX register plus the given offset value is tested and the Z flag is set according to the result. b may be any one of:

0 – 000	5 – 101
1 – 001	6 – 110
2 – 010	7 – 111
3 – 011	
4 – 100	



PROGRAMMING THE Z80

Timing: 5 M cycles; 20 T states: 10 usec @ 2 MHz

Addressing Mode: Indexed.

Byte Codes:

b:	0	1	2	3	4	5	6	7
DD-CB-d-	46	4E	56	5E	66	6E	76	7E

Flags:

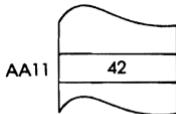
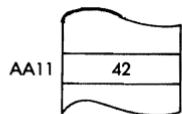
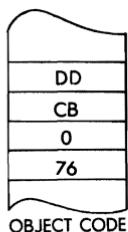
S	Z	H	P/V	N	C
?	●	1	?	0	

Example: BIT 6, (IX + 0)

Before:



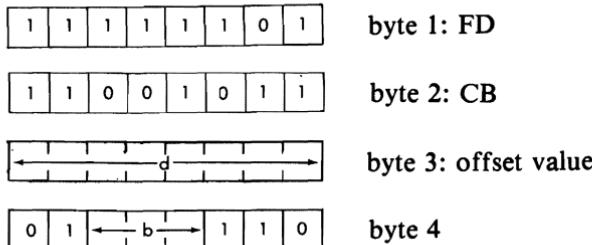
After:



BIT b, (IY + d) Test bit b of the indexed addressed memory location $(IY + d)$

Function: $Z \leftarrow \overline{(IY + d)_b}$

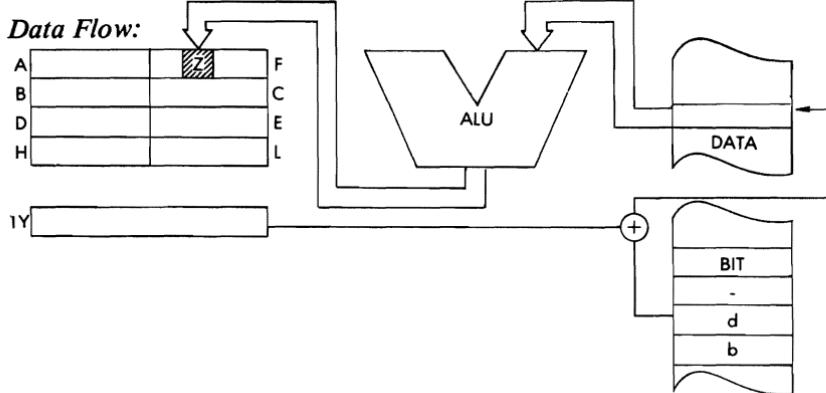
Format:



Description:

The specified bit of the memory location addressed by the contents of the IY register plus the given offset value is tested and the Z flag is set according to the result. b may be any one of:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111



PROGRAMMING THE Z80

Timing: 5 M cycles; 20 T states; 10 usec @ 2 MHz

Addressing Mode: Indexed.

Byte Codes: b : FD-CB-d-

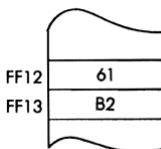
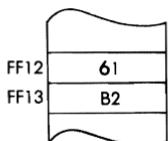
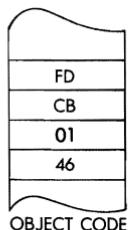
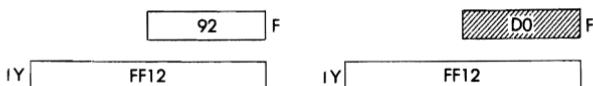
0	1	2	3	4	5	6	7
46	4E	56	5E	66	6E	76	7E

Flags:

S	Z	H	P/V	N	C
?	●	1	?	0	

Example: BIT 0, (IY + 1)

Before:



BIT b, r Test bit b of register r.

Function: $Z \leftarrow \overline{r_b}$

Format:

1	1	0	0	1	0	1	1
---	---	---	---	---	---	---	---

 byte 1: CB

0	1	$\xleftarrow{-} b \xrightarrow{-}$		$\xleftarrow{-} r \xrightarrow{-}$	
---	---	------------------------------------	--	------------------------------------	--

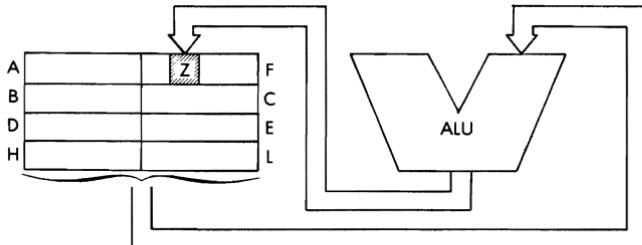
 byte 2

Description: The specified bit of the given register is tested and the zero flag is set according to the results. b and r may be any one of:

b:	0 – 000	4 – 100
	1 – 001	5 – 101
	2 – 010	6 – 110
	3 – 011	7 – 111

r:	A – 111	E – 011
	B – 000	H – 100
	C – 001	L – 101
	D – 010	

Data Flow:



Timing: 2 M cycles; 8 T states; 4 usec @ 2 MHz

Addressing Mode: Implicit.

PROGRAMMING THE Z80

Byte Codes:

CB-	b:	A	B	C	D	E	H	L
0	47	40	41	42	43	44	45	
1	4F	48	49	4A	4B	4C	4D	
2	57	50	51	52	53	54	55	
3	5F	58	59	5A	5B	5C	5D	
4	67	60	61	62	63	64	65	
5	6F	68	69	6A	6B	6C	6D	
6	77	70	71	72	73	74	75	
7	7F	78	79	7A	7B	7C	7D	

Flags:

S	Z	H	P/V	N	C
?	●	1	?	0	

Example:

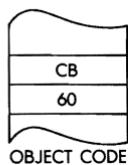
BIT 4, B

Before:

B 61

After:

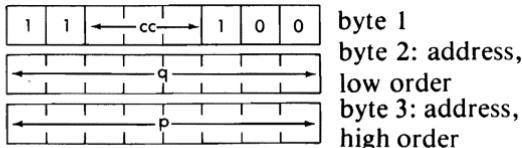
B 61 F 55 F



CALL cc, pq Call subroutine on condition.

Function: if cc true: $(SP - 1) \leftarrow PC_{high}$; $(SP - 2) \leftarrow PC_{low}$; $SP \leftarrow SP - 2$; $PC \leftarrow pq$
 If cc false: $PC \leftarrow PC + 3$

Format:



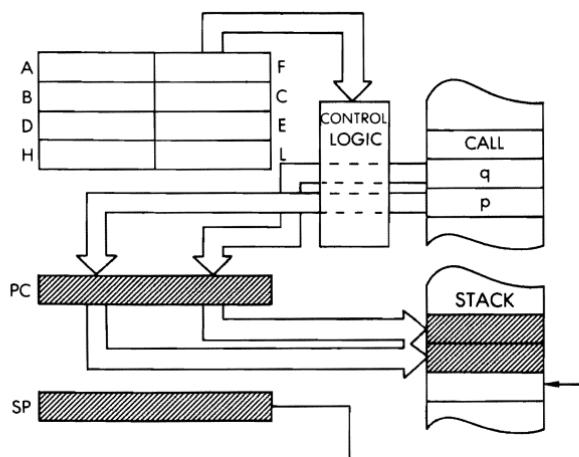
Description: If the condition is met, the contents of the program counter are pushed onto the stack as described for the PUSH instructions. Then, the contents of the memory location immediately following the opcode are loaded into the low order of the PC and the contents of the second memory location after the the opcode are loaded into the high order half of the PC. The next instruction fetched will be from this new address. If the condition is not met, the address pq is ignored and the following instruction is executed. cc may be any one of:

NZ – 000	PO – 100
Z – 001	PE – 101
NC – 010	P – 100
C – 011	M – 111

An RET instruction can be used at the end of the subroutine being called to restore the PC.

PROGRAMMING THE Z80

Data Flow:



Timing:

	M cycles:	T states:	usec @ 2 MHz
condition true:	5	17	8.5
condition not true:	3	10	5

Addressing Mode: Immediate.

Byte Codes:

CC:	NZ	Z	NC	C	P0	PE	P	M	
	C4	CC	D4	DC	E4	EC	F4	FC	-q-p

Flags:

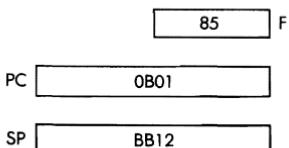
S	Z	H	P/V	N	C	(no effect)

THE Z80 INSTRUCTION SET

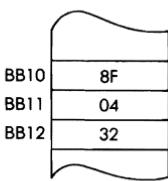
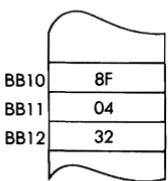
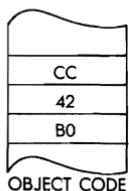
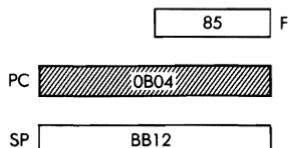
Example:

CALL Z, B042

Before:



After:

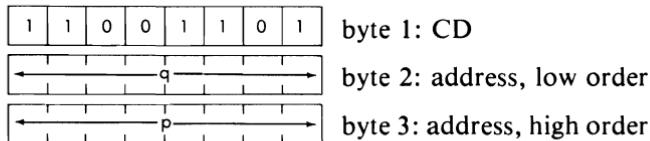


PROGRAMMING THE Z80

CALL pq Call subroutine at location pq.

Function: $(SP - 1) \leftarrow PC_{high}; (SP - 2) \leftarrow PC_{low}; SP \leftarrow SP - 2; PC \leftarrow pq$

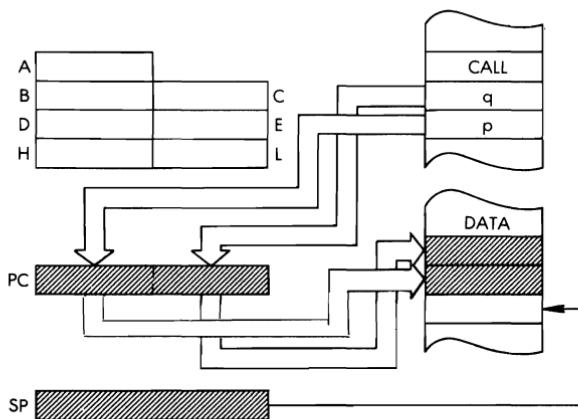
Format:



Description:

The contents of the program counter are pushed onto the stack as described for the PUSH instructions. The contents of the memory location immediately following the opcode are then loaded into the low order half of the PC and the contents of the second memory location after the opcode are loaded in the high order half of the PC. The next instruction will be fetched from this new address.

Data Flow:

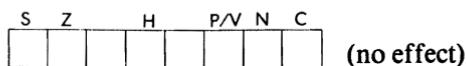


Timing: 5 M cycles; 17 T states: 8.5 usec @ 2 MHz

Addressing Mode: Immediate.

THE Z80 INSTRUCTION SET

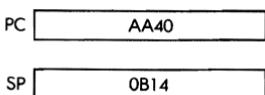
Flags:



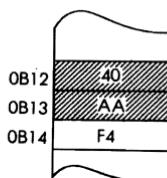
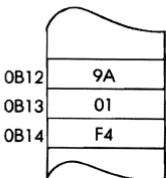
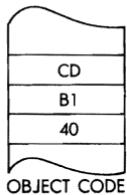
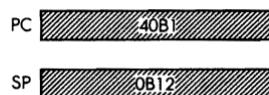
Example:

CALL 40B1

Before:



After:



PROGRAMMING THE Z80

CCF

Complement carry flag.

Function:

$$C \leftarrow \overline{C}$$

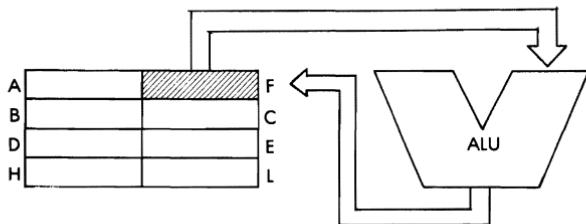
Format:

0	0	1	1	1	1	1	1	3F
---	---	---	---	---	---	---	---	----

Description:

The carry flag is complemented.

Data Flow:



Timing:

1 M cycle; 4 T states: 2 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:

S	Z	H	P/V	N	C
		?		○	●

CP s Compare operand s to accumulator.

Function: $A - s$

Format: s: may be r, n, (HL), (IX + d), or (IY + d).

