Z80

Z80 ASSEMBLY LANGUAGE PROGRAMMING BY LANCE A. LEVENTHAL

Z80

ASSEMBLY LANGUAGE
PROGRAMMING

Z80

ASSEMBLY LANGUAGE PROGRAMMING

Lance A. Leventhal

Cover design by K. L. T. van Genderen.

This book is dedicated to my colleagues at the Society for Computer Simulation — Romeo Favreau, Natalie Fowler, Alexander McKenna, John McLeod, Stanley Rogers, and Chip Stockton.

ACKNOWLEDGMENTS

Contents

Contents (Continued)

Contents (Continued)

Contents (Continued)

Contents (Continued)

Contents (Continued)

Contents (Continued)

Contents (Continued)

Chapter 1
INTRODUCTION TO ASSEMBLY
LANGUAGE PROGRAMMING

This book describes assembly language programming. It assumes that you are familiar with <u>An Introduction To Microcomputers: Volume 1 — Basic Concepts</u> (particularly Chapters 6 and 7). This book does not discuss the general features of computers, microcomputers, addressing methods, or instruction sets; you should refer to <u>An Introduction To Microcomputers: Volume 1</u> for that information.

HOW THIS BOOK HAS BEEN PRINTED

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

THE MEANING OF INSTRUCTIONS

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

An instruction is simply a binary bit pattern — it must be available at the data inputs to the microprocessor at the proper time in order to be interpreted as an instruction. For example, when the Z80 microprocessor receives the 8-bit binary pattern 10000000 as the input during an instruction fetch operation, the pattern means:

> "Add the contents of Register B to the contents of the Accumulator".

Similarly, the pattern 00111110 means:

> "Load the Accumulator with the contents of the next word of program memory".

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

A COMPUTER PROGRAM

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

Actually, a computer program includes more than instructions; it also contains the data and memory addresses that the microprocessor needs to accomplish the task defined by the in-

| BINARY INSTRUCTIONS |

| COMPUTER PROGRAM |

structions. Clearly, if the microprocessor is to perform an addition, it must have two numbers to add and a destination for the result. The computer program must determine the sources of the data and the destination of the result as well as specifying the operation to be performed.

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

Ultimately every program becomes translated into a set of binary numbers. For example, this is the Z80 program that adds the contents of memory locations 60_{16} and 61_{16} and places the result in memory location 62_{16}:

```
00111010
01100000
00000000
01000111
00111010
01100001
00000000
10000000
00110010
01100010
00000000
```

This is a machine language, or object, program. If this program were entered into the memory of a Z80-based microcomputer, the microcomputer would be able to execute it directly.

OBJECT PROGRAM
MACHINE LANGUAGE PROGRAM

THE PROGRAMMING PROBLEM

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

1) The programs are difficult to understand or debug (binary numbers all look the same, particularly after you have looked at them for a few hours).

2) The programs are slow to enter since you must enter each bit individually.

3) The programs do not describe the task which you want the computer to perform in anything resembling a human readable format.

4) The programs are long and tiresome to write.

5) The programmer often makes careless errors that are very difficult to find.

For example, the **following version of the addition object program contains a single bit error. Try to find it:**

```
00111010
01100000
00000000
01000111
01110010
01100001
00000000
10000000
00110010
01100010
00000000
```

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

USING OCTAL OR HEXADECIMAL

We can improve the situation somewhat by writing instructions using octal or hexadecimal, rather than binary, numbers. We will use hexadecimal numbers in this book because they are

<div style="float:right; border:1px solid black; padding:4px;">
OCTAL OR
HEXADECIMAL
</div>

shorter, and because they are the standard for the microprocessor industry. Table 1-1 defines the hexadecimal digits and their binary equivalents. **The Z80 program to add two numbers now becomes:**

```
3A
60
00
47
3A
61
00
80
32
62
00
```

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

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

```
3A
60
00
47
72
61
00
80
32
62
00
```

The mistake is easier to spot.

What do we do with this hexadecimal program? The microprocessor understands only binary instruction codes. The answer is that we must convert the hexadecimal numbers to binary numbers. This conversion is a repetitive, tiresome task. People who attempt it make all sorts of petty mistakes, such as looking at the wrong line, dropping a bit, or transposing a bit or a digit.

This repetitive, grueling task is, however, a perfect job for a computer. The computer never gets tired or bored and never makes silly mistakes. **The idea then is to write a program which takes**

<div style="float:right; border:1px solid black; padding:4px;">
HEXADECIMAL
LOADER
</div>

hexadecimal numbers and converts them into binary numbers. This is a standard program provided with many microprocessors; it is called a "hexadecimal loader."

Is a hexadecimal loader worth having? If you are willing to write a program using binary numbers, and you are prepared to enter the program in its binary form into the computer, then you will not need the hexadecimal loader.

If you choose the hexadecimal loader, you will have to pay a price for it. The hexadecimal loader is itself a program which you must load into memory. Furthermore, the hexadecimal loader will occupy memory — memory that you may want to use in some other way.

The basic tradeoff, therefore, is the cost and memory requirements of the hexadecimal loader versus the savings in programmer time.

A hexadecimal loader is well worth its small cost.

A hexadecimal loader certainly does not solve every programming problem. The hexadecimal version of the program is still difficult to read or understand; for example, it does not distinguish instructions from data or addresses, nor does the program listing provide any suggestion as to what the program does. What does 32 or 47 or 3A mean? Memorizing a card full of codes is hardly an appetizing proposition. Furthermore, the codes will be entirely different for a different microprocessor, and the program will require a large amount of documentation.

Table 1-1. Hexadecimal Conversion Table

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

INSTRUCTION CODE MNEMONICS

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

In fact, every microprocessor manufacturer (they can't remember hexadecimal codes either) provides a set of mnemonics for the microprocessor instruction set. **You do not have to abide by the manufacturer's mnemonics;** there is nothing sacred about them.

PROBLEM WITH MNEMONICS

However, they are standard for a given microprocessor and therefore understood by all users. These are the instruction names that you will find in manuals, cards, books, articles, and programs. The problem with selecting instruction mnemonics is that not all instructions have "obvious" names. Some instructions do have obvious names (e.g., ADD, AND, OR), others have obvious contractions (e.g., SUB for subtraction, XOR for exclusive OR), while still others have neither. The result is such mnemonics as WMP, PCHL, and even SOB (try and guess what that means!). Most manufacturers come up with mostly reasonable names and a few hopeless ones. However, users who devise their own mnemonics rarely seem to do much better than the manufacturer.

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

If we use standard Z80 instruction and register mnemonics, as defined by Zilog, our Z80 addition program becomes:

```
LD      A,(60H)
LD      B,A
LD      A,(61H)
ADD     A,B
LD      (62H),A
```

The program is still far from obvious, but at least some parts are comprehensible. ADD A,B is a considerable improvement over 80; LD does suggest loading data into a register or memory location. **Such a program is an assembly language program.**

THE ASSEMBLER PROGRAM

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

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

Instruction Name		Hexadecimal Equivalent
LD	A,(NN)	3A
LD	B,A	47
ADD	A,B	80
LD	(NN),A	32

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

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

The tradeoffs we discussed in connection with the hexadecimal loader are magnified in the case of the assembler. Assemblers are more expensive, occupy more memory, and require more peripherals and execution time than do hexadecimal loaders. While users may (and often do) write their own loaders, few care to write their own assemblers.

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

ADDITIONAL FEATURES OF ASSEMBLERS

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

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

All of these features, of course, involve additional cost and memory. Microcomputers generally have much simpler assemblers than do larger computers, but the tendency always is for the size of assemblers to increase. You will often have a choice of assemblers. The important criterion is not how many offbeat features the assembler has, but rather how convenient it is to work with in normal practice.

> **CHOOSING AN ASSEMBLER**

DISADVANTAGES OF ASSEMBLY LANGUAGE

The assembler, like the hexadecimal loader, does not solve all the problems of programming. One problem is the tremendous gap between the microcomputer instruction set and the tasks which the microcomputer is to perform. Computer instructions tend to do things like add the contents of two registers, shift the contents of the Accumulator one bit, or place a new value into the Program Counter. On the other hand, a user generally wants a microcomputer to do something like check if an analog reading has exceeded a threshold, look for and react to a particular command from a teletypewriter, or activate a relay at the proper time. An assembly language programmer must translate such tasks into a sequence of simple computer instructions. The translation can be a difficult, time-consuming job.

Furthermore, **if you are programming in assembly language, you must have detailed knowledge of the particular microcomputer that you are using.** You must know what registers and instructions the microcomputer has, precisely how the instructions affect the various registers, what addressing methods the computer uses, and a myriad of other information. None of this information is relevant to the task which the microcomputer must ultimately perform.

In addition, assembly language programs are not portable.
Each microcomputer has its own assembly language, which
reflects its own architecture. An assembly language program written for the Z80 will

> **PORTABILITY**

not run on the Motorola 6800, the Fairchild F8, or the National Semiconductor PACE. For example, the addition program written for the Motorola 6800 would be:

```
LDAA    $60
ADDA    $61
STAA    $62
```

The lack of portability not only means that you won't be able to use your assembly language program on another microcomputer, but it also means that you won't be able to use any programs that weren't specifically written for the microcomputer you are using. This is a particular drawback for microcomputers, since these devices are new and few assembly language programs exist for them. The result, too frequently, is that you are on your own. If you need a program to perform a particular task, you are not likely to find it in the small program libraries that most manufacturers provide. Nor are you likely to find it in an archive, journal article, or someone's old program file. You will probably have to write it yourself.

HIGH-LEVEL LANGUAGES

The solution to many of the difficulties associated with as- **COMPILER** **sembly language programs is to use, instead, "high-level" or "procedure-oriented" languages. Such languages allow you to describe tasks in forms that are problem oriented rather than computer oriented. Each statement in a high-level language performs a recognizable function; it will generally correspond to many assembly language instructions. A program called a compiler translates the high-level language source program into object code or machine language instructions.**

Many different high-level languages exist for different types of **FORTRAN** tasks. If, for example, you can express what you want the computer to do in algebraic notation, you can write your program in FORTRAN (Formula Translation Language), the oldest and one of the most widely used of the high-level languages. Now, if you want to add two numbers, you just tell the computer:

$$SUM = NUMB1 + NUMB2$$

That is a lot simpler (and a lot shorter) than either the equivalent machine language program or the equivalent assembly language program. Other high-level languages include COBOL (for business applications), PASCAL (another algebraic language), PL/1 (a combination of FORTRAN, ALGOL, and COBOL), and APL and BASIC (languages that are popular for time-sharing systems).

ADVANTAGES OF HIGH-LEVEL LANGUAGES

Clearly, high-level languages make programs easier and faster to write. A common estimate is that a programmer can write a program about ten times as fast in a high-level language as compared to assembly language. That is just writing the program; it does not include problem definition, program design, debugging, testing, or documentation, all of which become simpler and faster. The high-level language program is, for instance, partly self-documenting. Even if you do not know FORTRAN, you probably could tell what the statement illustrated above does.

High-level languages solve many other problems associ- **MACHINE** **ated with assembly language programming.** The high-level **INDEPENDENCE** language has its own syntax (usually defined by a national or **OF HIGH-LEVEL** international standard). The language does not mention the in- **LANGUAGES** struction set, registers, or other features of a particular computer. The compiler takes care of all such details. Programmers can concentrate on their own tasks; they do not need a detailed understanding of the underlying CPU architecture — for that matter, they do not need to know anything about the computer they are programming.

**Programs written in a high-level language are portable —
at least, in theory.** They will run on any computer or
microcomputer that has a standard compiler for that language.

PORTABILITY
OF HIGH-LEVEL
LANGUAGES

At the same time, all previous programs written in a high-level
language for prior computers are available to you when programming a new computer.
This can mean thousands of programs in the case of a common language like FORTRAN
or BASIC.

DISADVANTAGES OF HIGH-LEVEL LANGUAGES

**Well, if all the good things we have said about high-level languages are true, if you
can write programs faster and make them portable besides, why bother with as-
sembly languages? Who wants to worry about registers, instruction codes,
mnemonics, and all that garbage! As usual, there are disadvantages that balance
the advantages.**

One obvious problem is that **you have to learn the "rules" or
"syntax" of any high-level language** you want to use. A high-
level language has a fairly complicated set of rules. You will find

SYNTAX OF
HIGH-LEVEL
LANGUAGES

that it takes a lot of time just to get a program that is syntactically
correct (and even then it probably will not do what you want). A high-level computer
language is like a foreign language. If you have a little talent, you will get used to the
rules and be able to turn out programs that the compiler will accept. Still, learning the
rules and trying to get the program accepted by the compiler doesn't contribute
directly to doing your job.