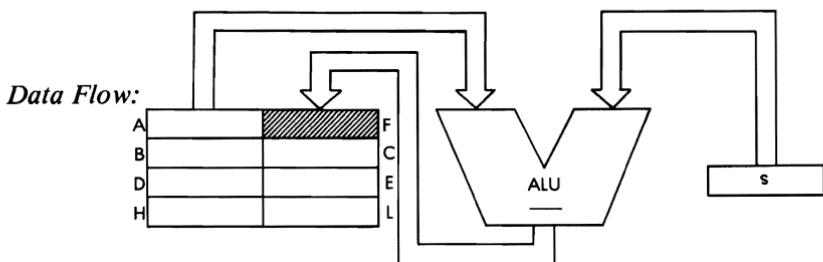
r		
n		FE
		byte 2: immediate data
(HL)		byte 1: BE
(IX + d)		byte 1: DD
		byte 2: BE
		byte 3: offset value
(IY + d)		byte 1: FD
		byte 2: BE
		byte 3: offset value

r may be any one of:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Description: The specified operand is subtracted from the accumulator, and the result is discarded. s is defined in the description of the similar ADD instructions.

PROGRAMMING THE Z80



Timing:

<i>s:</i>	<i>M cycles:</i>	<i>T states:</i>	<i>usec @ 2 MHz:</i>
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

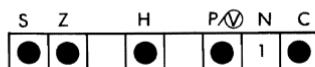
Addressing Modes: r: implicit; n: immediate; (HL): indirect;
 (IX + d), (IY + d): indexed

Byte Codes:

CP r:

r: A B C D E H L
 BF B8 B9 BA BB BC BD

Flags:



Example:

CP (HL)

Before:

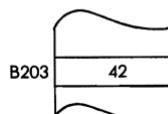
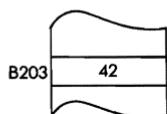
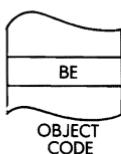
A [96] [36] F

H [B203] L

After:

A [96] [06] F

H [B203] L



CPD

Compare with decrement.

Function: $A - [HL]; HL \leftarrow HL - 1; BC \leftarrow BC - 1$ *Format:*

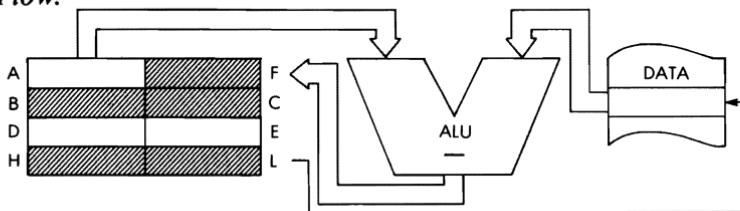
1	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: ED

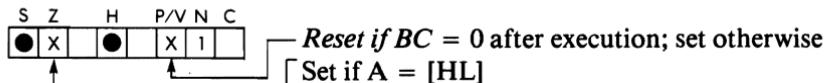
1	0	1	0	1	0	0	1
---	---	---	---	---	---	---	---

 byte 2: A9
Description:

The contents of the memory location addressed by the HL register pair are subtracted from the contents of the accumulator and the result is discarded. Then both the HL register pair and the BC register pair are decremented.

Data Flow:*Timing:*

4 M cycles; 16 T states: 8 usec @ 2 MHz

Addressing Mode: indirect.*Flags:*

PROGRAMMING THE Z80

Example:

CPD

Before:

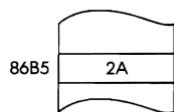
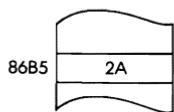
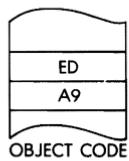
A	2A	06
B	3154	

H	86B5
---	------

After:

F	A	2A	46
C	B	3153	

L	H	86B4
---	---	------



CPDR

Block compare with decrement.

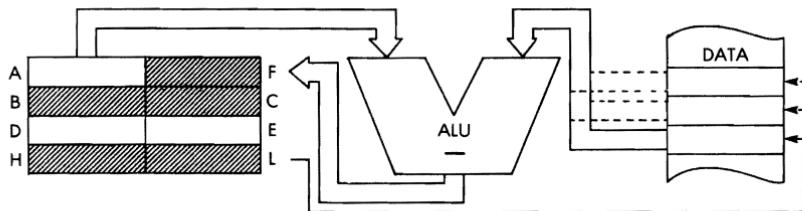
Function: $A - [HL]; HL \leftarrow HL - 1; BC \leftarrow BC - 1;$
 Repeat until $BC = 0$ or $A = [HL]$

Format:

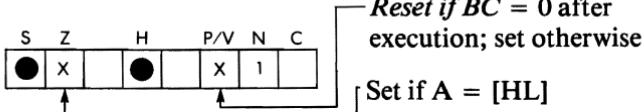
1	1	1	0	1	1	0	1
byte 1: ED							

1	0	1	1	1	0	0	1
byte 2: B9							

Description: The contents of the memory location addressed by the HL register pair are subtracted from the contents of the accumulator and the result is discarded. Then both the BC register pair and the HL register pair are decremented. If $BC \neq 0$ and $A \neq [HL]$, the program counter is decremented by two and the instruction is re-executed.

Data Flow:

Timing: $BC = 0$ or $A = [HL]$: 4 M cycles; 16 T states:
 8 usec @ 2 MHz
 $BC \neq 0$ and $A \neq [HL]$: 5 M cycles; 21 T states:
 10.5 usec @ 2 MHz

Flags:

PROGRAMMING THE Z80

Example:

CPDR

Before:

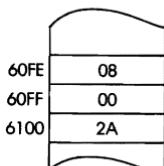
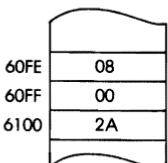
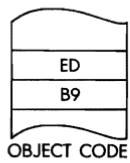
A	9A	00	F
B		0002	C

H	6100	L
---	------	---

After:

A	9A	82	F
B		0000	C

H	60FE	L
---	------	---



CPI

Compare with increment.

Function: $A - [HL]; HL \leftarrow HL + 1; BC \leftarrow BC - 1$

Format:

1	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---

byte 1: ED

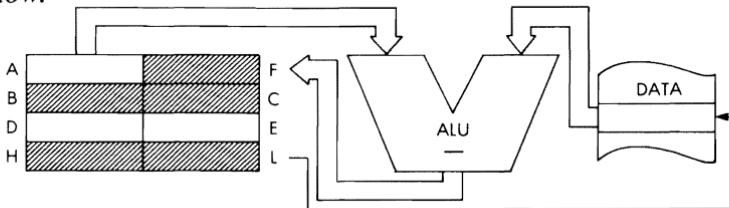
1	0	1	0	0	0	0	1
---	---	---	---	---	---	---	---

byte 2: A1

Description:

The contents of the memory location addressed by the HL register pair are subtracted from the contents of the accumulator and the result is discarded. The HL register pair is incremented and the BC register pair is decremented.

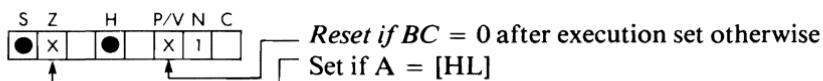
Data Flow:



Timing: 4 M cycles; 16 T states: 8 usec @ 2 MHz

Addressing Mode: indirect.

Flags:



PROGRAMMING THE Z80

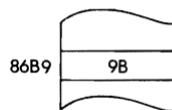
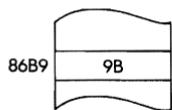
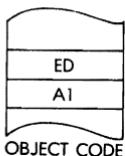
Example:

CPI

Before:

After:

A	09	00	F	A	09	16	F	
B	0510		C	B	050F		C	
H	86B9			L	H	86BA		L



CPIR

Block compare with increment.

Function:

$A \leftarrow [HL]; HL \leftarrow HL + 1; BC \leftarrow BC - 1;$
 Repeat until $BC = 0$ or $A = [HL]$

1	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---

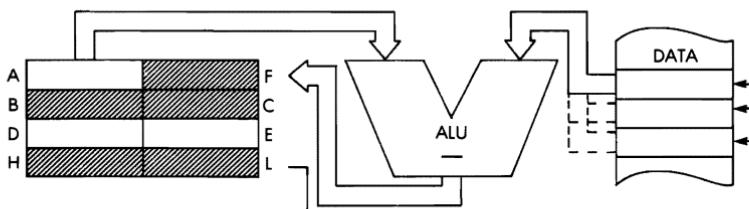
byte 1: ED

1	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

byte 2: B1

Description:

The contents of the memory location addressed by the HL register pair are subtracted from the contents of the accumulator and the result is discarded. Then the HL register pair is incremented and the BC register pair is decremented. If $BC \neq 0$ and $A \neq [HL]$, then the program counter is decremented by 2 and the instruction is re-executed.

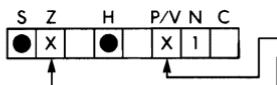
Data Flow:*Timing:*

$BC = 0$ or $A = [HL]$: 4 M cycles; 16 T states:
 8 usec @ 2 MHz
 $BC \neq 0$ and $A \neq [HL]$: 5 M cycles; 21 T states:
 10.5 usec @ 2 MHz

Addressing Mode: indirect.

PROGRAMMING THE Z80

Flags:



Reset if BC = 0 after execution; set otherwise
Set if A = [HL]

Example:

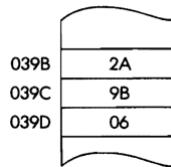
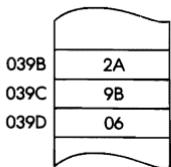
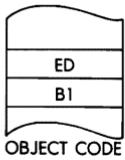
CPIR

Before:

A	9B	00
B	0051	
H	039B	L

After:

A	9B	4C
B	004F	
H	039D	L



CPL

Complement accumulator.

Function:

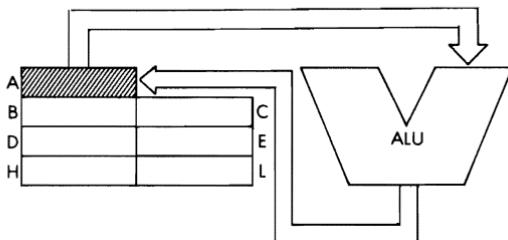
$$A \leftarrow \bar{A}$$

Format:

0	0	1	0	1	1	1	1	2F
---	---	---	---	---	---	---	---	----

Description:

The contents of the accumulator are complemented, or inverted, and the result is stored back in the accumulator (one's complement).

Data Flow:*Timing:*

1 M cycle; 4 T states; 2 usec @ 2 MHz

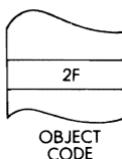
Addressing Mode: Implicit.*Flags:*

S	Z	H	P/V	N	C
		1		1	

Example:

CPL

Before:



A	3D
---	----

After:

A	C2
---	----

DAA

Decimal adjust accumulator.

Function:

See below.

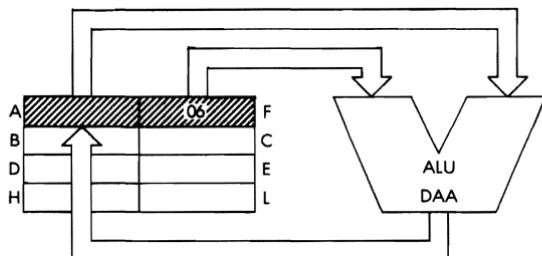
Format:

0	0	1	0	0	1	1	1
---	---	---	---	---	---	---	---

 27
Description:

The instruction conditionally adds “6” to the right and/or left nibble of the accumulator, based on the status register, for BCD conversion after arithmetic operations.

N	C	value of high nibble	H	value of low nibble	# added to A	C after execution
0 (ADD, ADC, INC)	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
	0	0-9	1	0-3	06	0
	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
	1	0-9	0	0-9	00	0
1 (SUB, SBC, DEC, NEG)	0	0-8	1	6-F	FA	0
	1	7-F	0	0-9	AO	1
	1	6-F	1	6-F	9A	1

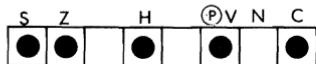
Data Flow:

THE Z80 INSTRUCTION SET

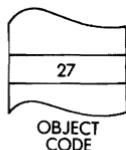
Timing: 1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:



Example: DAA



Before:



After:



PROGRAMMING THE Z80

DEC m

Decrement operand m.

Function: $m \leftarrow m - 1$

Format: m: may be r, (HL), (IX+d), (IY+d)

r	<table border="1"><tr><td>0</td><td>0</td><td colspan="2">r</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	r		1	0	1		
0	0	r		1	0	1				
(HL)	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	1	1	0	1	0	1	35
0	0	1	1	0	1	0	1			
(IX + d)	<table border="1"><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	1	1	0	1	1	1	0	1	byte 1: DD
1	1	0	1	1	1	0	1			
	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	1	1	0	1	0	1	byte 2: 35
0	0	1	1	0	1	0	1			
	<table border="1"><tr><td colspan="8">d</td></tr></table>	d								byte 3: offset value
d										
(IY + d)	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	1	1	1	1	1	1	0	1	byte 1: FD
1	1	1	1	1	1	0	1			
	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	1	1	0	1	0	1	byte 2: 35
0	0	1	1	0	1	0	1			
	<table border="1"><tr><td colspan="8">d</td></tr></table>	d								byte 3: offset value
d										

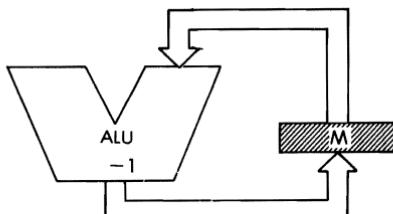
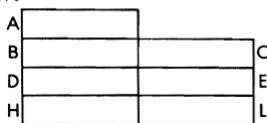
r may be any one of:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Description:

The contents of the location addressed by the specific operand are decremented and stored back at that location. m is defined in the description of the similar INC instructions.

Data Flow:



Timing:

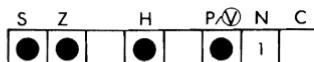
<i>m:</i>	<i>M cycles:</i>	<i>T states:</i>	<i>usec</i> @ 2 MHz:
r	1	4	2
(HL)	3	11	5.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Addressing Mode: r: implicit; (HL): indirect; (IX + d), (IY + d): indexed.

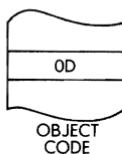
Byte Codes:

DEC r

r:	A	B	C	D	E	H	L
	3D	05	0D	15	1D	25	2D

Flags:*Example:*

DEC C

**Before:**

OF	C
----	---

After:

OE	C
----	---

PROGRAMMING THE Z80

DEC rr

Decrement register pair rr.

Function: $rr \leftarrow rr - 1$

Format:

0	0	r	r	1	0	1	1
---	---	---	---	---	---	---	---

Description:

The contents of the specified register pair are decremented and the result is stored back in the register pair. rr may be any one of:

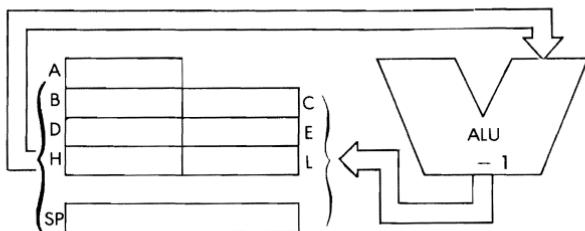
BC – 00

HL – 10

DE – 01

SP – 11

Data Flow:



Timing: 1 M cycle; 6 T states; 3 usec @ 2 MHz

Addressing Mode: Implicit.

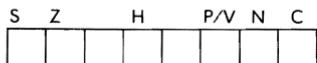
Byte Codes:

rr : BC DE HL SP

0B	1B	2B	3B
----	----	----	----

THE Z80 INSTRUCTION SET

Flags:



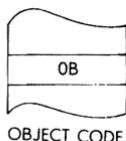
(no effect).

Example:

DEC BC

Before:

After:



DEC IX

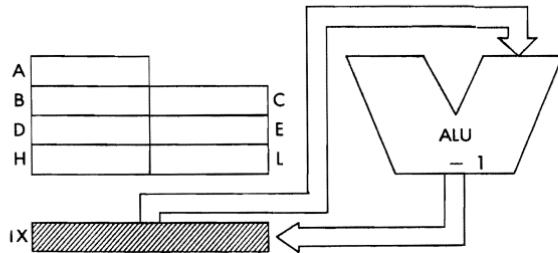
Decrement IX.

Function: $IX \leftarrow IX - 1$ *Format:*

1	1	0	1	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: DD

0	0	1	0	1	0	1	1
---	---	---	---	---	---	---	---

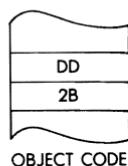
 byte 2: 2B
Description: The contents of the IX register are decremented and the result is stored back in IX.*Data Flow:**Timing:* 2 M cycles; 10 T states; 5 usec @ 2 MHz*Addressing Modes:* Implicit.*Flags:*

S	Z	H	P/V	N	C
[]	[]	[]	[]	[]	[]

 (no effect).
Example: DEC IX

Before:

After:

IX

6114

IX

6113

DEC IY

Decrement IY.

Function:

$$\text{IY} \leftarrow \text{IY} - 1$$

Format:

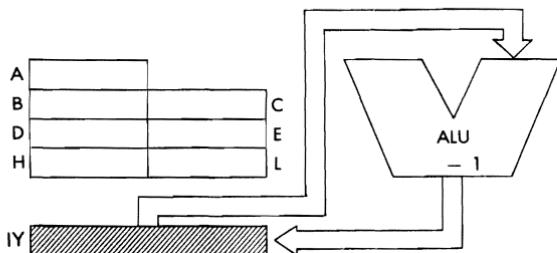
1	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: FD

0	0	1	0	1	0	1	1
---	---	---	---	---	---	---	---

 byte 2: 2B
Description:

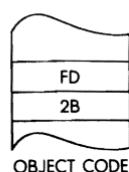
The contents of the IY register are decremented and the result is stored back in IY.

Data Flow:*Timing:* 2 M cycles; 10 T states; 5 usec @ 2 MHz*Addressing Mode:* Implicit.*Flags:*

S	Z	H	P/V	N	C

 (no effect).
Example:

DEC IY



Before:

IY

900F

After:

IY

900E

PROGRAMMING THE Z80

DI Disable interrupts.

Function: IFF $\leftarrow 0$

Format:

1	1	1	1	0	0	1	1	F3
---	---	---	---	---	---	---	---	----

Description: The interrupt flip-flops are reset, thereby disabling all maskable interrupts. It is reenabled by an EI instruction.

Timing: 1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.

Flags: S Z H P/V N C

--	--	--	--	--	--	--	--

 (no effect).

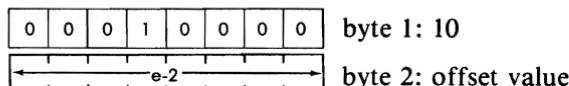
DJNZ e

Decrement B and jump e relative on no zero.

Function:

$$B \leftarrow B - 1; \text{ if } B \neq 0: PC \leftarrow PC + e$$

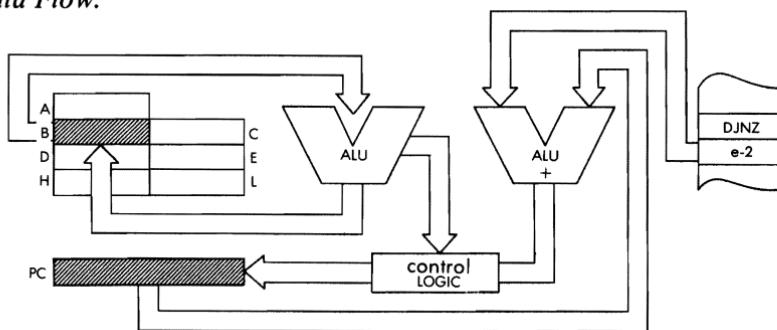
Format:



Description:

The B register is decremented. If the result is not zero, the immediate offset value is added to the program counter using two's complement arithmetic so as to enable both forward and backward jumps. The offset value is added to the value of $PC + 2$ (after the jump). As a result, the effective offset is -126 to +129 bytes. The assembler automatically subtracts from the source offset value to generate the hex code.

Data Flow:



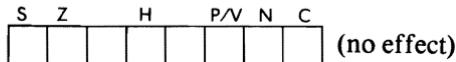
Timing:

$B \neq 0$: 3 M cycles; 13 T states; 6.5 usec @ 2 MHz.
 $B = 0$: 2 M cycles; 8 T states; 4 usec @ 2 MHz

Addressing Modes: Immediate.

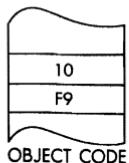
PROGRAMMING THE Z80

Flags:



Example: DJNZ \$ - 5 (\$ = current PC)

Before:



After:



EI

Enable interrupts.

Function: IFF \leftarrow 1

Format:

1	1	1	1	1	0	1	1	FB
---	---	---	---	---	---	---	---	----

Description: The interrupt flip-flops are set, thereby enabling maskable interrupts after the execution of the instruction following the EI instruction. In the meantime maskable interrupts are disabled.

Timing: 1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:

S	Z	H	P/V	N	C	(no effect).	
<input type="checkbox"/>							

Example: A usual sequence at the end of an interrupt routine is:
EI
RETI
The maskable interrupt is re-enabled following completion of RETI.

PROGRAMMING THE Z80

EX AF, AF'

Exchange accumulator and flags with alternate registers.

Function:

AF \leftrightarrow AF'

Format:

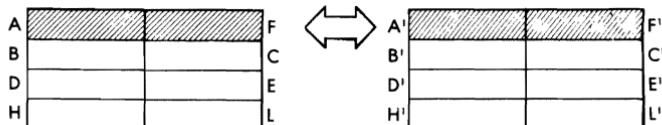
0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---

 08

Description:

The contents of the accumulator and status register are exchanged with the contents of the alternate accumulator and status register.

Data Flow:



Timing:

1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:

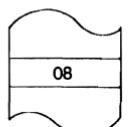
S	Z	H	P/V	N	C
●	●	●	●	●	●

Example:

EX AF, AF'

Before:

After:



OBJECT CODE

A	04	81	F	A	90	3A	F
A'	90	3A	F'	A'	04	81	F'

EX DE, HL Exchange the HL and DE registers.

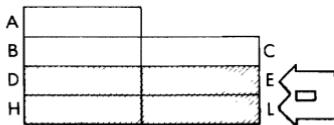
Function: $DE \longleftrightarrow HL$

Format:

1	1	1	0	1	0	1	1	EB
---	---	---	---	---	---	---	---	----

Description: The contents of the register pairs DE and HL are exchanged.

Data Flow:



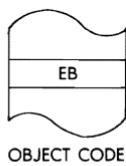
Timing: 1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:

S	Z	H	P/V	N	C	(no effect).

Example: EX DE, HL



Before:

D	A4E6	E
H	9604	L

After:

D	9604	E
H	A4E6	L

EX (SP), HL Exchange HL with top of stack.

Function: $(SP) \leftrightarrow L; (SP + 1) \leftarrow H$

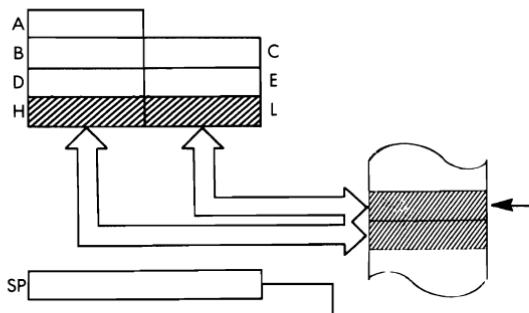
Format:

1	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---

 E3

Description: The contents of the L register are exchanged with the contents of the memory location addressed by the stack pointer. The contents of the H register are exchanged with the contents of the memory location immediately following the one addressed by the stack pointer.

Data Flow:



Timing: 5 M cycles; 19 T states; 9.5 usec @ 2 MHz

Addressing Mode: Indirect.

Flags:

S	Z	H	P/V	N	C

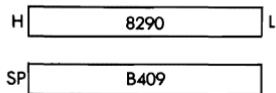
(no effect).

THE Z80 INSTRUCTION SET

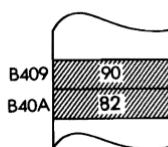
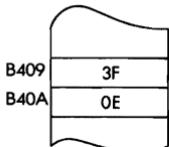
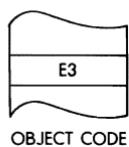
Example:

EX (SP), HL

Before:



After



{

PROGRAMMING THE Z80

EX (SP), IX Exchange IX with top of stack.

Function: $(SP) \leftrightarrow IX_{low}; (SP + 1) \leftrightarrow IX_{high}$

Format:

1	1	0	1	1	1	0	1
---	---	---	---	---	---	---	---

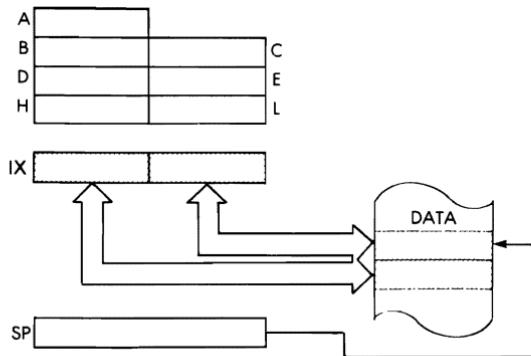
 byte 1: DD

1	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---

 byte 2: E3

Description: The contents of the low order of the IX register are exchanged with the contents of the memory location addressed by the stack pointer. The contents of the high order of the IX register are exchanged with the contents of the memory location immediately following the one addressed by the stack pointer.