Here, for example, are some FORTRAN rules:

- Labels must be numbers placed in the first five card columns
- Statements must start in column seven
- Integer variables must start with the letters I, J, K, L, M, or N

Another obvious problem is that **you need a compiler to transl-
ate programs written in a high-level language.** Compilers are
expensive and use a large amount of memory. While most assem-

COST OF
COMPILERS

blers occupy 2K to 16K bytes of memory (1K = 1024), compilers occupy 4K to 64K
bytes. So the amount of overhead involved in using the compiler is rather large.

Furthermore, **only some compilers will make the implementa-
tion of your task simpler.** FORTRAN, for example, is well-suited
to problems that can be expressed as algebraic formulas. If,

ALGEBRAIC
NOTATION

however, your problem is controlling a printer, editing a string of characters, or monitor-
ing an alarm system, your problem cannot be easily expressed in algebraic notation. In
fact, formulating the solution in algebraic notation may be more awkward and more
difficult than formulating it in assembly language. One answer is to use a more suitable
high-level language. Some such languages exist, but they are far less widely used and
standardized than FORTRAN. You will not get many of the advantages of high-level
languages if you use these so-called system implementation languages.

**High-level languages do not produce very efficient
machine language programs.** The basic reason for this is that
compilation is an automatic process which is riddled with com-

INEFFICIENCY
OF HIGH-LEVEL
LANGUAGES

promises to allow for many ranges of possibilities. The com-
piler works much like a computerized language translator — sometimes the words are
right but the sounds and sentence structures are awkward. A simple compiler cannot
know when a variable is no longer being used and can be discarded, or when a register
should be used rather than a memory location, or when variables have simple relation-
ships. The experienced programmer can take advantage of shortcuts to shorten execu-

1-8

tion time or reduce memory usage. A few compilers (known as optimizing compilers) can also do this, but such compilers are much larger and slower than regular compilers.

The general advantages and disadvantages of high-level languages are:

Advantages:

- More convenient descriptions of tasks
- More efficient program coding
- Easier documentation
- Standard syntax
- Independence of the structure of a particular computer
- Portability
- Availability of library and other programs

ADVANTAGES OF HIGH-LEVEL LANGUAGES

Disadvantages:

- Special rules
- Extensive hardware and software support required
- Orientation of common languages to algebraic or business problems
- Inefficient programs
- Difficulty of optimizing code to meet time and memory requirements
- Inability to use special features of a computer conveniently

DISADVANTAGES OF HIGH-LEVEL LANGUAGES

HIGH-LEVEL LANGUAGES FOR MICROPROCESSORS

Microprocessor users will encounter several special difficulties when using high-level languages. Among these are:

- Few high-level languages exist for microprocessors
- No standard languages are widely available
- Few compilers actually run on microcomputers. Those that do often require very large amounts of memory.
- Most microprocessor applications are not well-suited to high-level languages.
- Memory costs are often critical in microprocessor applications.

The lack of high-level languages is partly a result of the fact that microprocessors are quite new and are the products of semiconductor manufacturers rather than computer manufacturers.

Very few high-level languages exist for microprocessors. The most common are the PL/1 type languages (such as Intel's PL/M. Motorola's MPL. and Signetics' $PL\mu S$), BASIC. and PASCAL.

Even the few high-level languages that exist do not conform to recognized standards, so the microprocessor user cannot expect to gain much program portability, access to program libraries, or use of previous experience or programs. The main advantages remaining are the reduction in programming effort and the smaller amount of detailed understanding of the computer architecture that is necessary.

The overhead involved in using a high-level language with microprocessors is considerable. Microprocessors themselves are better suited to control and slow interactive applications than they are to the character manipulation and language analysis involved in compilation. Therefore, most compilers for microprocessors will not run on a microprocessor-based system. Instead, they require a much larger computer. i.e., they are cross-compilers rather than self-compilers. A user must not only

OVERHEAD FOR HIGH-LEVEL LANGUAGES

bear the expense of the larger computer but must also physically transfer the program from the larger computer to the micro.

A few self-compilers are available. These compilers run on the microcomputer for which they produce object code. Unfortunately, they require large amounts of memory (16K or more), plus special supporting hardware and software.

High-level languages also are not generally well-suited to microprocessor applications. Most of the common languages were devised either to help solve scientific problems or to handle large-scale business data processing. Few microprocessor

UNSUITABILITY OF HIGH-LEVEL LANGUAGES

applications fall in either of these areas. Most microprocessor applications involve sending data and control information to output devices and receiving data and status information from input devices. Often the control and status information consists of a few binary digits with very precise hardware-related meanings. If you try to write a typical control program in a high-level language, you often feel like someone who is trying to eat soup with chopsticks. For tasks in such areas as test equipment, terminals, navigation systems, signal processing, and business equipment, the high-level languages work much better than they do in instrumentation, communications, peripherals, and automotive applications.

Applications better suited to high-level languages are those which require large memories. If, as in a valve controller, electronic game, appliance controller, or small instrument, the cost of a single memory chip is important, then the inefficiency of high-level languages is intolerable. If, on the other hand, as in a terminal or

APPLICATION AREAS FOR LANGUAGE LEVELS

test equipment, the system has many thousands of bytes of memory anyway, the inefficiency of high-level languages is not as important. Clearly the size of the program and the volume of the product are important factors as well. A large program will greatly increase the advantages of high-level languages. On the other hand, a high-volume application will mean that fixed software development costs are not as important as memory costs that are part of each system.

WHICH LEVEL SHOULD YOU USE?

That depends on your particular application. Let us briefly note some of the factors which may favor particular levels:

Machine Language:

- Virtually no one programs in machine language. Its use cannot be justified considering the low cost of an assembler and the increase in programming speed an assembler provides.

APPLICATIONS FOR MACHINE LANGUAGE

Assembly Language:

- Short to moderate sized programs
- Applications where memory cost is a factor
- Real-time control applications
- Limited data processing
- High-volume applications
- More input/output or control than computation

APPLICATIONS FOR ASSEMBLY LANGUAGE

High-Level Languages:

- Long programs
- Low-volume applications requiring long programs
- Applications requiring large memories

APPLICATIONS FOR HIGH-LEVEL LANGUAGE

- **More computation than input/output or control**
- **Compatibility with similar applications using larger computers**
- **Availability of specific programs in a high-level language which can be used in the application**

Many other factors are also important, such as the availability of a larger computer for use in development, experience with particular languages, and compatibility with other applications.

If hardware will ultimately be the largest cost in your application, or if speed is critical you should favor assembly language. But be prepared to spend extra time in software development in exchange for lower memory costs and higher execution speeds. If software will be the largest cost in your application, you should favor a high-level language. But be prepared to spend the extra money required for the supporting hardware and software.

Of course, no one except some theorists will object if you use both assembly and high-level languages. You can write the program originally in a high-level language and then patch some sections in assembly language. However, most users prefer not to do this because of the havoc it creates in debugging, testing, and documentation.

HOW ABOUT THE FUTURE?

We expect that the future will tend to favor high-level languages for the following reasons:

- Programs always seem to add extra features and grow larger
- Hardware and memory are becoming less expensive
- Software and programmers are becoming more expensive
- Memory chips are becoming available in larger sizes, at lower "per bit" cost, so actual savings in chips are less likely
- More compilers are becoming available
- More suitable and more efficient high-level languages are being developed
- More standardization of high-level languages will occur

> **FUTURE TRENDS IN LANGUAGE LEVELS**

Assembly language programming of microprocessors will not be a dying art any more than it is now for large computers. But longer programs, cheaper memory, and more expensive programmers will make software costs a larger part of most applications. The edge in many applications will therefore go to high-level languages.

WHY THIS BOOK?

If the future would seem to favor high-level languages, why have a book on assembly language programming? The reasons are:

1) Most current microcomputer users program in assembly language (almost two-thirds, according to one recent survey).
2) Many microcomputer users will continue to program in assembly language since they need the detailed control that it provides.
3) No suitable high-level language has yet become widely available or standardized.
4) Many applications require the efficiency of assembly language.
5) An understanding of assembly language can help in evaluating high-level languages.

The rest of this book will deal exclusively with assemblers and assembly language programming. However, we do want readers to know that assembly language is not the only alternative. You should watch for new developments that may significantly reduce programming costs if such costs are a major factor in your application.

REFERENCES

Some overall comparisons of the time required to write various types of programs at different language levels are in M.H. Halstead, <u>Elements of Software Science</u>, American Elsevier, New York, 1977 and in V. Schneider, "Prediction of Software Effort and Project Duration - Four New Formulas", <u>SIGPLAN Notices</u>, June 1978, pp. 49-55.

Chapter 2
ASSEMBLERS

This chapter discusses the functions performed by assemblers, beginning with features common to most assemblers, and proceeding through more elaborate capabilities such as macros and conditional assembly. You may wish to skim this chapter for the present and return to it when you feel more comfortable with the material.

FEATURES OF ASSEMBLERS

As we mentioned previously, today's assemblers do much more than translate assembly language mnemonics into binary codes. But we will first describe how an assembler handles the translation of mnemonics before describing additional assembler features. Finally, we will explain how assemblers are used.

ASSEMBLER INSTRUCTIONS

Assembly language instructions (or "statements") are divided into a number of fields, as shown in Table 2-1.

The operation code field is the only field which can never be empty; it always contains either an instruction mnemonic or a directive to the assembler, called a pseudo-instruction, pseudo-operation, or pseudo-op.

> ASSEMBLY
> LANGUAGE
> FIELDS

The address field may contain an address or data, or it may be blank.

Table 2-1. The Fields of an Assembly Language Instruction

Label Field	Operation Code or Mnemonic Field	Operand or Address Field	Comment Field
START:	LD	A,(VAL1)	;LOAD FIRST NUMBER INTO A
	LD	B,A	;SAVE IN B
	LD	A,(VAL2)	;LOAD SECOND NUMBER INTO A
	ADD	A,B	;ADD FIRST NUMBER TO A
	LD	(SUM),A	;STORE SUM
NEXT:	?	?	;NEXT INSTRUCTION
VAL1:	DEFS	1	
VAL2:	DEFS	1	
SUM:	DEFS	1	

The comment and label fields are optional. A programmer will assign a label to a statement or add a comment as a personal convenience, e.g., to make the program easier to code and read.

Of course, the assembler must have some way of telling **where one field ends and another begins.** Assemblers that use punched card input often require that each field start in a specific card column. This is a <u>fixed format</u>. However, fixed formats may be inconvenient when the input medium is paper tape; fixed formats are also a nuisance to programmers. The alternative is a <u>free format</u>, where the fields may appear anywhere on the line.

	FORMAT

If the assembler cannot use the position in the line to tell the fields apart, it must use something else. **Most assemblers use a special symbol or delimiter at the beginning or end of each field.** The most obvious delimiter is the space character. Commas, periods, semicolons, colons, slashes, question marks and other characters that would not otherwise be used in assembly language programs also may serve as delimiters. Table 2-2 lists standard Zilog Z80 assembler delimiters.

	DELIMITERS

Table 2-2. Standard Z80 Assembler Delimiters

:	after a label
'space'	between operation code and address
,	between operands in the address field
;	before a comment

You will have to exercise a little care with delimiters. Some assemblers are fussy about extra spaces or the appearance of delimiters in comments or labels. A well-written assembler will handle these minor problems, but many assemblers are not well-written. Our recommendation is simple: avoid potential problems if you can. The following rules will help:

1) Do not use extra spaces, particularly after commas that separate operands.

2) Do not use delimiter characters in names or labels.

3) Include standard delimiters even if your assembler does not require them. Your programs will then be assembled by any assembler.

LABELS

The label field is the first field in an assembly language instruction: it may be blank. If a label is present, the assembler assigns to the label the value of the address for the memory location into which the first object program byte for that instruction is loaded. You may subsequently use the label as data or as an address in another instruction's operand field. The assembler will replace the label with the assigned value when creating an object program.

	LABEL FIELD

Labels are most frequently used in Jump, Call or Branch instructions. These instructions place a new value in the Program Counter and so alter the normal sequential execution of instructions. JUMP 150_{16} means "place the value 150_{16} into the Program Counter". The next instruction to be executed will be the one in memory location 150_{16}. The instruction JUMP START means "place the value assigned to the label START into the Program Counter". The next instruction to be executed will be the one in the memory location to which the label START has been assigned. Table 2-3 contains an example.

	LABELS IN JUMP INSTRUCTIONS

Table 2-3. Assigning and Using a Label

```
                    ASSEMBLY LANGUAGE PROGRAM

        START        LOAD ACCUMULATOR 100
                        .
                        .
                        . (MAIN PROGRAM)
                        .
                        .
                        .
                     JUMP START
```

When the machine language version of this program is executed, the instruction JUMP START causes the address of the instruction labeled START to be placed into the Program Counter. The instruction with the label START will be executed next.

Why use a label? Here are some reasons:

1) A label makes a program location easier to find and remember.

2) The label can be moved to change or correct a program. You do not have to change any subsequent instructions that use the label; the assembler will make all the necessary changes.

3) The assembler or loader can relocate the whole program by adding a constant (a relocation constant) to each address in which a label was used. Thus we can move the program to allow for the insertion of other programs or simply to rearrange memory.

RELOCATION CONSTANT

4) The program is easier to use as a library program, i.e., it is easier for someone else to take your program and add it to some totally different program.

5) You do not have to figure out memory addresses. Figuring out memory addresses is particularly difficult with microprocessors which have instructions that vary in length.

It makes sense to assign a label to any instruction that you might want to use as a destination or otherwise identify.

The next question is what label to use. The assembler often places some restrictions on the number of characters (usually 5 or 6), the leading character (often must be a letter), and the trailing characters (often must be letters, numbers, or one of a few special characters). Beyond these restrictions, the choice is up to you.

CHOOSING LABELS

Our own preference is to **use labels that suggest their purpose,** i.e., mnemonic labels. Typical examples are ADDW in a routine that adds one word into a sum, SRETX in a routine that searches for the ASCII character ETX, or NKEYS for a location in data memory that contains the number of key entries. Meaningful labels are easier to remember and contribute to program documentation. Some programmers prefer to use a standard format for labels, such as starting with L0000. These labels are self-sequencing (you can skip a few numbers to permit insertions), but they do not help document the program.

Some label selection rules will keep you out of trouble. We recommend the following:

RULES OF LABELING

1) Do not use labels that are the same as operation codes or other mnemonics. Most assemblers will not allow this usage; others will, but it is very confusing.

2-3

2) Do not use labels that are longer than the assembler permits. Assemblers have various truncation rules.

3) Avoid special characters (non-alphabetic and non-numeric) and lower-case letters. Some assemblers will not permit them; others allow only certain ones. The simplest practice is to stick to capital letters and numbers.

4) Start each label with a letter. Such labels are always acceptable.

5) Do not use labels that could be confused with each other. Avoid the letters I, O and Z and the numbers 0, 1 and 2. Also avoid things like XXXX and XXXXX. There's no sense tempting fate and Murphy's laws.

6) When you are not sure if a label is legal, do not use it. You will not get any real benefit from discovering exactly what the assembler will accept.

These are recommendations, not rules. You do not have to follow them, but don't blame us if you waste time on silly problems.

ASSEMBLER OPERATION CODES (MNEMONICS)

The main task of the assembler is the translation of mnemonic operation codes into their binary equivalents. The assembler performs this task using a fixed table much as you would if you were doing the assembly by hand.

The assembler must, however, do more than just translate the operation codes. It **must also somehow determine how many operands the instruction requires and what type they are.** This may be rather complex — some instructions (like a Halt) have no operands, others (like an Addition or a Jump instruction) have one, while still others (like a transfer between registers or a multiple-bit shift) require two. Some instructions may even allow alternatives, e.g., some computers have instructions (like Shift or Clear) that can apply either to the Accumulator or to a memory location. We will not discuss how the assembler makes these distinctions; we will just note that it must do so.

PSEUDO-OPERATIONS

Some assembly language instructions are not directly translated into machine language instructions. These instructions are directives to the assembler; they assign the program to certain areas in memory, define symbols, designate areas of RAM for temporary data storage, place tables or other fixed data in memory, allow references to other programs, and perform minor housekeeping functions.

PSEUDO-
OPERATIONS

ASSEMBLER
DIRECTIVE

To use these assembler directives, or pseudo-operations, a programmer places the pseudo-operation's mnemonic in the operation code field and, if the specified pseudo-operation requires it, an address or data in the address field.

The most common pseudo-operations are:

> DATA
> EQUATE or DEFINE
> ORIGIN
> RESERVE

Linking pseudo-operations are:

> ENTRY
> EXTERNAL

Different assemblers use different names for these operations, but the purposes are the same. Housekeeping pseudo-operations include:

END
LIST
NAME
PAGE
SPACE
TITLE

We will discuss these pseudo-operations briefly, although their functions are usually obvious.

THE DATA PSEUDO-OPERATION

The DATA pseudo-operation allows the programmer to enter fixed data into memory. This data may include:

- Lookup tables
- Code conversion tables
- Messages
- Synchronization patterns
- Thresholds
- Names
- Coefficients for equations
- Commands
- Conversion factors
- Weighting factors
- Characteristic times or frequencies
- Subroutine addresses
- Key identifications
- Test patterns
- Character generation patterns
- Identification patterns
- Tax tables
- Standard forms
- Masking patterns
- State transition tables

The DATA pseudo-operation treats the data as a permanent part of the program.

The format of a DATA pseudo-operation is usually quite simple. An instruction like:

DZCON   DATA   12

will place the number 12 in the next available memory location and assign that location the name DZCON. Usually every DATA pseudo-operation has a label, unless it is one of a series of DATA pseudo-operations. The data and label may take any form that the assembler permits.

Most assemblers allow more elaborate DATA instructions that handle a large amount of data at one time, e.g.:

EMESS   DATA   'ERROR'
SQRS   DATA   1,4,9,16,25

A single instruction may fill many words of program memory, limited only by the length of a line. Note that if you cannot get all the data on one line, you can always follow one DATA instruction with another, e.g.:

```
MESSG   DATA    'NOW IS THE '
        DATA    'TIME FOR ALL '
        DATA    'GOOD MEN '
        DATA    'TO COME TO THE '
        DATA    'AID OF THEIR '
        DATA    'COUNTRY'
```

Microprocessor assemblers typically have some variations of standard DATA pseudo-operations. DEFINE BYTE or FORM CONSTANT BYTE handles 8-bit numbers; DEFINE WORD or FORM CONSTANT WORD handles 16-bit numbers or addresses. Other special pseudo-operations may handle character-coded data.

THE EQUATE (or DEFINE) PSEUDO-OPERATION

The EQUATE pseudo-operation allows the programmer to equate labels and names with addresses or data. This pseudo-operation is almost always given the mnemonic EQU. The

> **DEFINING NAMES**

names may refer to device addresses, numeric data, starting addresses, fixed addresses, etc.

The EQUATE pseudo-operation assigns the numeric value in its operand field to the label in its label field. Here are two examples:

```
TTY     EQU     5
LAST    EQU     5000
```

Most assemblers will allow you to define one label in terms of another, e.g.:

```
LAST    EQU     FINAL
ST1     EQU     START+1
```

The label in the operand field must, of course, have been previously defined. Often, the operand field may contain more complex expressions, as we shall see later. Double name assignments (two names for the same data or address) may be useful in patching together programs which use different names for the same variable (or different spellings of what was supposed to be the same name).

Note that an EQU pseudo-operation does not cause the assembler to place anything into memory. The assembler simply enters an additional name into a table (called a symbol table)

> **SYMBOL TABLE**

which the assembler maintains. This table, unlike the mnemonic table, must be in RAM since it varies with each program. The assembler program will always need some RAM to hold the symbol table; the more RAM it has, the more symbols it can accept. This RAM is in addition to any which the assembler needs as temporary storage.

When do you use a name? The answer is: whenever you have a parameter that has some meaning besides its ordinary numeric value, or the numeric value of the parameter might be changed.

> **USE OF NAMES**

We typically assign names to time constants, device addresses, masking patterns, conversion factors, and the like. A name like DELAY, TTY, KBD, NROW, or OPEN not only makes the parameter easier to change, but it also adds to program documentation. We also assign names to memory locations that have special purposes; they may hold data, mark the start of the program, or be available for intermediate storage.

What name do you use? The best rules are much the same as in the case of labels, except that here meaningful names really count. Why not call the teletypewriter TTY instead of X15, a bit time delay BTIME or BTDLY rather than WW, the number of the

> **CHOICE OF NAMES**

"GO" key on a keyboard GOKEY rather than HORSE? This advice seems straightforward, but a surprising number of programmers do not follow it.

Where do you place the EQUATE pseudo-operations? The best place is at the start of the program, under appropriate comment headings such as I/O ADDRESSES, TEMPORARY STORAGE, TIME CONSTANTS, or PROGRAM LOCATIONS. This

```
PLACEMENT
OF
DEFINITIONS
```

makes the definitions easy to find if you want to change them. Furthermore, another user will be able to look up all the definitions in one centralized place. Clearly this practice improves documentation and makes the program easier to use.

Definitions used only in a specific subroutine should appear at the start of the subroutine.

THE ORIGIN PSEUDO-OPERATION

The ORIGIN pseudo-operation (almost always abbreviated ORG) allows the programmer to locate programs, subroutines, or data anywhere in memory. Programs and data may be located in different areas of memory depending on the memory configuration. Startup routines, interrupt service routines, and other required programs may be scattered around memory at fixed or convenient addresses.

The assembler maintains a Location Counter (comparable to the computer's Program Counter) which contains the location in memory at which the next byte of object code generated by

```
LOCATION
COUNTER
```

the assembler will reside when the program is loaded. An ORG pseudo-operation causes the assembler to place a new value into the Location Counter, much as a Jump instruction causes the CPU to place a new value into the Program Counter. The output from the assembler must not only contain instructions and data, but must also indicate to the loader program where in memory it should place the instructions and data.

Microprocessor programs often contain several ORIGIN statements for the following purposes:

Reset (startup) address	Main program
Interrupt service addresses	Subroutines
Trap addresses	Memory addresses for
RAM storage	input/output devices
Memory stack	or special functions

Still other ORIGIN statements may allow room for later insertions, place tables or data in memory, or assign vacant RAM space for data buffers. Program and data memory in microcomputers may occupy widely scattered addresses to simplify the hardware.

Typical ORIGIN statements are:

```
ORG     RESET
ORG     1000
ORG     INT3
```

Some assemblers assume an origin of zero if the programmer does not put an ORG statement at the start of the program. The convenience is slight; we recommend the inclusion of an ORG statement to avoid confusion.

THE RESERVE PSEUDO-OPERATION

The RESERVE pseudo-operation allows the programmer to allocate RAM for various purposes such as data tables, temporary storage, indirect addresses, a Stack, etc.

```
ALLOCATING
RAM
```

Using the RESERVE pseudo-operation, you assign a name to the memory area and declare the number of locations to be assigned. Here are some examples:

NOKEY	RESERVE	1
TEMP	RESERVE	50
VOLTG	RESERVE	80
BUFR	RESERVE	100

You can use the RESERVE pseudo-operation to reserve memory locations in program memory or in data memory; however the nature of the RESERVE pseudo-operation is more meaningful when applied to data memory.

In reality, all the RESERVE pseudo-operation does is increase the assembler's Location Counter by the amount declared in the operand field. The assembler does not actually produce any object code.

Note the following features of RESERVE:

1) The label of the RESERVE pseudo-operation is assigned the value of the first address reserved. For example, the sequence:

	ORG	3000
BUF1	RESERVE	100
BUF2	RESERVE	50
VOLTS	RESERVE	5

assigns to the label BUF1 the value 3000, to BUF2 3100, and to VOLTS 3150.

2) You must specify the number of locations to be reserved. There is no default case.

3) No data is placed into the reserved locations. Any data that, by chance, may be in these locations will be left there.

Some assemblers allow the programmer to place initial values in RAM. We strongly recommend that you do not use this feature — it assumes that the program (along with

INITIALIZING RAM

the initial values) will be loaded from an external device (e.g., paper tape or floppy disk) each time it is run. Most microprocessor programs, on the other hand, reside in non-volatile ROM and start when power comes on. The RAM in such situations does not retain its contents, nor is it reloaded. Always include instructions to initialize the RAM in your program.

LINKING PSEUDO-OPERATIONS

We often want statements in one program or subroutine to use names that are defined elsewhere. Such names are called external references; a special linker program is necessary to ac-

EXTERNAL REFERENCES

tually fill in the external values and determine if any names are undefined or doubly defined.

The pseudo-operation EXTERNAL, usually abbreviated EXT, signifies that the name is defined elsewhere.

The pseudo-operation ENTRY, usually abbreviated ENT, signifies that the name is available for use elsewhere, i.e., it is defined in this program.

The precise way in which linking pseudo-operations are implemented varies greatly from assembler to assembler. We will not refer to such **pseudo-operations** again, but they **are very useful in actual applications.**

HOUSEKEEPING PSEUDO-OPERATIONS

There are various housekeeping pseudo-operations, which affect the operation of

the assembler and its program listing rather than the output program itself. Common housekeeping pseudo-operations include:

1) END. which marks the end of the assembly language source program.

2) LIST. which tells the assembler to print the source program. Some assemblers allow such variations as NO LIST or LIST SYMBOL TABLE to avoid long. repetitive listings.

3) NAME or TITLE. which prints a name at the top of each page of the listing.