Data Flow:



Timing: 6 M cycles; 23 T states; 11.5 usec @ 2 MHz

Addressing Mode: Indirect.

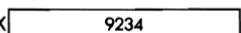
Flags:

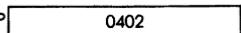
S	Z	H	P/V	N	C

 (no effect).

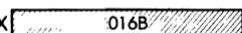
Example: EX (SP), IX

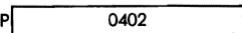
Before:

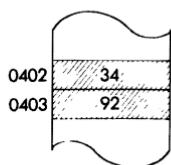
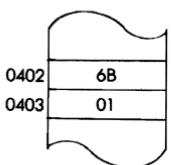
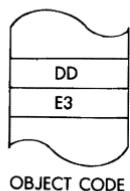
IX  9234

SP  0402

After:

IX  016B

SP  0402



EX (SP), IY Exchange IY with top of stack.

Function: $(SP) \leftrightarrow IY_{low}; (SP + 1) \leftrightarrow IY_{high}$

Format:

1	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

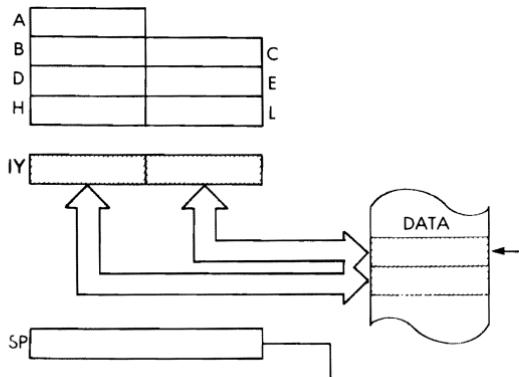
byte 1: FD

1	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---

byte 2: E3

Description: The contents of the low order of the IY register are exchanged with the contents of the memory location addressed by the stack pointer. The contents of the high order of the IY register are exchanged with the contents of the memory location immediately following the one addressed by the stack pointer.

Data Flow:



Timing: 6 M cycles; 23 T states; 11.5 usec @ 2 MHz

Addressing Mode: Indirect.

Flags:

S	Z	H	P/V	N	C

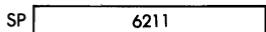
(no effect).

THE Z80 INSTRUCTION SET

Example: EX (SP), IY

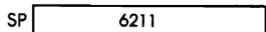
Before:

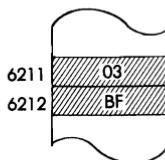
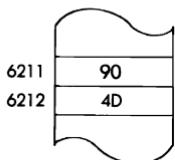
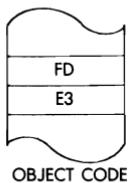
IY 

SP 

After:

IY 

SP 



PROGRAMMING THE Z80

EXX

Exchange alternate registers.

Function:

$BC \leftrightarrow BC'; DE \leftrightarrow DE'; HL \leftrightarrow HL'$

Format:

1	1	0	1	1	0	0	1
---	---	---	---	---	---	---	---

 D9

Description:

The contents of the general purpose registers are exchanged with the contents of the corresponding alternate registers.

Data Flow:



Timing:

1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:

S	Z	H	P/V	N	C

 (no effect).

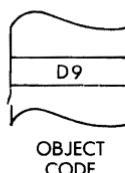
Example:

EXX

Before:

After:

A	04	2B	F	04	2B	F
B	39	26	C	8C	00	C
D	54	02	E	93	D0	E
H	F1	D0	L	4F	E3	L



A'	3F	2A	F'	3F	2A	F'
B'	8C	00	C'	39	26	C'
D'	93	D0	E'	54	02	E'
H'	4F	E3	L'	F1	D0	L'

HALT*Halt CPU.**Function:*

CPU suspended.

Format:

0	1	1	1	0	1	1	0	76
---	---	---	---	---	---	---	---	----

Description:

CPU suspends operation and executes NOP's so as to continue memory refresh cycles, until interrupt or reset is received.

Timing:

1 M cycle; 4 T states; 2 usec @ 2 MHz + infinite Nop's.

Addressing Mode: Implicit.*Flags:*

S	Z	H	P/V	N	C	
						(no effect).

IM 0 Set interrupt mode 0 condition.

Function: Internal interrupt control.

Format:

1	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: ED

0	1	0	0	0	1	1	0
---	---	---	---	---	---	---	---

 byte 2: 46

Description: Sets interrupt mode 0. In this condition, the interrupting device may insert one instruction onto the data bus for execution, the first byte of which must occur during the interrupt acknowledge cycle.

Timing: 2 M cycle; 8 T states; 4 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:

S	Z		H		P/V	N	C
---	---	--	---	--	-----	---	---

 (no effect).

IM 1

Set interrupt mode 1 condition.

Function:

Internal interrupt control.

Format:

1	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---

byte 1: ED

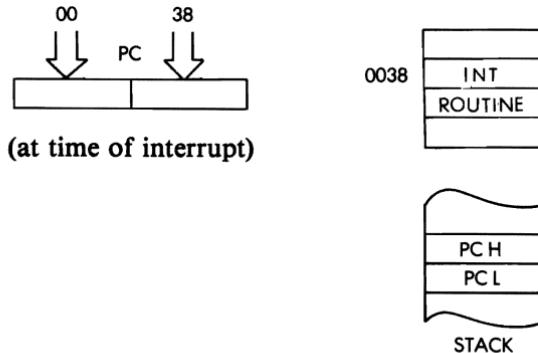
0	1	0	1	0	1	1	0
---	---	---	---	---	---	---	---

byte 2: 56

Description:

Sets interrupt mode 1. A RST 0038H instruction will be executed when an interrupt occurs.

Data Flow:



Timing: 2 M cycles; 8 T states; 4 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:

S	Z	H	P/V	N	C
[]	[]	[]	[]	[]	[]

(no effect).

IM 2

Set interrupt mode 2 condition.

Function:

Internal interrupt control.

Format:

1	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: ED

0	1	0	1	1	1	1	0
---	---	---	---	---	---	---	---

 byte 2: 5E

Description:

Set interrupt mode 2. When an interrupt occurs, one byte of data must be provided by the peripheral which is used as the low order of an address. The high order of this vector address is taken from the contents of the I register. This points to a second address stored in memory, which is loaded into the program counter and begins execution.

Timing:

2 M cycles; 8 T states; 4 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:

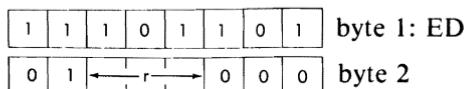
S	Z	H	P/V	N	C
---	---	---	-----	---	---

 (no effect)

IN r, (C) Load register r from port(C)

Function: $r \leftarrow (C)$

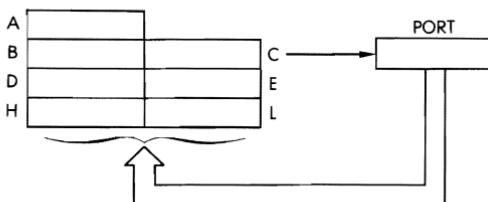
Format:



Description: The peripheral device addressed by the contents of the C register is read and the result is loaded into the specified register.

C provides bits A0 to A7 of the address bus.
B provides bits A8 to A15.

Data Flow:



r may be any one of:

- | | |
|---------|---------|
| A – 111 | E – 011 |
| B – 000 | H – 100 |
| C – 001 | L – 101 |
| D – 010 | |

Timing: 3 M cycles; 12 T states; 6 usec @ 2 MHz

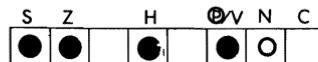
Addressing Mode: External.

Byte Codes:

r:	A	B	C	D	E	H	L
ED	78	40	48	50	58	60	68

PROGRAMMING THE Z80

Flags:

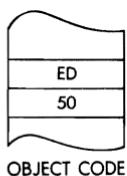


It is important to note that IN A,(N) does not have any effect on the flags, while IN r, (C) does.

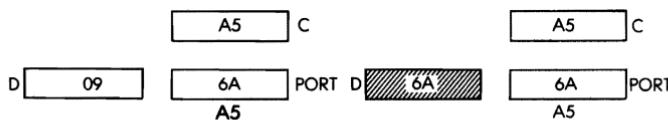
Example:

IN D, (C)

Before:



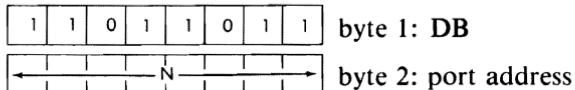
After:



IN A, (N) Load accumulator from input port N.

Function: $\mathbf{A} \leftarrow (\mathbf{N})$

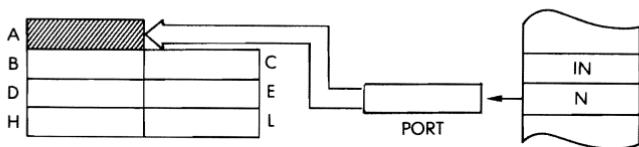
Format:



Description: The peripheral device N is read and the result is loaded into the accumulator.

The literal N is placed on lines A0 to A7 of the address bus. A supplies bits A8 to A15.

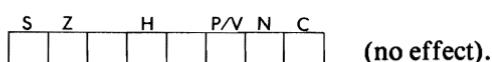
Data Flow:



Timing: 3 M cycles; 11 T states; 5.5 usec @ 2 MHz

Addressing Mode: External.

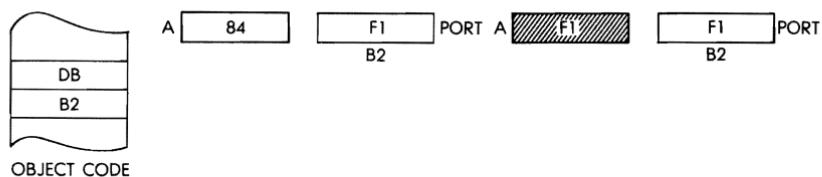
Flags:



Example: IN A. (B2)

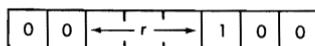
Before:

After:



INC r*Increment register r.***Function:**

$$r \leftarrow r + 1$$

Format:**Description:**

The contents of the specified register are incremented. r may be any one of:

A - 111

E - 011

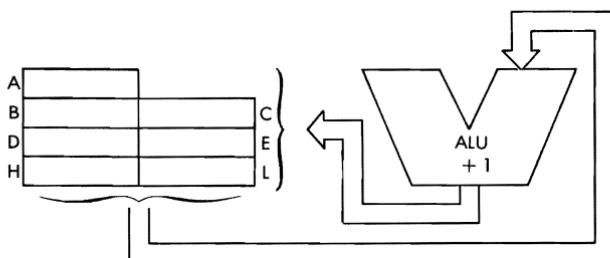
B - 000

H - 100

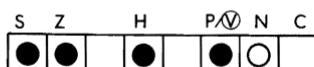
C - 001

L - 101

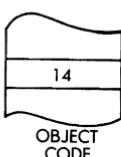
D - 010

Data Flow:**Timing:** 1 M cycle; 4 T states; 2 usec @ 2 MHz**Addressing Mode:** Implicit.**Byte Codes:**

r:	A	B	C	D	E	H	L
	3C	04	0C	14	1C	24	2C

Flags:**Example:**

INC D



Before:
D [] 06

After:
D [] 07

INC rr

Increment register pair rr.

Function: $rr \leftarrow rr + 1$

Format:

0	0	r	r	0	0	1	1
---	---	---	---	---	---	---	---

Description: The contents of the specified register pair are incremented and the result is stored back in the register pair. rr may be any one of:

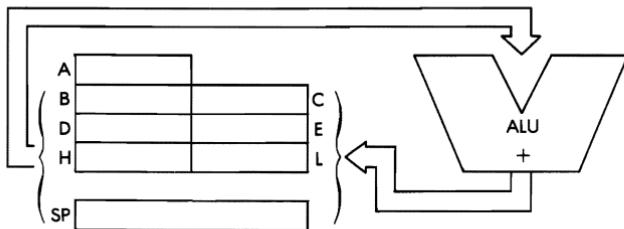
BC - 00

HL - 10

DE - 01

SP - 11

Data Flow:



Timing: 1 M cycle; 6 T states; 3 usec @ 2 MHz

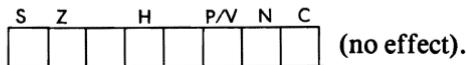
Addressing Mode: Implicit.

Byte Codes:

rr:	BC	DE	HL	SP
	03	13	23	33

PROGRAMMING THE Z80

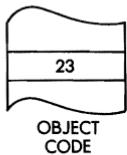
Flags:



Example:

INC HL

Before:



After:



INC (HL)

Increment indirectly addressed memory location (HL).

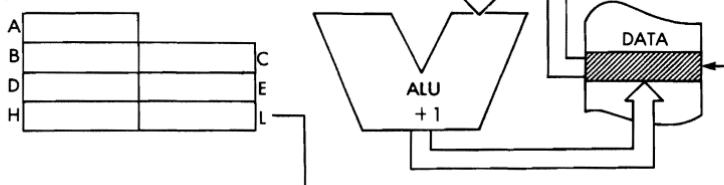
Function: $(HL) \leftarrow (HL) + 1$

Format:

0	0	1	1	0	1	0	0	34
---	---	---	---	---	---	---	---	----

Description: The contents of the memory location addressed by the HL register pair are incremented and stored back at that location.

Data Flow:



Timing: 3 M cycles; 11 T states; 5.5 usec @ 2 MHz

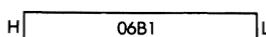
Addressing Mode: Indirect.

Flags:

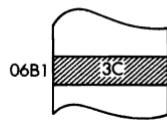
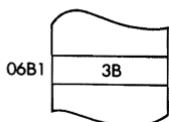
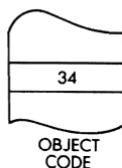
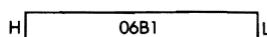
S	Z	H	P/V	N	C
●	●	□	●	●	○

Example: INC (HL)

Before:



After:



PROGRAMMING THE Z80

INC (IX + d) Increment indexed addressed memory location (IX + d).

Function: $(IX + d) \leftarrow (IX + d) + 1$

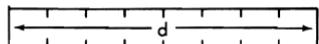
Format:

1	1	0	1	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: DD

0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

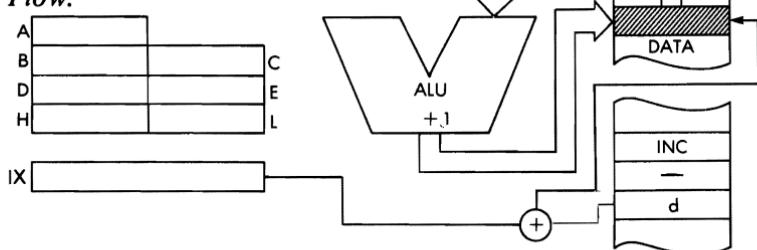
 byte 2: 34

 byte 3: offset value

Description:

The contents of the memory location addressed by the contents of the IX register plus the given offset value are incremented and stored back at that location.

Data Flow:



Timing: 6 M cycles; 23 T states; 11.5 usec @ 2 MHz

Addressing Mode: Indexed.

Flags:

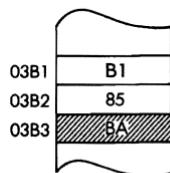
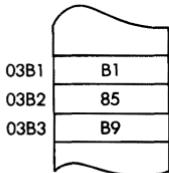
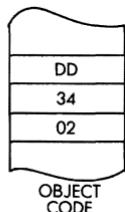
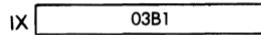
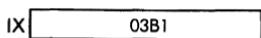
S	Z		H	P/V	N	C
●	●		●		●	○

THE Z80 INSTRUCTION SET

Example: INC (IX + 2)

Before:

After:



PROGRAMMING THE Z80

INC (IY + d) Increment indexed addressed memory location (IY + d).

Function: $(IY + d) \leftarrow (IY + d) + 1$

Format:

1	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: FD

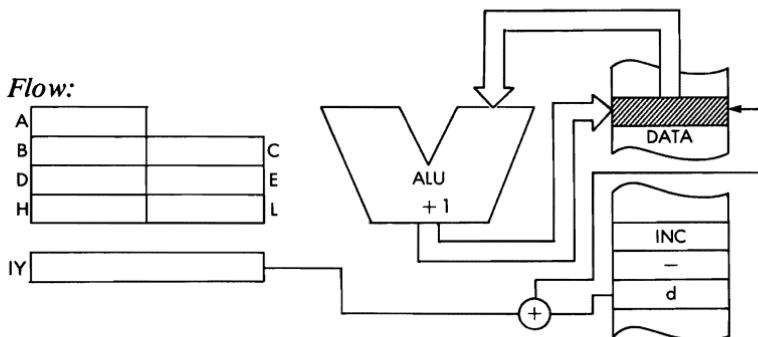
0	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

 byte 2: 34

byte 3: offset value

Description: The contents of the memory location addressed by the contents of the IY register plus the given offset value are incremented and stored back at that location.

Data Flow:



Timing: 6 M cycles; 23 T states; 11.5 usec @ 2 MHz

Addressing Mode: Indexed.

Flags:

S	Z	H	P/V	N	C
●	●	●	●	○	

Example: INC (IY + 0)

Before:

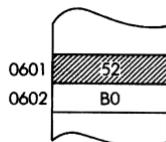
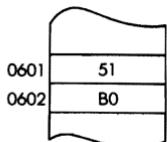
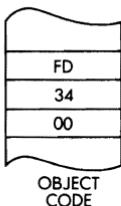
IY

0601

After:

IY

0601



PROGRAMMING THE Z80

INC IX

Increment IX.

Function: $IX \leftarrow IX + 1$

Format:

1	1	0	1	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: DD

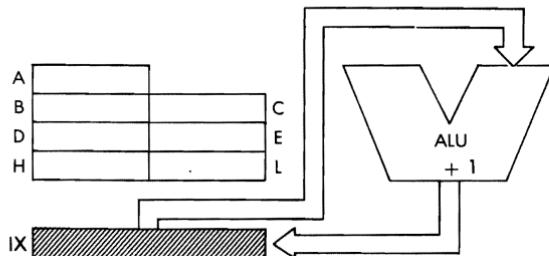
0	0	1	0	0	0	1	1
---	---	---	---	---	---	---	---

 byte 2: 23

Description:

The contents of the IX register are incremented and the result is stored back in IX.

Data Flow:



Timing: 2 M cycles; 10 T states; 5 usec @ 2 MHz

Addressing Mode: Implicit.

Flags:

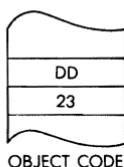
S	Z	H	P/V	N	C
---	---	---	-----	---	---

 (no effect).

Example: INC IX

Before:

After:



IX

B1B0

IX

B1B1

INC IY

Increment IY

Function: $IY \leftarrow IY + 1$ *Format:*

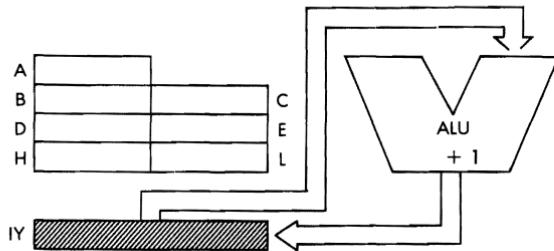
1	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

 byte 1: FD

0	0	1	0	0	0	1	1
---	---	---	---	---	---	---	---

 byte 2: 23
Description:

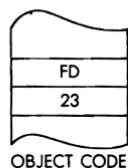
The contents of the IY register are incremented and the result is stored back in IY.

Data Flow:*Timing:* 2 M cycles; 10 T states; 5 usec @ 2 MHz*Addressing Mode:* Implicit.*Flags:*

S	Z	H	P/V	N	C

 (no effect).
Example:

INC IY

*Before:**After:*IY

36B1

IY

36B2

PROGRAMMING THE Z80

IND

Input with decrement.

Function: $(HL) \leftarrow (C); B \leftarrow B - 1; HL \leftarrow HL - 1$

Format:

1	1	1	0	1	1	0	1
1	0	1	0	1	0	1	0

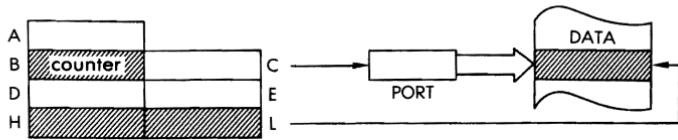
byte 1: ED

byte 2: AA

Description:

The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair. The B register and the HL register pair are then each decremented.

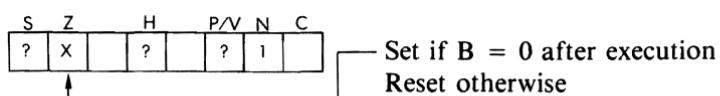
Data Flow:



Timing: 4 M cycles; 16 T states; 8 usec @ 2 MHz

Addressing Mode: External.

Flags:

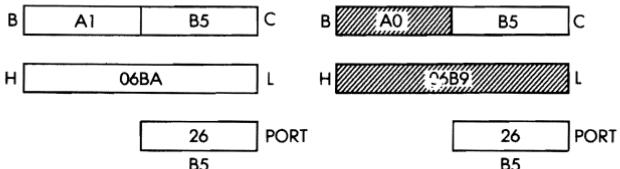


THE Z80 INSTRUCTION SET

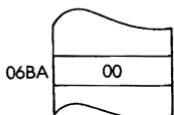
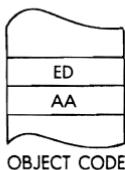
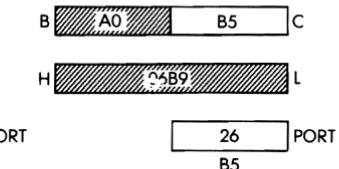
Example:

IND

Before:



After:



INDR

Block input with decrement.

Function: $(HL) \leftarrow (C); B \leftarrow B - 1; HL \leftarrow HL - 1$
 Repeat until $B = 0$

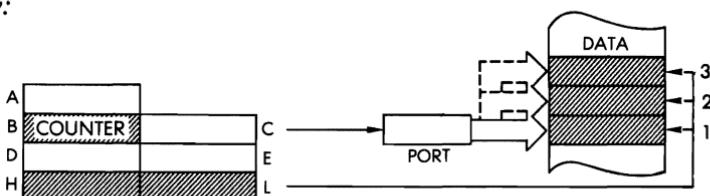
Format:

1	1	1	0	1	1	0	1	byte 1: ED
1	0	1	1	1	0	1	0	byte 2: BA

Description:

The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair. Then the B register and the HL register pair are decremented. If B is not zero, the program counter is decremented by 2 and the instruction is re-executed.

Data Flow:



Timing:

$B = 0:4$ M cycles; 16 T states; 8 usec @ 2 MHz.
 $B \neq 0:5$ M cycles; 21 T states; 10.5 usec @ 2 MHz.

Addressing Mode: External

Flags:

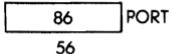
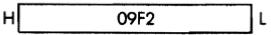
S	Z	H	P/V	N	C
?	1	?	?	1	

THE Z80 INSTRUCTION SET

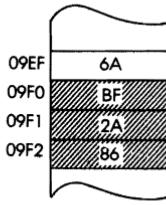
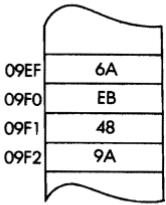
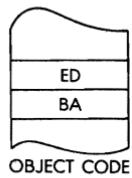
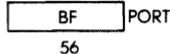
Example:

INDR

Before:



After:



PROGRAMMING THE Z80

INI

Input with increment.

Function: $(HL) \leftarrow (C); B \leftarrow B - 1; HL \leftarrow HL + 1$

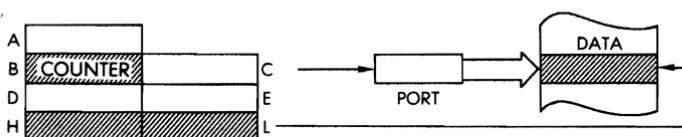
Format:

1	1	1	0	1	1	0	1
byte 1: ED							
1	0	1	0	0	0	1	0
byte 2: A2							

Description: The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair. The B register is decremented and the HL register pair is incremented.

The contents of C are placed on the low half of the address bus. The contents of B are placed on the high half. I/O selection is generally made by C, i.e., by A0 to A7. B is a byte counter.

Data Flow:



Timing: 4 M cycles; 16 T states; 8 usec @ 2 MHz

Addressing Mode: External.

Flags:

S	Z		H	P/V	N	C
?	X		?	?	1	

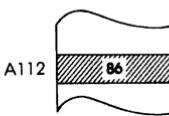
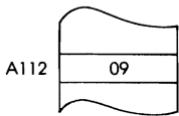
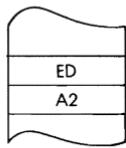
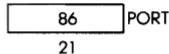
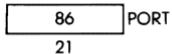
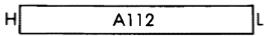
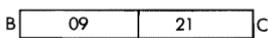
Z is set if B = 0 after execution,
Reset otherwise

Example:

INI

Before:

After:



OBJECT CODE

PROGRAMMING THE Z80

INIR

Block input with increment.