4) PAGE or SPACE. which skips to the next page or next line. respectively. and improves the appearance of the listing. making it easier to read.

5) PUNCH. which transfers subsequent object code to the paper tape punch. This pseudo-operation may in some cases be the default option and therefore unnecessary.

LABELS WITH PSEUDO-OPERATIONS

Users often wonder if or when they can assign a label to a pseudo-operation. These are our recommendations:

1) All EQUATE pseudo-operations must have labels; they do not make any sense otherwise. since their purpose is to define the meaning of the labels.

2) DATA and RESERVE pseudo-operations usually have labels. The label identifies the first memory location used or assigned.

3) Other pseudo-operations should not have labels. Some assemblers allow other pseudo-operations to have labels. but the meaning of the labels varies. We recommend that you avoid this practice.

ADDRESSES AND THE OPERAND FIELD

Most assemblers allow the programmer a lot of freedom in describing the contents of the Operand Address field. But remember, the assembler has built-in names for registers and instructions and may have other built-in names.

Some common options for the operand field are:

1) Decimal numbers

Most assemblers assume all numbers to be decimal unless they are marked otherwise. So:

> DECIMAL
> DATA OR
> ADDRESSES

ADD   100

means "add the contents of memory location 100 decimal to the contents of the Accumulator"

2) Other number systems

> OTHER
> NUMBER
> SYSTEMS

Most assemblers will also accept binary. octal. or hexadecimal entries. But you must identify these number systems in some way. e.g., by preceding or following the number with an identifying character or letter. Here are some common identifiers:

B or % for binary

O. Q. C or @ for octal (we avoid O because of the confusion with zero).

H or $ for hexadecimal

D for decimal. D may be omitted: it is the default case.

Assemblers generally require hexadecimal numbers to start with a decimal digit (e.g., 0A36 instead of A36) in order to distinguish between numbers and names or labels. It is good practice to enter numbers in the base in which their meaning is the clearest — i.e., decimal constants in decimal; addresses and BCD numbers in hexadecimal; masking patterns or bit outputs in binary if they are short and in hexadecimal if they are long.

3) Symbolic names

Names can appear in the operand field; they will be treated as the data that they represent. But remember, **there is a difference between data and addresses.** The sequence:

```
FIVE     EQU     5
         ADD     FIVE
```

will add the contents of memory location 5 (not necessarily the number 5) to the contents of the Accumulator.

4) The current value of the location counter (usually referred to as * or $).

This is useful mainly in Jump instructions; for example:

```
JUMP     $+6
```

causes a Jump to the memory location six words beyond the word that contains the first byte of the JUMP instruction:
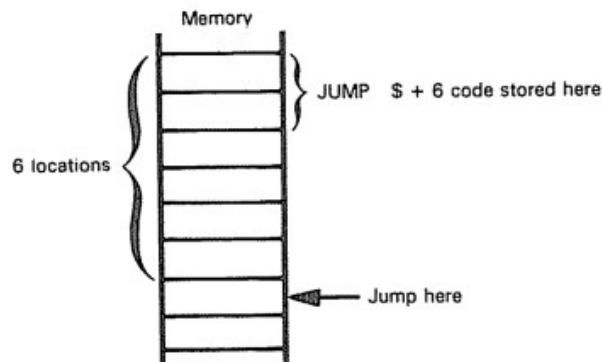
Most microprocessors have many two and three-word instructions. Thus, you will have difficulty determining exactly how far apart two assembly language statements are. Therefore, using offsets from the Location Counter frequently results in errors that you can avoid if you use labels.

5) Character codes

Most assemblers allow text to be entered as ASCII strings. Such strings may be surrounded either with single or double quotation marks; strings may also use a beginning or ending symbol such as A or C. A few assemblers also permit EBCDIC strings.

ASCII CHARACTERS

We recommend that you use character strings for all text. It improves the clarity and readability of the program.

6) Combinations of 1) through 5) with arithmetic, logical, or special operators.

Almost all assemblers allow simple arithmetic combinations such as START+1. Some assemblers also permit multiplication, division, logical functions, shifts, etc. These are referred to as expressions. Note that the assembler evaluates expressions at assembly time. Even though an expression in the operand field may involve multiplication, you

ARITHMETIC AND LOGICAL EXPRESSIONS

may not be able to use multiplication in the logic of your own program — unless you write a subroutine for that specific purpose.

Assemblers vary in what expressions they accept and how they interpret them. Complex expressions make a program difficult to read and understand.

We have made some recommendations during this section but will repeat them and add others here. In general, **the user should emphasize clarity and simplicity.** There is no payoff for being an expert in the intricacies of assemblers or in having the most complex expression on the block. **We suggest the following approach:**

1) Use the clearest number system or character code for data. Masks and BCD numbers in decimal, ASCII characters in octal, or ordinary numerical constants in hexadecimal serve no purpose and therefore should not be used.
2) Remember to distinguish data and addresses.
3) Don't use offsets from the Location Counter.
4) Keep expressions simple and obvious. Don't rely on obscure features of the assembler.

CONDITIONAL ASSEMBLY

Some assemblers allow you to include or exclude parts of the source program, depending on conditions existing at assembly time. This is called conditional assembly; it gives the assembler some of the flexibility of a compiler. **Most microcomputer assemblers have limited capabilities for conditional assembly. A usual form is:**

                IF COND

                .CONDITIONAL PROGRAM

                .

                ENDIF

If the expression COND is true at assembly time, the instructions between IF and ENDIF (two pseudo-operations) are included in the program.

Typical uses of conditional assembly are:

1) To include or exclude extra variables.
2) To place diagnostics or special conditions in test runs.
3) To allow data of various bit lengths.
4) To create specialized versions of a common program.

Unfortunately, conditional assembly tends to clutter programs and make them difficult to read. Use conditional assembly only if it is necessary.

MACROS

You will often find that particular sequences of instructions occur many times in a source program. Repeated instruction sequences may reflect the needs of your program logic, or they may be compensating for deficiencies in your microprocessor's

> **DEFINING A SEQUENCE OF INSTRUCTIONS**

instruction set. You can avoid repeatedly writing out the same instruction sequence by using a macro.

Macros allow you to assign a name to an instruction sequence. You then use the macro name in your source program instead of the repeated instruction sequence.

The assembler will replace the macro name with the appropriate sequence of instructions. This may be illustrated as follows:

Source Program	Object Program

```
MAC1    MACRO        (macro definition)
        instruction M1 ⎫
        instruction M2 ⎬
        instruction M3 ⎭

        ENDM         (end of macro definition)

        instruction P1  (main program) ⎫      instruction P1
        instruction P2                  ⎬      instruction P2
        instruction P3                  ⎭      instruction P3

                                               instruction M1
        MAC1 ──────────────────────────────    instruction M2
                                               instruction M3

        instruction P4  ⎫                      instruction P4
        instruction P5  ⎬                      instruction P5
        instruction P6  ⎬                      instruction P6
        instruction P7  ⎭                      instruction P7

                                               instruction M1
        MAC1 ──────────────────────────────    instruction M2
                                               instruction M3

        instruction P8  ⎫                      instruction P8
        instruction P9  ⎭                      instruction P9

                                               instruction M1
        MAC1 ──────────────────────────────    instruction M2
                                               instruction M3

        instruction P10 ⎫                      instruction P10
        instruction P11 ⎭                      instruction P11
            •                                      •
            •                                      •
            •                                      •
```

Macros are not the same as subroutines. A subroutine occurs once in a program, and program execution branches to the subroutine. A macro is expanded to an actual instruction sequence each time the macro occurs; thus a macro does not cause any branching.

Macros have the following advantages:

ADVANTAGES OF MACROS

1) Shorter source programs.
2) Better program documentation.
3) Use of debugged instruction sequences — once the macro has been debugged, you are sure of an error-free instruction sequence every time you use the macro.
4) Easier changes. Change the macro definition and the assembler makes the change for you every time the macro is used.
5) Inclusion of commands, keywords, or other computer instructions in the basic instruction set. You use the macro as an extension of your instruction set.

The disadvantages of macros are:

DISADVANTAGES OF MACROS

1) Repetition of the same instruction sequences since the macro is expanded every time it is used.

2) A single macro may create a lot of instructions.

3) Lack of standardization that may make the program difficult to read and understand.

4) Possible effects on registers and flags that may not be clearly stated.

One problem is that variables used in a macro are known only within it (i.e., they are local rather than global). This can often create a great deal of confusion without any gain in return. You should be aware of this problem when using macros.

LOCAL OR
GLOBAL
VARIABLES

COMMENTS

All assemblers allow you to place comments in a source program. Comments have no effect on the object code, but they help you to read, understand, and document the program. Good commenting is an essential part of writing assembly language programs; without comments, programs are very difficult to understand.

We will discuss commenting along with documentation in a later chapter, but here are some guidelines:

COMMENTING
TECHNIQUES

1) Use comments to tell what the program is doing, not what instructions do.

   Comments should say things like "IS TEMPERATURE ABOVE LIMIT?", "LINE FEED TO TTY", or "EXAMINE LOAD SWITCH".

   Comments should not say things like "ADD 1 TO ACCUMULATOR" "JUMP TO START", or "LOOK AT CARRY". You should describe how the program is affecting the system; internal effects on the CPU are seldom of any interest.

2) Keep comments brief and to the point. Details should be available elsewhere in the documentation.

3) Comment all key points.

4) Do not comment standard instructions or sequences that change counters and pointers; pay special attention to instructions that may not have an obvious meaning.

5) Do not use obscure abbreviations.

6) Make the comments neat and readable.

7) Comment all definitions. describing their purposes. Also mark all tables and data storage areas.

8) Comment sections of the program as well as individual instructions.

9) Be consistent in your terminology. You can (should) be repetitive; you do not need to consult a thesaurus.

10) Leave yourself notes at points which you find confusing, e.g.. "REMEMBER CARRY WAS SET BY LAST INSTRUCTION". You may drop these in the final documentation.

A well-commented program is easy to work with. You will recover the time spent in commenting many times over. We will try to show good commenting style in the programming examples. although we often over-comment for instructional purposes.

TYPES OF ASSEMBLERS

Although all assemblers perform the same tasks, their implementations vary greatly. We will not try to describe all the existing types of assemblers; we will merely define the terms and indicate some of the choices.

A cross-assembler is an assembler that runs on a computer other than the one for which it assembles object programs.

CROSS-ASSEMBLER

The computer on which the cross-assembler runs is typically a large computer with extensive software support and fast peripherals — such as an IBM 360 or 370, a Univac 1108, or a Burroughs 6700. The computer for which the cross-assembler assembles programs is typically a microcomputer like the Z80 or MC6800. Most cross-assemblers are written in FORTRAN so that they are portable.

A self-assembler or resident assembler is an assembler that runs on the computer for which it assembles programs. The self-assembler will require some memory and peripherals, and it may run quite slowly.

RESIDENT ASSEMBLER

A macroassembler is an assembler that allows you to define sequences of instructions as macros.

MACRO-ASSEMBLER

A microassembler is an assembler used to write the microprograms that define the instruction set of a computer. Microprogramming has nothing specifically to do with microcomputers.

MICRO-ASSEMBLER

A meta-assembler is an assembler that can handle many different instruction sets. The user must define the particular instruction set being used.

META-ASSEMBLER

A one-pass assembler is an assembler that goes through the assembly language program only once. Such an assembler must have some way of resolving forward references, e.g., Jump instructions which use labels that appear later in the source program, i.e., that have not yet been defined.

ONE-PASS ASSEMBLER

A two-pass assembler is an assembler that goes through the assembly language source program twice. The first time the assembler simply collects and defines all the symbols; the

TWO-PASS ASSEMBLER

second time it replaces the references with the actual definitions. A two-pass assembler solves most of the forward reference problems. However, macro expansion and conditional assembly can cause problems. On some large machines seven or more passes are needed to insure that all forward references are resolvable. A two-pass assembler may be quite slow if no backup storage (like a floppy disk) is available; then the assembler must physically read the program twice from a slow input medium (like a teletypewriter paper tape reader). Most microprocessor-based assemblers require two passes.

ERRORS

Assemblers normally provide error messages, often consisting of a single coded letter. Some typical errors are:

1) Undefined name (often a misspelling or an omitted definition).
2) Illegal character (e.g., a 2 in a binary number).

3) Illegal format (wrong delimiter or incorrect operands).

4) Invalid expression (e.g.. two operators in a row).

5) Illegal value (usually too large).

6) Missing operand.

7) Double definition (i.e.. two different values assigned to one name).

8) Illegal label (e.g.. a label on a pseudo-operation that cannot have one).

9) Missing label.

10) Undefined operation code.