Function: $(HL) \leftarrow (C); B \leftarrow B - 1; HL \leftarrow HL + 1;$ Repeat until $B = 0$

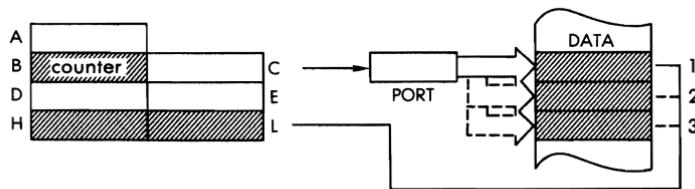
Format:

1	1	1	0	1	1	0	1
1	0	1	1	0	0	1	0

byte 1: ED
byte 2: B2

Description: The peripheral device addressed by the C register is read and the result is loaded into the memory location addressed by the HL register pair. The B register is decremented and the HL register pair is incremented. If B is not zero, the program counter is decremented by 2 and the instruction is re-executed.

Data Flow:



Timing: $B = 0: 4 M$ cycles; 16 T states; 8 used @ 2 MHz.
 $B \neq 0: 5 M$ cycles; 21 T states; 10.5 usec @ 2 MHz.

Addressing Mode: External.

Flags:

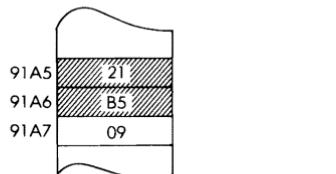
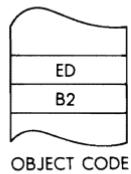
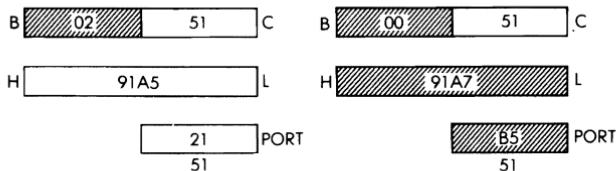
S	Z	H	P/V	N	C
?	1	?	?	1	

THE Z80 INSTRUCTION SET

Example:

INIR

Before:

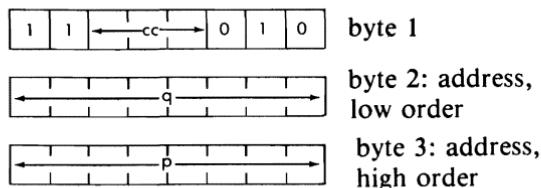


PROGRAMMING THE Z80

JP cc, pq Jump on condition to location pq.

Function: if cc true: $PC \leftarrow pq$

Format:

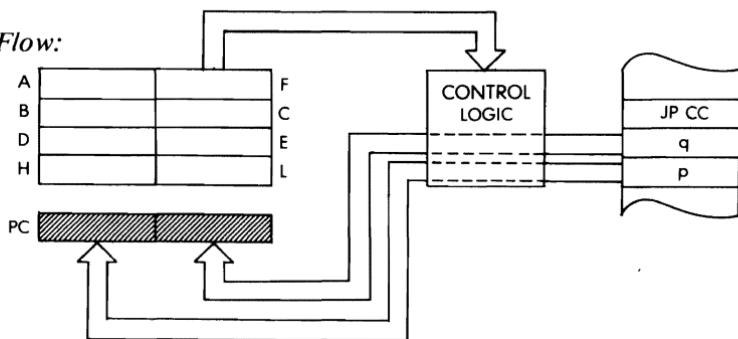


Description:

If the specified condition is true, the two-byte address immediately following the opcode will be loaded into the program counter with the first byte following the opcode being loaded into the low order of the PC. If the condition is not met, the address is ignored. cc may be any one of:

NZ - 000	no zero
Z - 001	zero
NC - 010	no carry
C - 011	carry
PO - 100	parity odd
PE - 101	parity even
P - 110	plus
M - 111	minus

Data Flow:



THE Z80 INSTRUCTION SET

Timing: 3 M cycles; 10 T states; 5 usec @ 2 MHz

Addressing Mode: Immediate.

Byte Codes:

C C	NZ	Z	NC	C	P0	PE	P	M
C2	CA	D2	DA	E2	EA	F2	FA	

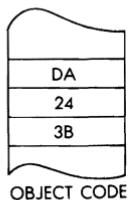
Flags:

S	Z	H	P/V	N	C

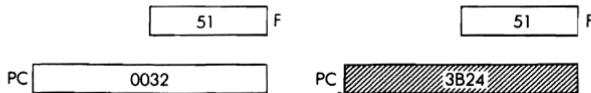
(no effect)

Example: JP C, 3B24

Before:



After:



PROGRAMMING THE Z80

JP pq

Jump to location pq.

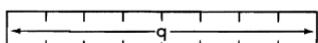
Function:

$$PC \leftarrow pq$$

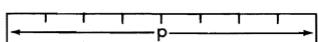
Format:

1	1	0	0	0	0	1	1
---	---	---	---	---	---	---	---

byte 1: C3



byte 2: address,
low order

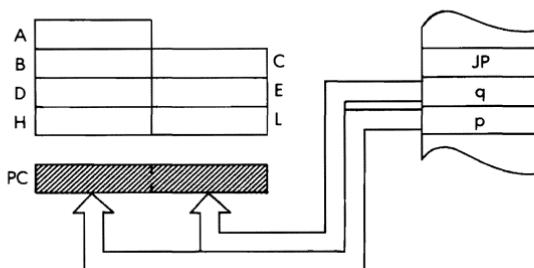


byte 3: address,
high order

Description:

The contents of the memory location immediately following the opcode are loaded into the low order half of the program counter and the contents of the second memory location immediately following the opcode are loaded into the high order of the program counter. The next instruction will be fetched from this new address.

Data Flow:



Timing: 3 M cycles; 10 T states; 5 usec @ 2 MHz

Addressing Mode: Immediate.

Flags:

S	Z	H	P/V	N	C	(No effect)

Example:

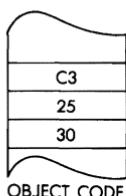
JP 3025

Before:

PC 5520

After:

PC 3025



JP (HL) Jump to HL.

Function: $PC \leftarrow HL$

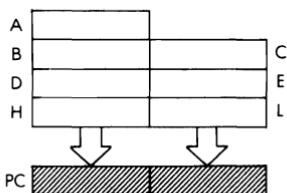
Format:

1	1	1	0	1	0	0	1
---	---	---	---	---	---	---	---

 E9

Description: The contents of the HL register pair are loaded into the program counter. The next instruction is fetched from this new address.

Data Flow:



Timing: 1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.

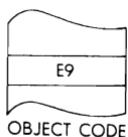
Flags:

S	Z	H	P/V	N	C
---	---	---	-----	---	---

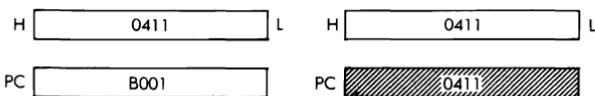
 (no effect).

Example: JP (HL)

Before:



After:



PROGRAMMING THE Z80

JP (IX) Jump to IX.

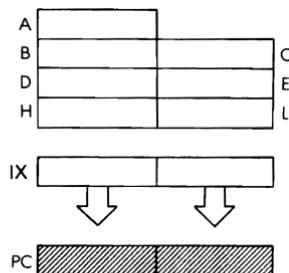
Function: $PC \leftarrow IX$

Format:

<table border="1"><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	1	1	0	1	1	1	0	1	byte 1: DD
1	1	0	1	1	1	0	1		
<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	1	1	1	0	1	0	0	1	byte 2: E9
1	1	1	0	1	0	0	1		

Description: The contents of the IX register are loaded into the program counter. The next instruction is fetched from this new address.

Data Flow:



Timing: 2 M cycles; 8 T states; 4 usec @ 2 MHz

Addressing Mode: Implicit.

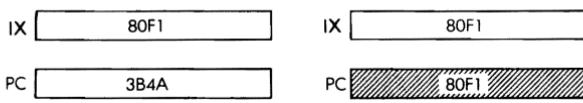
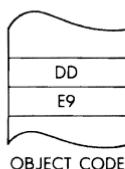
Flags:

S	Z	H	P/V	N	C	(no effect).

Example: JP (IX)

Before:

After:



JP (IY)

Jump to IY.

Function: $PC \leftarrow IY$ *Format:*

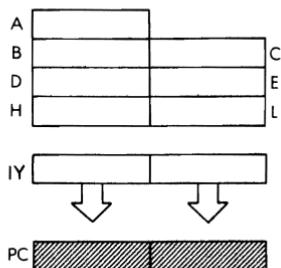
1	1	1	1	1	1	0	1
1	1	1	0	1	0	0	1

byte 1: FD

byte 2: E9

Description:

The contents of the IY register are moved into the program counter. The next instruction will be fetched from this new address.

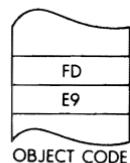
Data Flow:*Timing:* 2 M cycles; 8 T states; 4 usec @ 2 MHz*Addressing Mode:* Implicit.*Flags:*

S	Z	H	P/V	N	C

(no effect).

Example:

JP (IY)

Before:**After:**

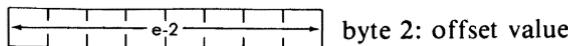
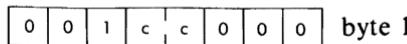
IY	AA4B	IY	AA4B
PC	E410	PC	AA4B

PROGRAMMING THE Z80

JR cc, e Jump e relative on condition.

Function: if cc true, $PC \leftarrow PC + e$

Format:



Description:

If the specified condition is met, the given offset value is added to the program counter using two's complement arithmetic so as to enable both forward and backward jumps. The offset value is added to the value of $PC + 2$ (after the jump). As a result, the effective offset is -126 to +129 bytes. The assembler automatically subtracts 2 from the source offset value to generate the hex code. If the condition is not met, the offset value is ignored and instruction execution continues in sequence. cc may any one of:

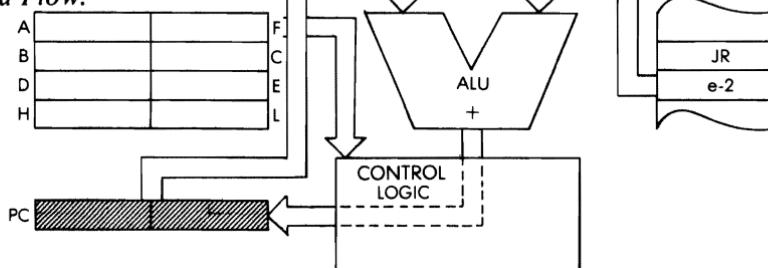
NZ – 00

Z – 01

NC – 10

C – 11

Data Flow:



Timing:

	<i>M cycles:</i>	<i>T states:</i>	<i>usec @ 2 MHz:</i>
condition met:	3	12	6
condition not met:	2	7	3.5

THE Z80 INSTRUCTION SET

Addressing Mode: Relative.

* NOT Z80

Byte Codes:

cc:	NZ	Z	NC	C
	20	28	30	38

Flags:

S	Z	H	P/V	N	C

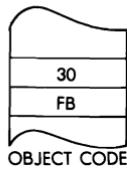
Example:

JR NC, \$ - 3

\$ = current PC

Before:

After:



PROGRAMMING THE Z80

JR e Jump e relative.

Function: $PC \leftarrow PC + e$

Format:

0	0	0	1	1	0	0	0
---	---	---	---	---	---	---	---

 byte 1: 18

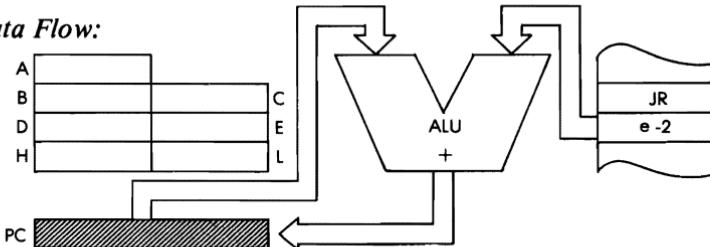
e-2							
-----	--	--	--	--	--	--	--

 byte 2: offset value

Description:

The given offset value is added to the program counter using two's complement arithmetic so as to enable both forward and backward jumps. The offset value is added to the value of $PC + 2$ (after the jump). As a result, the effective offset is -126 to + 129 bytes. The assembler automatically subtracts 2 from the source offset value to generate the hex code.

Data Flow:



Timing: 3 M cycles; 12 T states; 6 usec @ 2 MHz

Addressing Mode: Relative.

Flags:

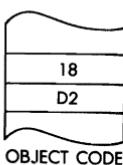
S	Z	H	P/V	N	C
---	---	---	-----	---	---

 (no effect)

Example: JR D4

Before:

After:



PC B100

PC B0D4

(This is a backwards jump.)

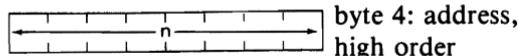
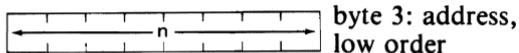
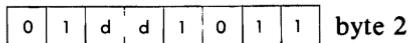
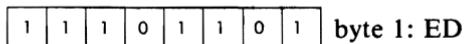
LD dd, (nn)

Load register pair dd from memory locations addressed by nn.

Function:

$$\text{dd}_{\text{low}} \leftarrow (\text{nn}); \text{dd}_{\text{high}} \leftarrow (\text{nn} + 1)$$

Format:



Description:

The contents of the memory location addressed by the memory locations immediately following the opcode are loaded into the low order of the specified register pair. The contents of the memory location immediately following the one previously loaded are then loaded into the high order of the register pair. The low order byte of the nn address immediately follows the opcode. dd may be any one of:

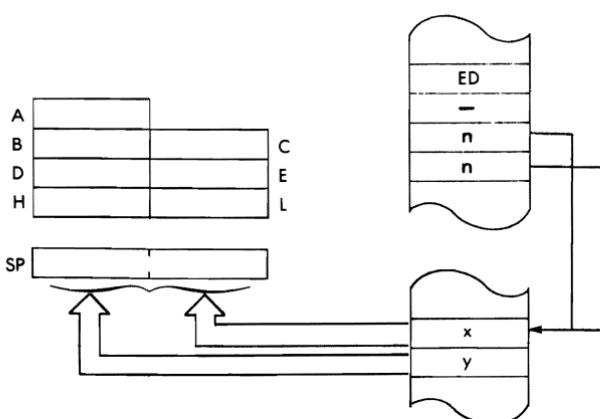
BC – 00

HL – 10

DE – 01

SP – 11

Data Flow:



PROGRAMMING THE Z80

Timing: 6 M cycles; 20 T states; 10 usec @ 2 MHz

Addressing Mode: Direct.

Byte Codes: dd: BC DE HL SP
ED-

4B	5B	6B	7B
----	----	----	----

Flags: S Z H P/V N C

--	--	--	--	--	--	--

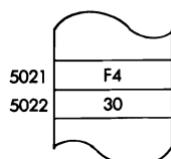
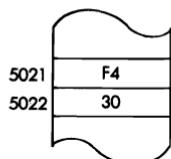
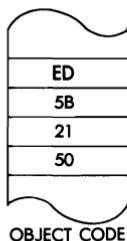
 (no effect)

Example: LD DE, (5021)

Before:



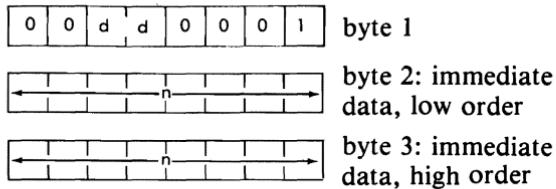
After:



LD dd, nn Load register pair dd with immediate data nn.

Function: $dd \leftarrow nn$

Format:



Description: The contents of the two memory locations immediately following the opcode are loaded into the specified register pair. The lower order byte of the data occurs immediately after the opcode. dd may be any one of:

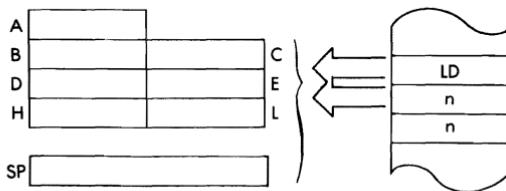
BC – 00

DE – 01

HL – 10

SP – 11

Data Flow:



Timing: 3 M cycles; 10 T states; 5 usec @ 2 MHz

Addressing Mode: Immediate.

Byte Codes:

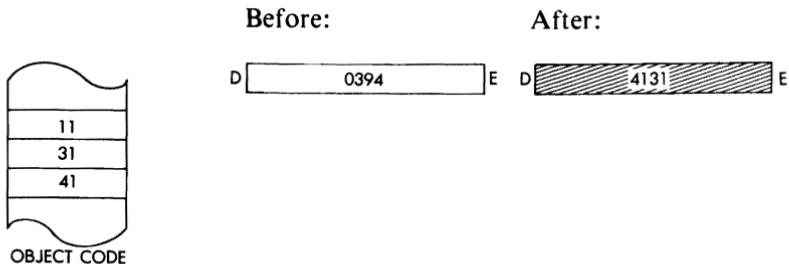
dd:	BC	DE	HL	SP
	01	11	21	31

Flags:

S	Z	H	P/V	N	C	
						(no effect)

PROGRAMMING THE Z80

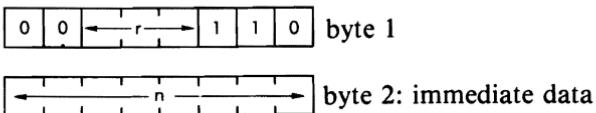
Example: LD DE, 4131



LD r, n Load register r with immediate data n.

Function: $r \leftarrow n$

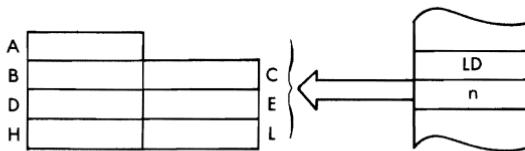
Format:



Description: The contents of the memory location immediately following the opcode location are loaded into the specified register. r may be any one of:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Data Flow:



Timing: 2 M cycles; 7 T states; 3.5 usec @ 2 MHz

Addressing Mode: Immediate.

Byte Codes:

r:	A	B	C	D	E	H	L
	3E	06	0E	16	1E	26	2E

Flags:

S	Z	H	P/V	N	C	
						(no effect).

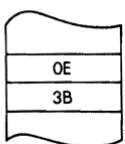
PROGRAMMING THE Z80

Example:

LD C, 3B

Before:

After:



OBJECT CODE

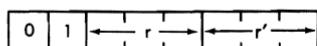


LD r, r'

Load register r from register r'.

Function:

$$r \leftarrow r'$$

Format:*Description:*

The contents of the specified source register are loaded into the specified destination register. r and r' may be any one of:

A - 111

B - 000

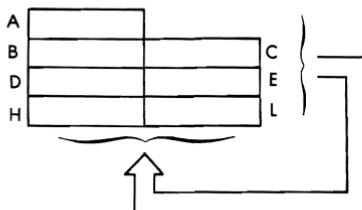
C - 001

D - 010

E - 011

H - 100

L - 101

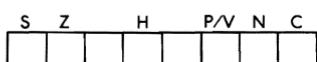
Data Flow:*Timing:*

1 M cycle; 4 T states; 2 usec @ 2 MHz

Addressing Mode: Implicit.*Byte Codes:*

	A	B	C	D	E	H	L	(source)
A	7F	78	79	7A	7B	7C	7D	
B	47	40	41	42	43	44	45	
C	4F	48	49	4A	4B	4C	4D	
D	57	50	51	52	53	54	55	
E	5F	58	59	5A	5B	5C	5D	
H	67	60	61	62	63	64	65	
L	6F	68	69	6A	6B	6C	6D	

(dest.)

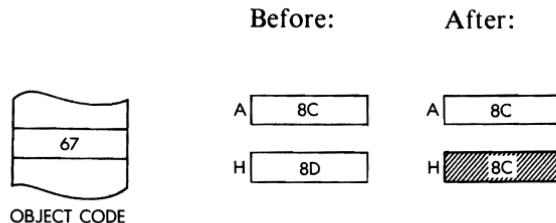
Flags:

(no effect).

PROGRAMMING THE Z80

Example:

LD H, A



LD (BC), A Load indirectly addressed memory location (BC) from the accumulator.

Function: $(BC) \leftarrow A$

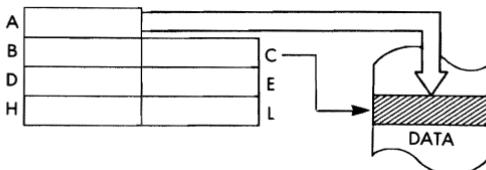
Format:

0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

 02

Description: The contents of the accumulator are loaded into the memory location addressed by the contents of the BC register pair.

Data Flow:



Timing: 2 M cycles; 7 T states; 3.5 usec @ 2 MHz

Addressing Mode: Indirect.

Flags:

S	Z	H	P/V	N	C
---	---	---	-----	---	---

 (no effect).

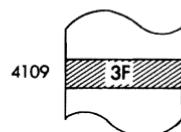
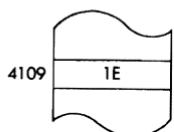
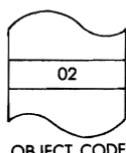
Example: LD (BC), A

Before:

A	3F
B	4109

After:

A	3F
B	4109



PROGRAMMING THE Z80

LD (DE), A Load indirectly addressed memory location (DE) from the accumulator.

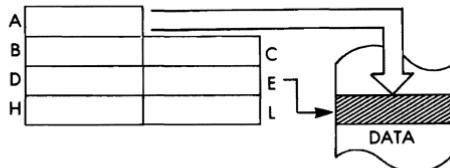
Function: $(DE) \leftarrow A$

Format:

0	0	0	1	0	0	1	0	12
---	---	---	---	---	---	---	---	----

Description: The contents of the accumulator are loaded into the memory location addressed by the contents of the DE register pair.

Data Flow:



Timing: 2 M cycles; 7 T states; 3.5 usec @ 2 MHz

Addressing Mode: Indirect.

Flags:

S	Z	H	P/V	N	C
---	---	---	-----	---	---

 (no effect)

Example: LD (DE), A

Before:

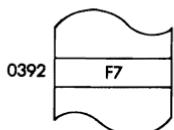
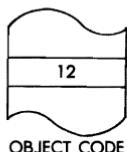
A [ED]

D [0392] E D [0392] E

After:

A [ED]

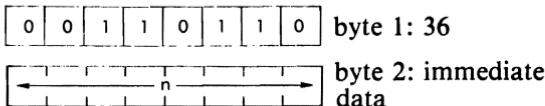
0392 [ED] 0392 E



LD (HL), n Load immediate data n into the indirectly addressed memory location (HL).

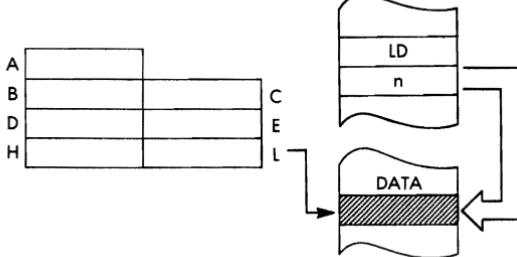
Function: $(HL) \leftarrow n$

Format:



Description: The contents of the memory location immediately following the opcode are loaded into the memory location indirectly addressed by the HL data pointer

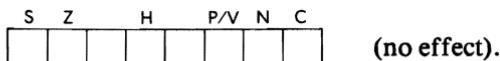
Data Flow:



Timing: 3 M cycles; 10 T states; 5 usec @ 2 MHz

Addressing Mode: Immediate/indirect.

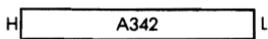
Flags:



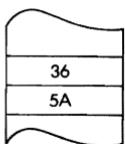
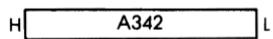
PROGRAMMING THE Z80

Example: LD (HL), 5A

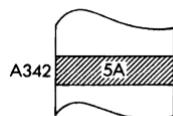
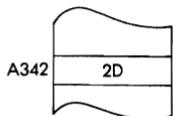
Before:



After:



OBJECT CODE

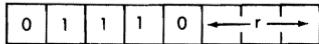


LD (HL), r

Load indirectly addressed memory location (HL) from register r.

Function: $(HL) \leftarrow r$

Format:

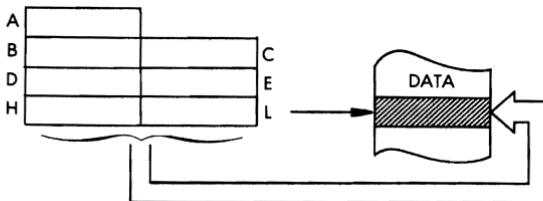


Description:

The contents of the specified register are loaded into the memory location addressed by the HL register pair. r may be any one of:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Data Flow:



Timing: 2 M cycles; 7 T states; 3.5 usec @ 2 MHz

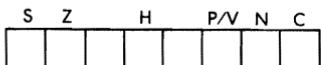
Addressing Mode: Indirect.

Byte Codes:

r:	A	B	C	D	E	H	L
	77	70	71	72	73	74	75

PROGRAMMING THE Z80

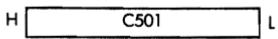
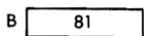
Flags:



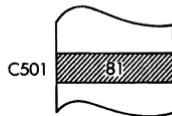
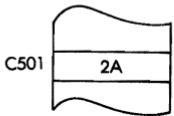
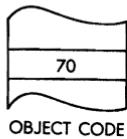
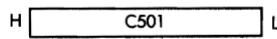
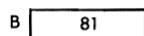
(no effect).