In interpreting assembler errors, you must remember that the assembler may get off on the wrong track if it finds a stray letter, an extra space, or incorrect punctuation. Many assemblers will then proceed to misinterpret the succeeding instructions and produce meaningless error messages. Always look at the first error very carefully; subsequent ones may depend on it. Caution and consistent adherence to standard formats will eliminate many annoying mistakes.

LOADERS

The loader is the program which actually takes the output (object code) from the assembler and places it in memory. Loaders range from the very simple to the very complex. We will describe a few different types.

A bootstrap loader is a program that uses its own first few instructions to load the rest of itself or another loader program into memory. The bootstrap loader may be in ROM. or you may

| BOOTSTRAP |
| LOADER |

have to enter it into the computer memory using front panel switches. The assembler may place a bootstrap loader at the start of the object program that it produces.

A relocating loader can load programs anywhere in memory. It typically loads each program into the memory space immediately following that used by the previous program. The programs,

| RELOCATING |
| LOADER |

however. must themselves be capable of being moved around in this way, i.e.. they must be relocatable. An absolute loader, in contrast. will always place the programs in the same area of memory.

A linking loader loads programs and subroutines that have been separately assembled; it resolves external references — that is. an instruction in one module that refers to a label in

| LINKING |
| LOADERS |

another module. Object programs loaded by a linking loader must be created by an assembler that permits and marks external references.

An alternative approach is to separate the linking and loading functions and have the linking performed by a program called a link editor.

| LINK |
| EDITOR |

REFERENCES

A complete monograph on macros is M. Campbell-Kelly, An Introduction to Macros. American Elsevier, New York, 1973.

Microprogramming is described conceptually in An Introduction to Microcomputers: Volume 1 — Basic Concepts, Chapter 4. A more technical description is in A.K. Agrawala and T.G. Rauscher, Foundations of Microprogramming, Academic Press, New York, 1976.

You can find more detailed descriptions of assemblers and loaders in D.W. Barron, "Assemblers and Loaders", American Elsevier, New York, 1972 and in C.W. Gear, Computer Organization and Programming, McGraw-Hill, New York, 1974.

Chapter 3
THE Z80 ASSEMBLY LANGUAGE
INSTRUCTION SET

We are now ready to start writing assembly language programs. We begin in this chapter by defining the individual instructions of the Z80 assembly language instruction set, plus the syntax rules of the Zilog assembler.

We do not discuss any aspects of microcomputer hardware, signals, interfaces, or CPU architecture in this book. This information is described in detail in An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors and Volume 3 — Some Real Support Devices, while Z80 Programming for Logic Design discusses assembly language as an extension of digital logic. In this book. **we look at programming techniques from the assembly language programmer's viewpoint, where pins and signals are irrelevant and there are no important differences between a minicomputer and a microcomputer.**

Interrupts, direct memory access, and the Stack architecture for the Z80 will be described in later chapters of this book, in conjunction with assembly language programming discussions of the same subjects.

This chapter contains a detailed definition of each assembly language instruction. These definitions are identical to those found in Chapter 6 of Z80 Programming for Logic Design.

The detailed description of individual instructions is preceded by a general discussion of the Z80 instruction set that divides instructions into those which are commonly used, infrequently used, and rarely used. If you are an experienced assembly language programmer, this categorization is not particularly important — and, depending on your own programming prejudices, it may not even be accurate. If you are a novice assembly language programmer, we recommend that you begin by writing programs using only instructions in the "commonly used" category. Once you have mastered the concepts of assembly language programming, you may examine other instructions and use them where appropriate.

CPU REGISTERS AND STATUS FLAGS

The CPU registers and status flags for the Z80 may be illustrated as follows:

```
                                                    ┌──── Sign
                                                    │ ┌── Zero
                                                    │ │ ┌── Auxiliary Carry
                                                    │ │ │ ┌── Panty/Overflow
                                                    │ │ │ │ ┌── Subtract
                                                    │ │ │ │ │ ┌── Carry
                                                    ▼ ▼ ▼ ▼ ▼ ▼

                                        | S | Z |  |Ac|  |P/o| N | C |  Flags
                                        |        A            |  Accumulator
        ┌──  |    B    |    C    |                               Secondary Data Counters
Secondary│   |    D    |    E    |
Accumulators |    H    |    L    |                               Primary Data Counter
                                        | S'| Z'|  |Ac'| |P/o'| N'| C'|  Alternate Flags
                                        |        A'           |  Alternate Accumulator
         ┌── |   B'    |   C'    |                               Alternate Secondary
Alternate│   |   D'    |   E'    |                               Data Counters
Secondary│   |   H'    |   L'    |                               Alternate Primary Data Counter
Accumulators
             |          SP          |                           Stack Pointer
             |          PC          |                           Program Counter
             |          IX          |                           Index Register
             |          IY          |                           Index Register
                                    |     I     |               Interrupt Vector Register
                                    |     R     |               Refresh Register
```

The Accumulator is the primary source and destination for one-operand and two-operand instructions. For example, the shortest and fastest data transfers between the CPU and I/O devices are performed through the Accumulator. In addition, more Memory Reference instructions move data between the Accumulator and memory than between any other register and memory. All 8-bit arithmetic and Boolean instructions take one of the operands from the Accumulator and return the result to the Accumulator. An instruction must therefore **load the Accumulator before the Z80 can perform any 8-bit arithmetic or Boolean operations.**

The B, C, D, E, H, and L registers are all secondary registers. Data stored in any of these six registers may be accessed with equal ease; such data can be moved to any other register or can be used as the second operand in two-operand instructions.

There are, however, some important differences in the functions of Registers B, C, D, E, H, and L.

Registers H and L are the primary Data Pointer for the Z80. That is to say, you will normally use these two registers to hold the 16-bit memory address of data being accessed. Data may be transferred between any registers and the memory location addressed by H and L. Since HL is the primary Data Pointer, it often takes fewer bytes of object code and less instruction cycles to perform operations with it. The Z80 programmer should try to address data memory via Registers H and L whenever possible.

Within your program logic, always reserve Registers H and L to hold a data memory address.

Registers B, C, D, and E provide secondary data storage; frequently, the second operand for two-operand instructions is stored in one of these four registers. (The first operand is stored in the Accumulator, which is also the destination for the result.)

There are **a limited number of instructions** that **treat Registers B and C, or D and E, as 16-bit Data Pointers.** But these instructions move data between memory and the Accumulator only.

In your program logic you should normally use Registers B, C, D, and E as temporary storage for data or addresses.

Registers IX and IY are index registers. They provide a limited indexing capability of the type described in An Introduction to Microcomputers: Volume 1 for short instructions.

The alternate registers F', A', B', C', D', E', H', and L' provide a duplicate set of general purpose registers. Just two single-byte Exchange instructions select and deselect all alternate registers; one instruction exchanges AF and the alternate AF' as a register pair, and one instruction exchanges BC, DE, and HL with the alternate BC', DE', and HL'. Once selected, all subsequent register operations are performed on the active set until the next exchange selects the inactive set. **The alternate registers can be reserved for use when a fast interrupt response is required.** Or, they may be used in any desired way by the programmer.

There are a number of instructions that handle 16 bits of data at a time. These instructions refer to pairs of CPU registers as follows:

High-order byte		Low-order byte
F	and	A
B	and	C
D	and	E
H	and	L
F'	and	A'
B'	and	C'
D'	and	E'
H'	and	L'

The combination of the Accumulator and flags, treated as a 16-bit unit, is used only for Stack operations and alternate register switches. Arithmetic operations access B and C, D and E, or H and L as 16-bit data units.

The Carry status flag holds carries out of the most significant bit in any arithmetic operation. The Carry flag is also included in Shift instructions; it is reset by Boolean instructions.

The Subtract flag is designed for internal use during decimal adjust operations. This flag is set to 1 for all Subtract instructions and reset to 0 for all Add instructions.

The Parity/Overflow flag is a multiple use flag, depending on the operation being performed. For arithmetic operations, it is an overflow flag. For input, rotate, and Boolean operations, it is a parity flag, with 1 = even parity and 0 = odd parity. During block transfer and search operations, it remains set until the byte counter decrements to zero; then it is reset to zero. It is also set to the current state of the interrupt enable flip-flop (IFF2) when a LD A,I or LD A,R instruction is executed.

The Zero flag is set to 1 when any arithmetic or Boolean operation generates a zero result. The Zero status is set to 0 when such an operation generates a non-zero result.

The Sign status flag acquires the value of the most significant bit of the result following the execution of any arithmetic or Boolean instruction.

The Auxiliary Carry status flag holds any carry from bit 3 to 4 resulting from the execution of an arithmetic instruction. The purpose of this status flag is to simplify Binary-Coded-Decimal (BCD) operations; this is the standard use of an Auxiliary Carry status flag as described in An Introduction to Microcomputers: Volume 1. Chapter 3.

All of the above status flags keep their current value until an instruction that modifies them is executed. Merely changing the value of the Accumulator will not necessarily change the value of the status flags. For example, if the Zero flag is set, and a load immediate to the Accumulator is executed, that causes the Accumulator to acquire a non-zero value; the value of the Zero flag remains unchanged.

The 16-bit Stack Pointer allows you to implement a Stack anywhere in addressable memory. The size of the Stack is limited only by the amount of addressable memory present. In reality you will rarely use more than 256 bytes of memory for your Stack. You should use the Stack for accessing subroutines and processing interrupts. Do not use the Stack to pass parameters to subroutines. This is not very efficient within the limitations of the Z80 instruction set. The Z80 Stack is started at its highest address. A Push decrements the Stack Pointer contents; a Pop increments the Stack Pointer contents.

The Interrupt Vector register and the Refresh register are special-purpose registers not normally used by the programmer.

The Interrupt Vector register is used to store the page address of an interrupt response routine; the location on the page is provided by the interrupting device. This scheme allows the address of the interrupt response routine to be changed while still providing a very fast response time for the interrupting device.

The Refresh register contains a memory refresh counter in the low-order seven bits. This counter is incremented automatically after each instruction fetch and provides the next refresh address for dynamic memories. The high-order bit of the Refresh register will remain set or reset, depending on how it was loaded at the last LD R,A instruction.

Z80 MEMORY ADDRESSING MODES

The Z80 provides extensive addressing modes. These include:

- **Implied**
- **Implied Block Transfer with Auto-Increment/Decrement**
- **Implied Stack**
- **Indexed**
- **Direct**
- **Program Relative**
- **Base Page**
- **Register Indirect**
- **Immediate**

Implied

In implied memory addressing, the H and L registers hold the address of the memory location being accessed. Data may be moved between the identified memory location and any one of the seven CPU registers A, B, C, D, E, H, or L. For example, the instruction

LD C,(HL)

loads the C register with the contents of the memory location currently pointed to by HL. This is illustrated as follows:
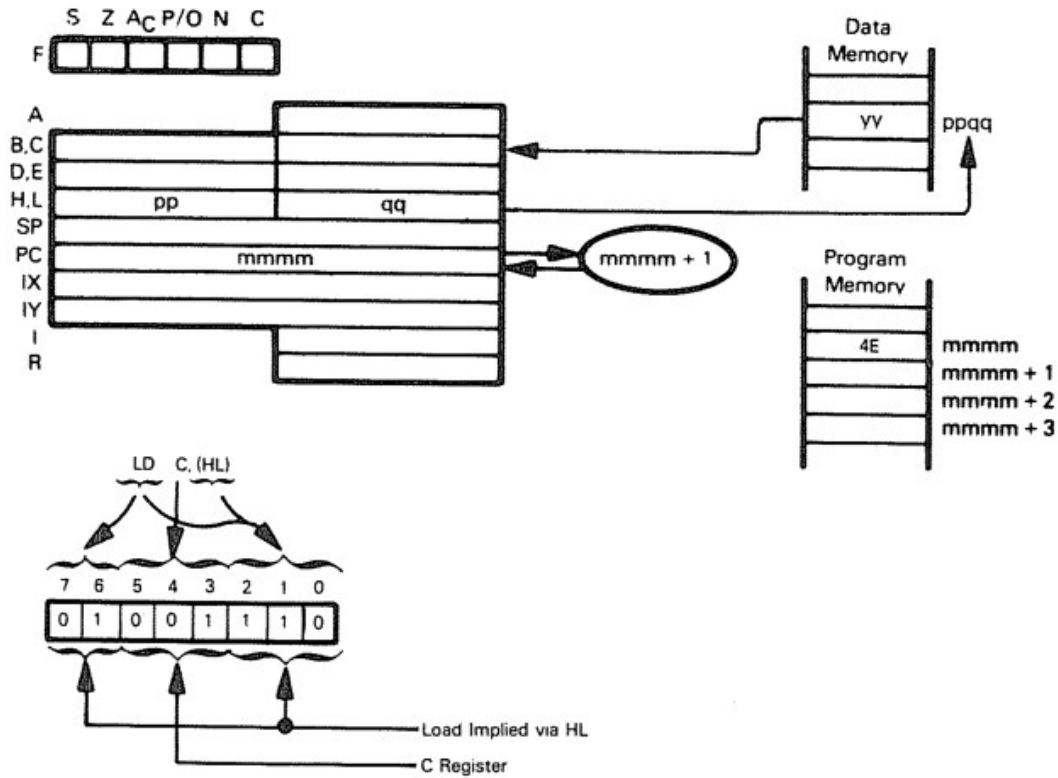
A limited number of instructions use Registers B and C or D and E as the Data Pointer. These instructions move data between the Accumulator and the memory location addressed by Registers B and C or Registers D and E. The instruction