Example: LD (HL), B

Before:



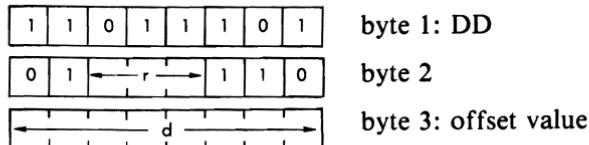
After:



LD r, (IX + d) Load register r indirect from indexed memory location (IX + d)

Function: $r \leftarrow (IX + d)$

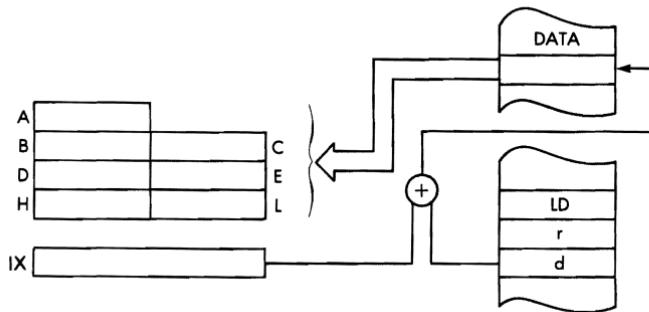
Format:



Description: The contents of the memory location addressed by the IX index register plus the given offset value, are loaded into the specified register. r may be any one of:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Data Flow:



Timing: 5 M cycles; 19 T states; 9.5 usec @ 2 MHz

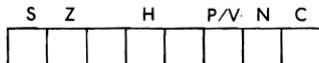
Addressing Mode: Indexed.

Byte Codes:

r:	A	B	C	D	E	H	L	-d
DD-	7E	46	4E	56	5E	66	6E	

PROGRAMMING THE Z80

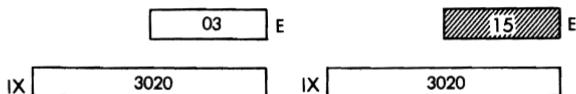
Flags:



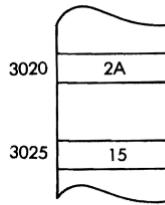
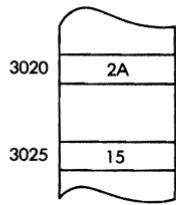
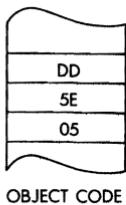
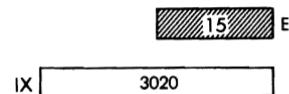
(no effect).

Example: LD E, (IX + 5)

Before:



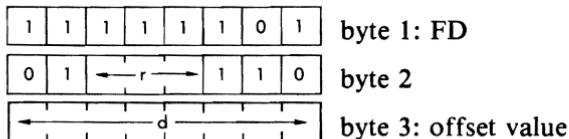
After:



LD r, (IY + d) Load register r indirect from indexed memory location (IY + d)

Function: $r \leftarrow (IY + d)$

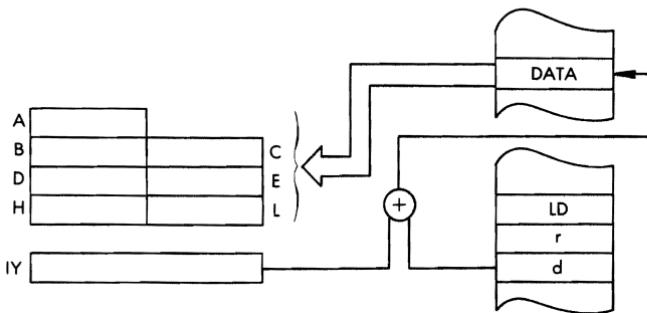
Format:



Description: The contents of the memory location addressed by the IY index register plus the given offset value, are loaded into the specified register. r may be any one of:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Data Flow:

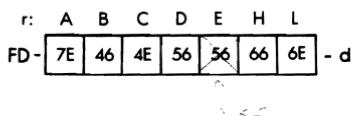


Timing: 5 M cycles, 19 T states; 9.5 usec @ 2 MHz

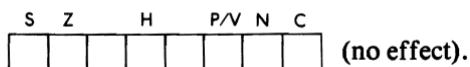
Addressing Mode: Indexed.

PROGRAMMING THE Z80

Byte Codes:

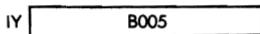
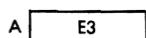


Flags:

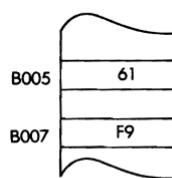
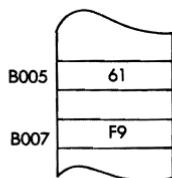
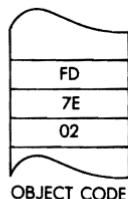
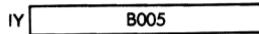


Example: LD A, (IY + 2)

Before:



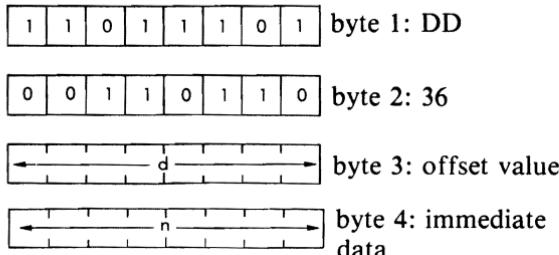
After:



LD (IX + d), n Load indexed addressed memory location (IX + d) with immediate data n.

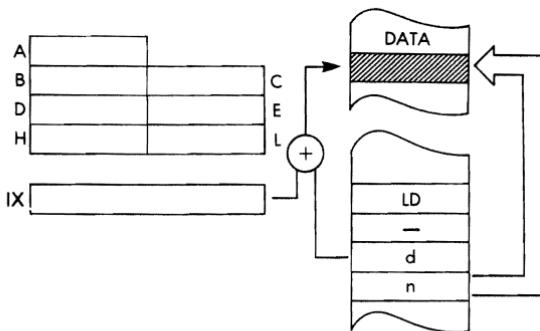
Function: $(IX + d) \leftarrow n$

Format:



Description: The contents of the memory location immediately following the offset are transferred into the memory location addressed by the contents of the index register plus the given offset value.

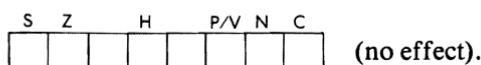
Data Flow:



Timing: 5 M cycles; 19 T states; 9.5 usec @ 2 MHz

Addressing Mode: Indexed/immediate.

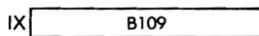
Flags:



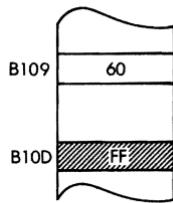
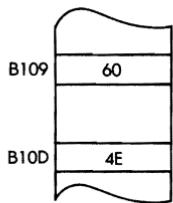
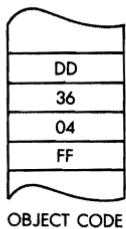
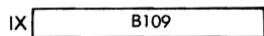
PROGRAMMING THE Z80

Example: LD (IX + 4), FF

Before:



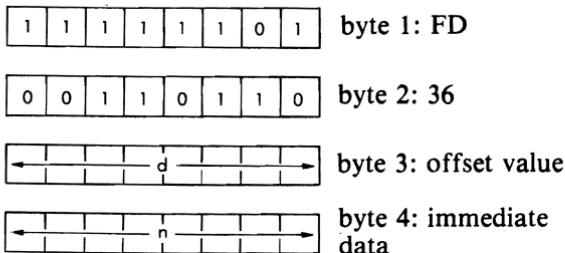
After:



LD (IY + d), n Load indexed addressed memory location (IY + d) with immediate data n.

Function: $(IY + d) \leftarrow n$

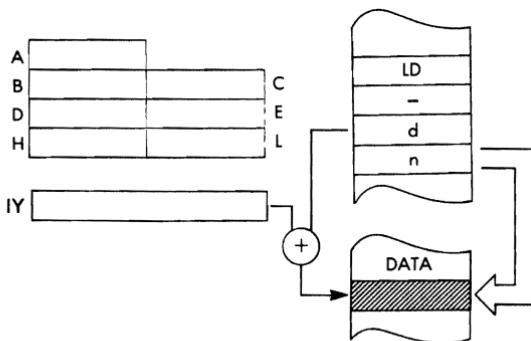
Format:



Description:

The contents of the memory location immediately following the offset are transferred into the memory location addressed by the contents of the index register plus the given offset value.

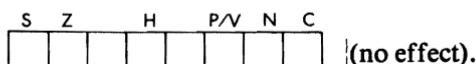
Data Flow:



Timing: 5 M cycles; 19 T states; 9.5 usec @ 2 MHz

Addressing Mode: Indexed/immediate.

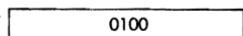
Flags:



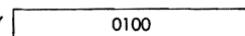
PROGRAMMING THE Z80

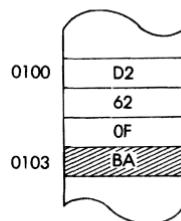
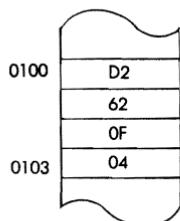
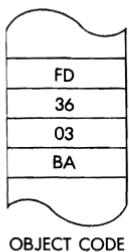
Example: LD (IY + 3), BA

Before:

IY  0100

After:

IY  0100



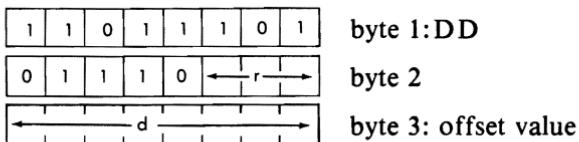
LD (IX + d),r

Load indexed addressed memory location (IX + d) from register r.

Function:

$(IX + d) \leftarrow r$

Format:



Description:

The contents of specified register are loaded into the memory location addressed by the contents of the index register plus the given offset value. r may be any one of:

A - 111

B - 000

C - 001

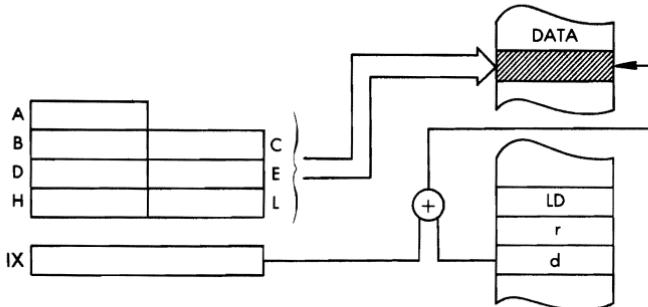
D - 010

E - 011

H - 100

L - 101

Data Flow:



Timing:

5 M cycles; 19 T states; 9.5 usec @ 2 MHz

PROGRAMMING THE Z80

Addressing Mode: Indexed.

Byte Codes:

R:	A	B	C	D	E	H	L
DD-	77	70	71	72	73	74	75

Flags:

S	Z	H	P/V	N	C

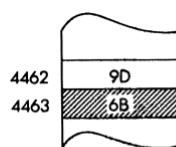
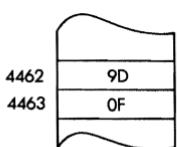
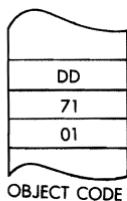
(no effect).

Example: LD (IX + 1), C

Before:



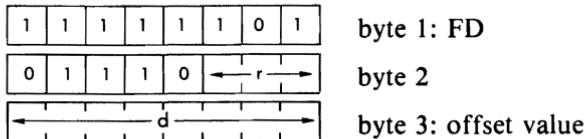
After:



LD (IY + d), r Load indexed addressed memory location (IY + d) from register r.

Function: $(IY + d) \leftarrow r$

Format:

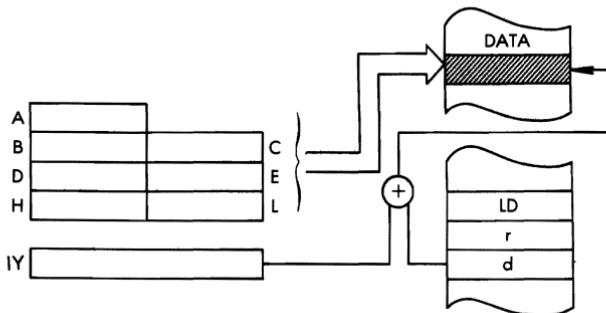


Description:

The contents of the specified register are loaded into the memory location addressed by the contents of the index register plus the given offset value. r may be any one of:

A - 111	E - 011
B - 000	H - 100
C - 001	L - 101
D - 010	

Data Flow:



Timing: 5 M cycles; 19 T states; 9.5 usec @ 2 MHz

Addressing Mode: Indexed.

Byte Codes:

r:	A	B	C	D	E	H	L
FD-	77	70	71	72	73	74	75
	-d						