LD   (BC),A

stores the contents of A into the memory location currently addressed by Register Pair BC. This is illustrated as follows:
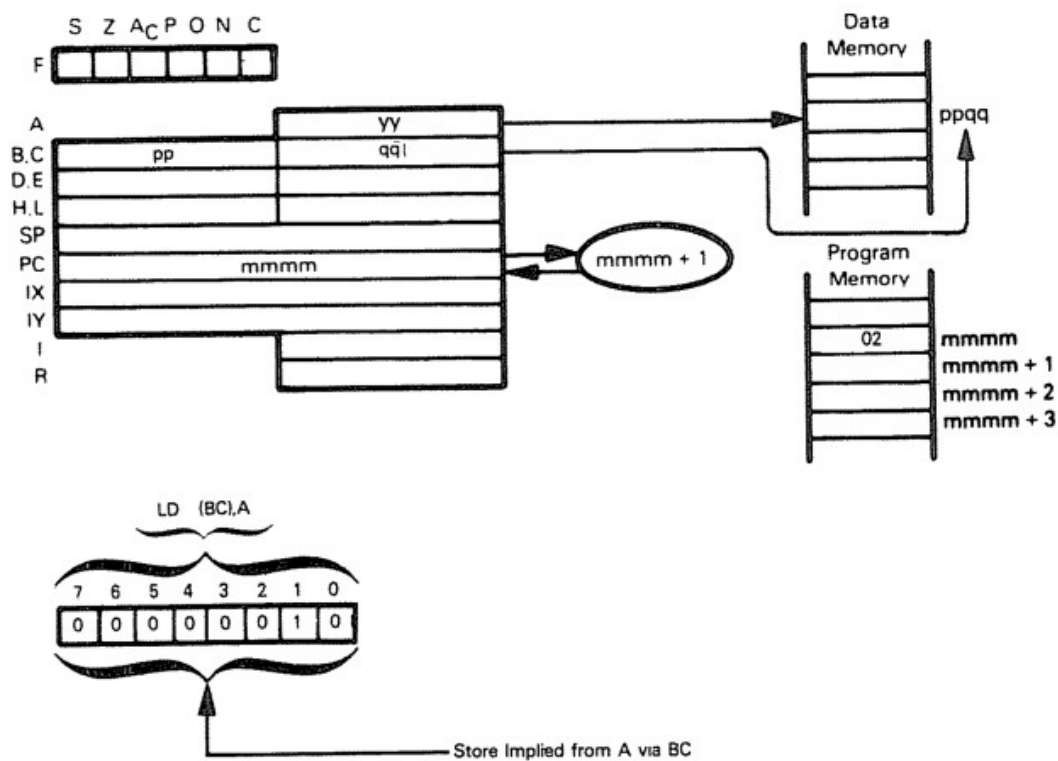
Implied Block Transfer With Auto-Increment/Decrement

Block Transfer and Search instructions operate on a block of data whose size is set by the programmer as the contents of the BC register pair. In this form of addressing, a byte of data is moved from the memory location addressed by HL to the memory location addressed by DE; then HL and DE are incremented and BC is decremented. Data transfer continues until BC reaches zero, at which point the instruction is terminated. Variations include allowing other instructions to follow each data transfer, with the programmer supplying the loopback; auto-decrementing HL and DE instead of auto-incrementing; and a complementary set of Block Search instructions that compare the memory byte addressed by HL with the contents of the A register, setting a flag if a match is found.

The Load, Increment, and Repeat instruction

<div align="center">LDIR</div>

is illustrated as follows:
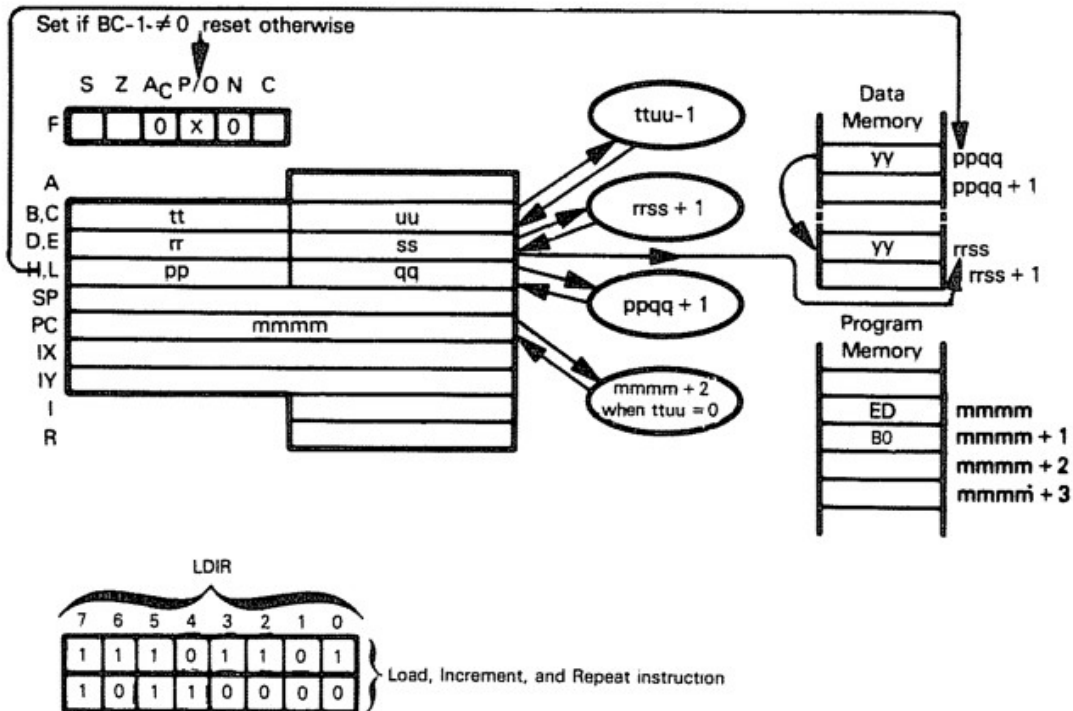
A similar group of Input/Output instructions is provided, allowing a block of data to be input or output between memory and an I/O device. The I/O port number is taken as the contents of the C register, with the single B register used as the byte counter. Memory is addressed by HL.

Implied Stack

Since the Stack is part of Read/Write memory, we must consider Stack instructions as Memory Reference instructions. **Push and Pop instructions move two bytes of data between a register pair and the addressed Stack Pointer location,** i.e., current top-of-stack. The Z80 Stack address is decremented with each Push and incremented with each Pop. The instruction

PUSH   DE

is illustrated as follows:
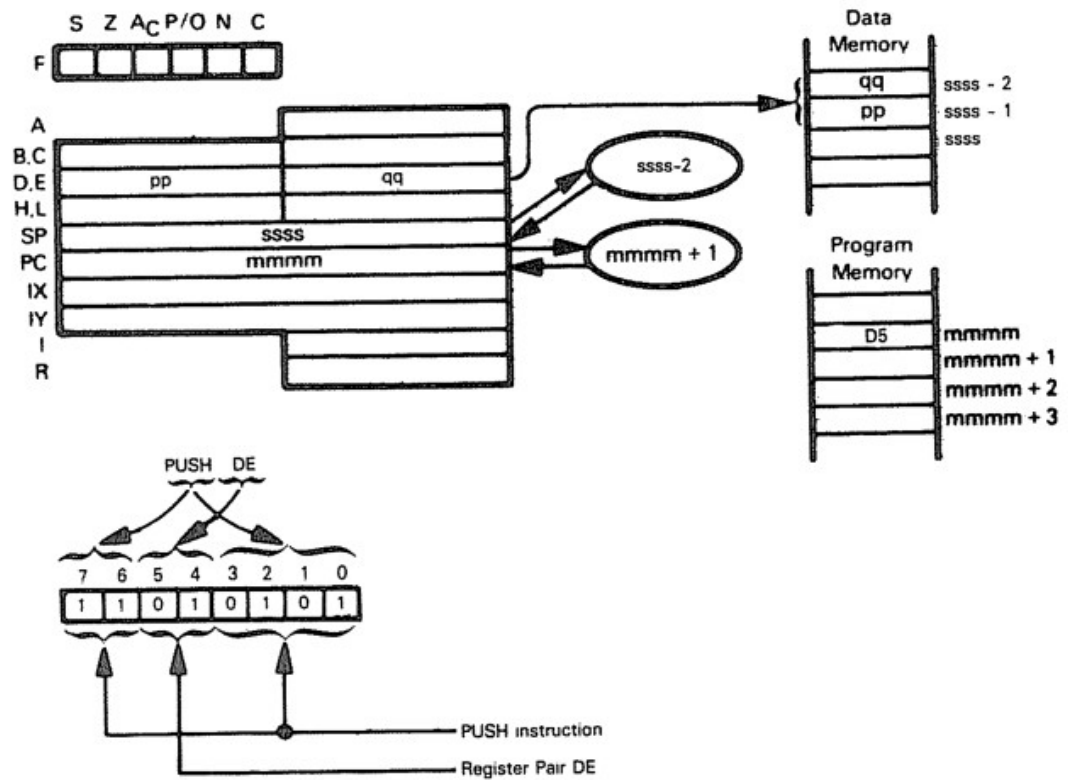
The Z80 also has instructions that exchange the two top-of-stack bytes with a 16-bit register — HL or one of the two index registers. The instruction

$$\text{EX} \quad \text{(SP),HL}$$

is illustrated as follows:
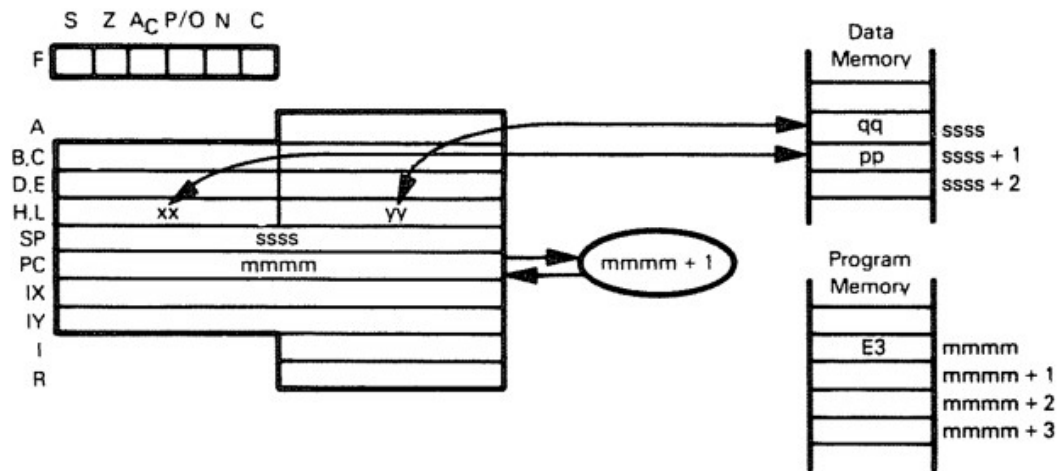
Indexed

The Z80 has two 16-bit index registers, called IX and IY. They may be used interchangeably. All memory reference operations for which (HL) can be specified can alternatively be specified as an indexed operation. The difference between implied addressing using HL and indexed addressing using IX and IY is that **the index operand includes a displacement value that is added to the index address.** In the instruction

$$\text{ADD} \quad \text{A,(IX+40H)}$$

the memory address is the sum of the contents of the IX register and 40_{16}. This may be illustrated as follows:
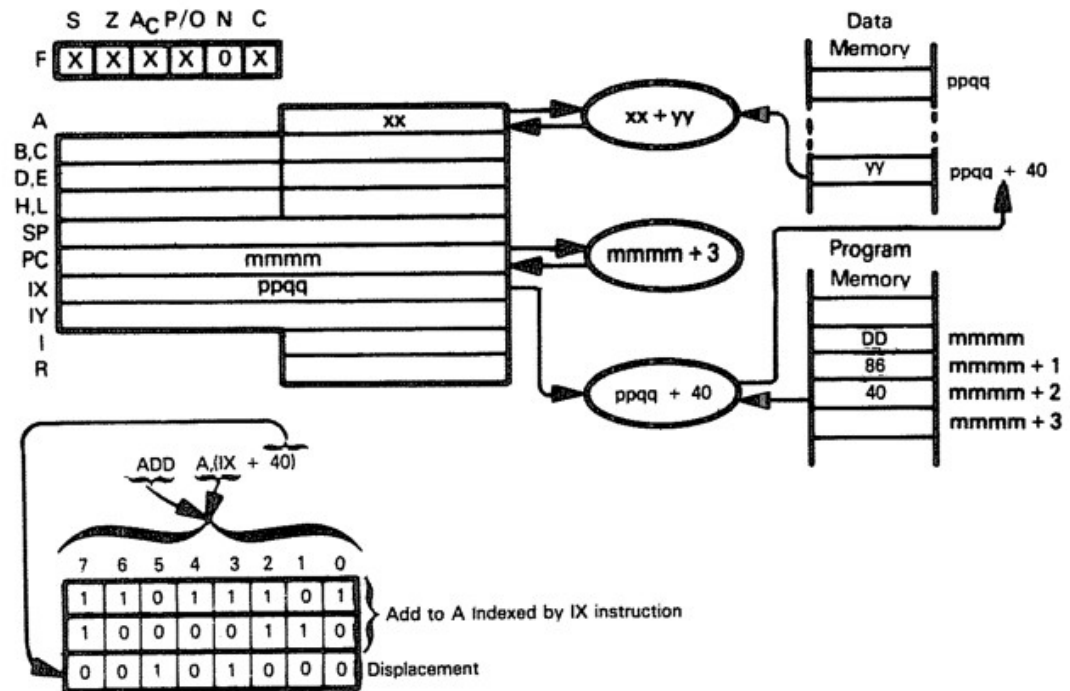
Direct

Direct addressing can be used to load the Accumulator with any 8-bit value from memory, load BC, DE, HL, SP, IX, or IY with any 16-bit memory value, and jump or call subroutines direct at any memory location. The 16-bit direct address is stored in the last two bytes of the instruction, in low-byte high-byte order (this is the reverse of the standard high-low scheme).

The instruction

LD   A,(NETX)

loads the A register with the contents of the memory location addressed by the label NETX. The instruction

LD   HL,(1FFH)

loads the L register with the contents of memory location $01FF_{16}$ and the H register with the contents of memory location 0200_{16}. This may be illustrated as follows:
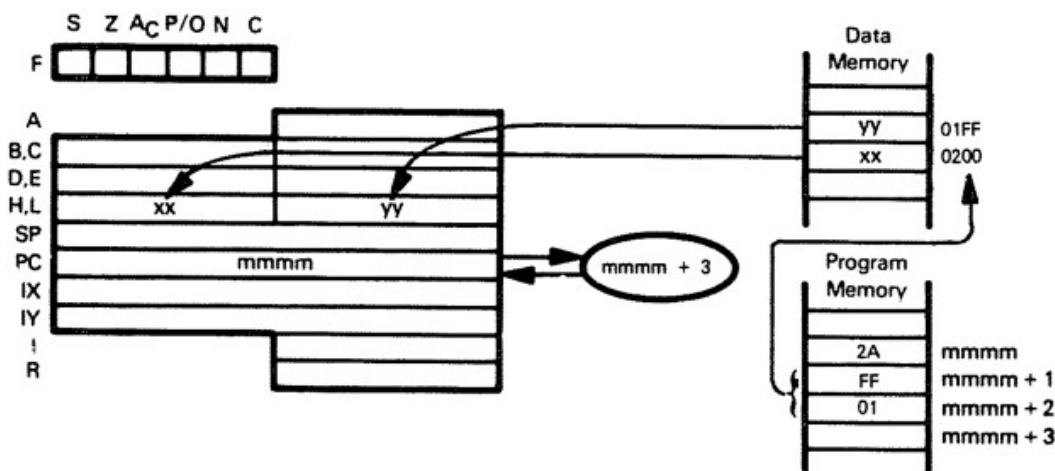
LD   HL,(1FFH)

7	6	5	4	3	2	1	0	
0	0	1	0	1	0	1	0	Load HL Direct instruction
1	1	1	1	1	1	1	1	Direct address - low byte
0	0	0	0	0	0	0	1	Direct address - High byte

The direct Jump instructions provide jumps and jumps-to-subroutines, both unconditional and conditional. These are all 3-byte instructions, with the direct address stored in the second and third bytes of the instruction, as shown above for Load Direct.

There are three additional addressing modes used by Z80 Branch instructions: program relative, base page, and register indirect. In general, they are shorter and/or faster than direct jumps but may have more limited addressing capabilities.

Program Relative

Jump Relative instructions provide program relative addressing in the range -126, +129 bytes from the first byte of the Program Relative instruction. These instructions are all 2-byte instructions, with the signed displacement value stored in the second byte of the instruction. **There are unconditional and conditional relative jumps, as well as a Decrement and Jump If Not Zero instruction (DJNZ) that facilitates loop control.**

Given the instruction

JR   SRCH

assume that SRCH is a label addressing a location $5A_{16}$ bytes up in memory from the JR op-code byte. The operation may be illustrated as follows:
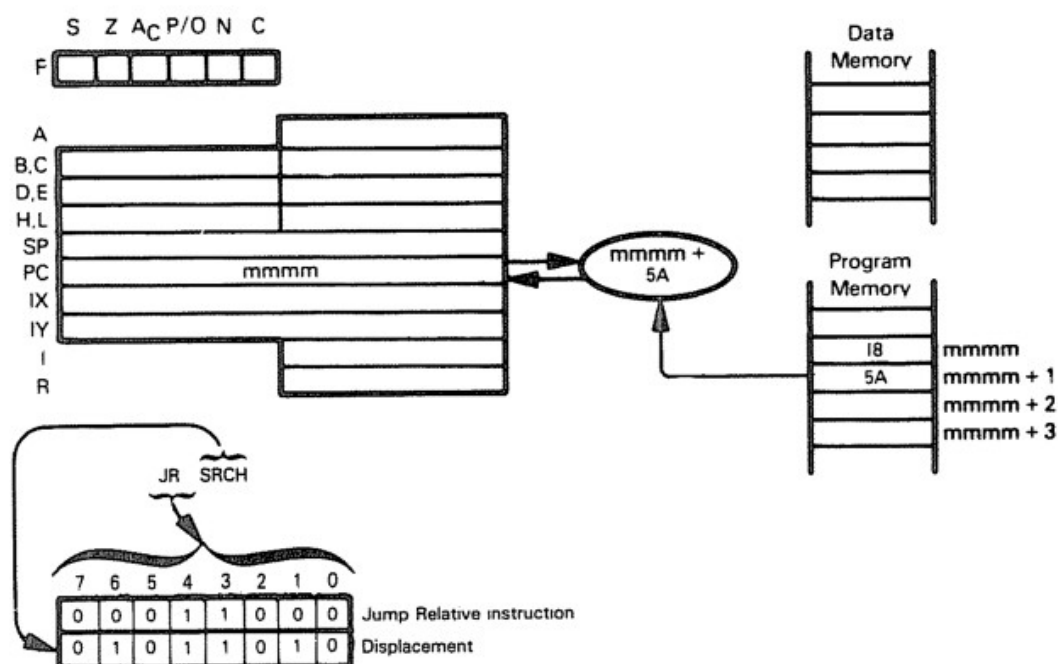
Base Page

The Z80 has a **modified base page addressing** mode for the Restart instruction. This is a special Call instruction that **allows a single-byte instruction to jump to one of eight subroutines located at specific points in lower core.** The effective address is calculated from a 3-bit code stored in the instruction, as follows:

Lower Core Address	3-Bit Code
00H	000
08H	001
10H	010
18H	011
20H	100
28H	101
30H	110
38H	111

The decoded address value is loaded into the low-order byte of the Program Counter; the high-order byte of the Program Counter is set to zero. For example, the instruction

RST   00H

is illustrated as follows:

3-13
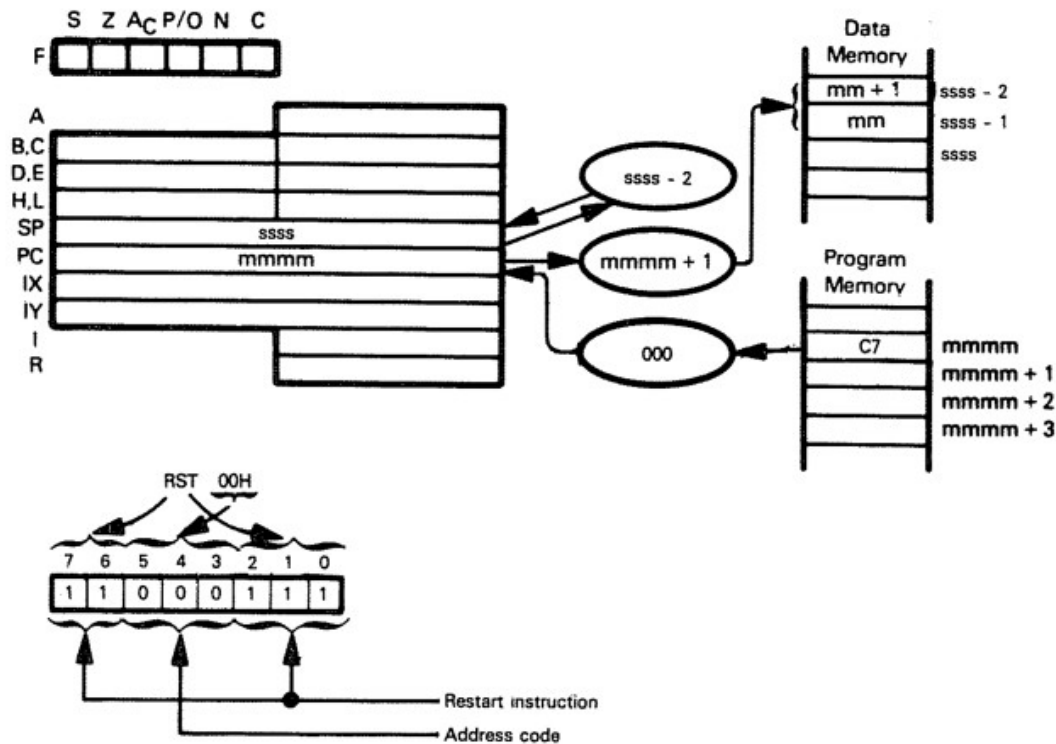
Register Indirect

In standard indirect addressing, a memory location contains the effective address, and the instruction specifies the address of the memory location containing the effective address. In register indirect addressing, a register contains the effective address, and the instruction specifies which of the registers contains the effective address. Note that for a Load, for instance, this is just another way of describing implied addressing. However, **the Z80 has Jump instructions that allow a jump to the memory location whose address is contained in the specified register.** This is a form of indirect addressing, and is described separately because, while most microcomputers have implied addressing, very few have register indirect jumps.

The instruction

$$JP \quad (HL)$$

directs that a jump is to be taken to the memory location whose address is contained in HL. This may be illustrated as follows:

3-14
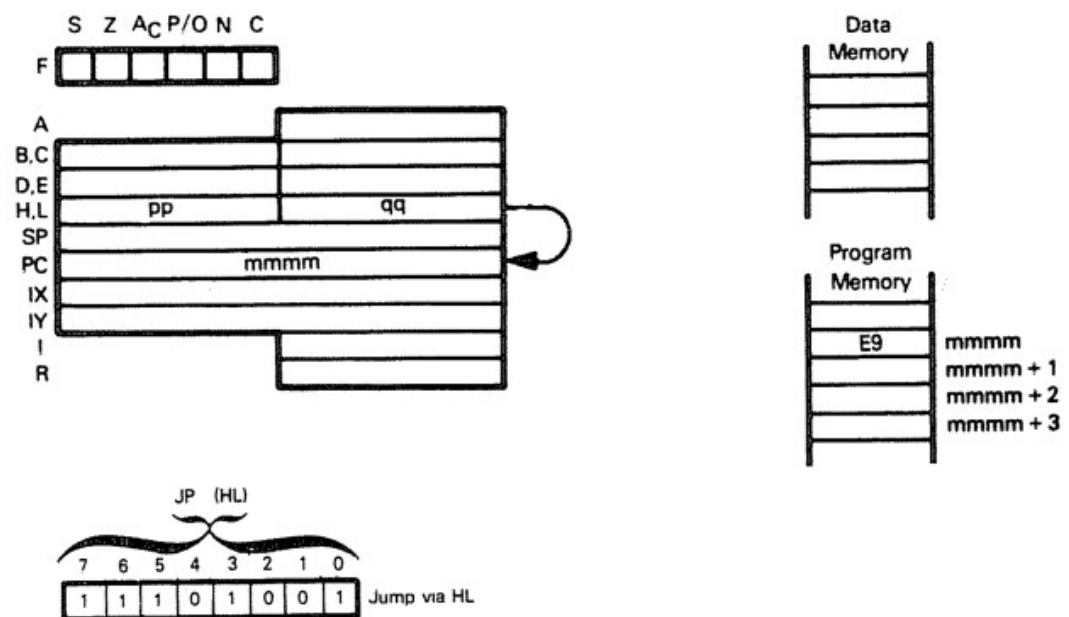
Immediate

Some texts identify Immediate instructions as Memory Reference instructions. An Immediate instruction is a 2-, 3-, or 4-byte instruction in which the last one or two bytes hold fixed data that is loaded into a register or memory location. **The Z80 provides Immediate instructions to:**

- **load 8-bit data into any of the 8-bit registers,**
- **load 16-bit data into any of the register pairs or 16-bit registers,**
- **store 8-bit data into any memory location using implied or indexed addressing,**
- **perform arithmetic and logical operations using the Accumulator and 8-bit immediate data.**

The instruction

LD BC,0BCH

loads the immediate data value BC_{16} into Register Pair BC. This may be illustrated as follows:
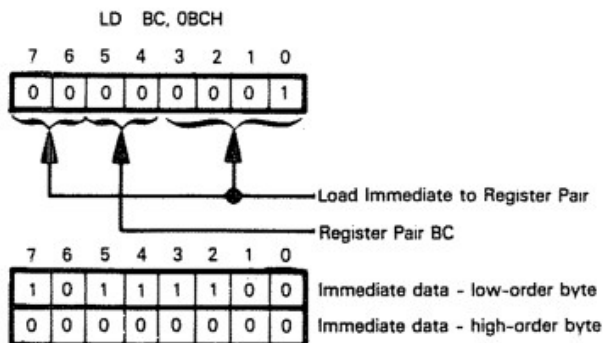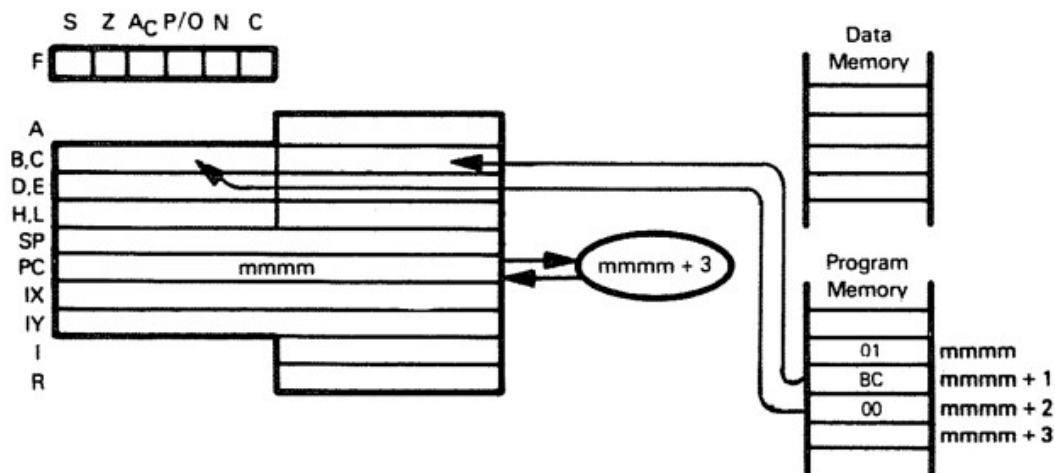
Table 3-1. Frequently Used Instructions of the Z80

Instruction Code	Meaning
ADC A	Add with Carry to Accumulator
ADD	Add
AND	Logical AND
CALL addr	Call Subroutine
CALL cond,addr	Call Conditional
CP	Compare
DEC	Decrement
DJNZ	Decrement and Jump If Not Zero
IN	Input
INC	Increment
JR	Jump Relative
JR cond,addr	Jump Relative Conditional
LD reg,(HL)	Load Register
LD A,(addr)	Load Accumulator Direct
LD data	Load Immediate
LD (HL),reg	Store Register
LD (addr),A	Store Accumulator Direct
LD dst,src	Move Register-to-Register
OUT	Output
POP	Pop from Stack
PUSH	Push to Stack
RET	Return from Subroutine
RET cond	Return Conditional
RLA	Rotate Accumulator Left Through Carry
RRA	Rotate Accumulator Right Through Carry
SLA	Shift Left Arithmetic
SRL	Shift Right Logical
SUB	Subtract

Table 3-2. Occasionally Used Instructions of the Z80

Instruction Code		Meaning
BIT		Test Bit
CPD. CPDR		Compare, Decrement, (Repeat)
CPI. CPIR		Compare, Increment, (Repeat)
CPL		Complement Accumulator
DAA		Decimal Adjust Accumulator
DI		Disable Interrupts
EI		Enable Interrupts
EX		Exchange
HALT		Halt
IND. INDR		Input, Decrement, (Repeat)
INI. INIR		Input, Increment, (Repeat)
JP	addr	Jump
JP	cond,addr	Jump Conditional
LD	A,(BC) or (DE)	Load Accumulator Secondary
LD	HL,(addr)	Load HL Direct
LD	reg,(xy+disp)	Load Register Indexed
LD	rp,(addr)	Load Register Pair Direct
LD	xy,(addr)	Load Index Register Direct
LD	(BC) or (DE),A	Store Accumulator Secondary
LD	(addr),HL	Store HL Direct
LD	(xy+disp),reg	Store Register Indexed
LD	(addr),rp	Store Register Pair Direct
LD	(addr),xy	Store Index Register Direct
LD	(HL),data	Store Immediate to Memory
LD	(xy+disp),data	Store Immediate to Memory Indexed
LDD. LDDR		Load, Decrement, (Repeat)
LDI. LDIR		Load, Increment, (Repeat)
NEG		Negate (Twos Complement) Accumulator
NOP		No Operation
OR		Logical OR
OUTD. OTDR		Output, Decrement, (Repeat)
OUTI. OTIR		Output, Increment, (Repeat)
RES		Reset Bit
RETI		Return from Interrupt
RL		Rotate Left Through Carry
RLC		Rotate Left Circular
RLCA		Rotate Accumulator Left Circular
RR		Rotate Right Through Carry
RRC		Rotate Right Circular
RRCA		Rotate Accumulator Right Circular
SET		Set Bit
SRA		Shift Right Arithmetic
XOR		Logical Exclusive OR

Table 3-3. Seldom Used Instructions of the Z80

Instruction Code	Meaning
ADC HL,rp	Add Register Pair with Carry to HL
CCF	Complement Carry Flag
EXX	Exchange Register Pairs and Alternatives
IM n	Set Interrupt Mode
RETN	Return from Non-Maskable Interrupt
RLD	Rotate Accumulator and Memory Left Decimal
RRD	Rotate Accumulator and Memory Right Decimal
RST	Restart
SBC	Subtract with Carry (Borrow)
SCF	Set Carry Flag
LD A,I	Load Accumulator from Interrupt Vector Register
LD A,R	Load Accumulator from Refresh Register
LD I,A	Store Accumulator to Interrupt Vector Register
LD R,A	Store Accumulator to Refresh Register
LD SP,HL	Move HL to Stack Pointer
LD SP,xy	Move Index Register to Stack Pointer

ABBREVIATIONS

These are the abbreviations used in this chapter:

A,F,B,C,D,E,H,L	The 8-bit registers. A is the Accumulator and F is the Flag Word.
AF',BC',DE',HL'	The alternate register pairs
addr	A 16-bit memory address
x(b)	Bit b of 8-bit register or memory location x
cond	Condition for program branching. Conditions are:
	NZ - Non-Zero (Z = 0)
	Z - Zero (Z = 1)
	NC - Non-carry (C = 0)
	C - Carry (C = 1)
	PO - Parity Odd (P = 0)
	PE - Parity Even (P = 1)
	P - Positive Sign (S = 0)
	M - Negative Sign (S = 1)
data	An 8-bit binary data unit
data16	A 16-bit binary data unit
disp	An 8-bit signed binary address displacement
xx(HI)	The high-order 8 bits of a 16-bit quantity xx
I	Interrupt Vector register (8 bits)
IX IY	The Index registers (16 bits each)
label	A 16-bit instruction memory address
xx(LO)	The low-order 8 bits of a 16-bit quantity xx
LSB	Least Significant Bit (Bit 0)
MSB	Most Significant Bit (Bit 7)
PC	Program Counter
port	An 8-bit I/O port address

pr	Any of the following register pairs:
	BC
	DE
	HL
	AF

R	The Refresh register (8 bits)

reg	Any of the following registers:
	A
	B
	C
	D
	E
	H
	L

rp	Any of the following register pairs:
	BC
	DE
	HL
	SP

SP	Stack Pointer (16 bits)

xy	Either one of the Index registers (IX or IY)

Object Code	bbb	Bit number 000 (LSB) to 111 (MSB)
	ccc	Condition code	000 = non-zero
			001 = zero
			010 = no carry
			011 = carry
			100 = parity odd
			101 = parity even
			110 = positive sign
			111 = negative sign
	ddd	Destination register — same coding as rrr
	ppqq	A 16-bit memory address
	rrr	Register	111 = A
			000 = B
			001 = C
			010 = D
			011 = E
			100 = H
			101 = L
	sss	Source register — same coding as rrr
	x	Index register	0 = IX
			1 = IY
	xx	Register pair	00 = BC
			01 = DE
			10 = HL
			11 = SP (rp) or AF (pr)
	xxx	Restart code (000 to 111)
	yy	An 8-bit binary data unit
	yyyy	A 16-bit binary data unit

Statuses

The Z80 has the following status flags:

C	- Carry status
Z	- Zero status
S	- Sign status
P/O	- Parity/Overflow status
A_C	- Auxiliary Carry status
N	- Subtract status

The following symbols are used in the status columns:

X	- flag is affected by operation
(blank)	- flag is not affected by operation
1	- flag is set by operation
0	- flag is reset by operation
U	- flag is unknown after operation
P	- flag shows parity status
O	- flag shows overflow status
I	- flag shows interrupt enabled/disabled status

[[ ]]   Memory addressing: 1)   the contents of the memory location whose address is contained in the designated register. 2)   an I/O port whose address is contained in the designated register.

[ ]   The contents of a register or memory location.

For example:

$$[[HL]] \leftarrow [[HL]] + 1$$

indicates that the contents of the memory location addressed by the contents of HL are incremented. whereas:

$$[HL] \leftarrow [HL] + 1$$

indicates that the contents of the HL register itself are incremented.

Λ Logical AND

V Logical OR

$\underline{V}$ Logical Exclusive-OR

$\leftarrow$ Data is transferred in the direction of the arrow

$\leftarrow \rightarrow$ Data is exchanged between the two locations designated on either side of the arrows.

INSTRUCTION MNEMONICS

Table 3-4 summarizes the Z80 instruction set. The MNEMONIC column shows the instruction mnemonic (IN, OUT, LD). The OPERAND column shows the operands, if any, used with the instruction mnemonic.

The fixed part of an assembly language instruction is shown in UPPER CASE. The variable part (immediate data, I/O device number, register name, label or address) is shown in lower case.

For closely related operands, each type is listed separately without repeating the mnemonic. For instance, examples of the format entry

        LD rp,(addr)
            xy,(addr)

are: LD BC,(DAT2)
        LD IX,(MEM)

INSTRUCTION OBJECT CODES

The object code and instruction length in bytes are shown in Table 3-4 for each instruction variation. Table 3-5 lists the object codes in numerical order.

For instruction bytes without variations, object codes are represented as two hexadecimal digits (e.g., 3F).

For instruction bytes with variations in one of the two digits, the object code is shown as one 4-bit binary digit and one hexadecimal digit (e.g., 11 x 1 D) in Table 3-5. For other instruction bytes with variations, the object code is shown as eight binary digits (e.g., 01sss001).

INSTRUCTION EXECUTION TIMES

Table 3-4 lists the instruction execution times in clock periods. Real time can be obtained by dividing the given number of clock periods by the clock frequency. For example, for an instruction that requires 7 clock periods, a 4 MHz clock will result in a 1.75 microsecond execution time.

When two possible execution times are shown (i.e., 5/11), it indicates that the number of clock periods depends on condition flags. The first time is for "condition not met," whereas the second is for "condition met."

STATUS

The six status flags are stored in the Flag register (F) as follows:

| S | Z | | Aᴄ | | P/O | N | C |

These bits are not used
Carry status (carry out of bit 7)
Subtract status
(1 after subtract operation, 0 otherwise)
Parity/Overflow
(for logical operations, 1 for even, 0 for odd parity.
For arithmetic, 1 for overflow)
Auxiliary Carry status (carry out of bit 3)
Zero status (1 for zero, 0 for nonzero)
Sign status (value of bit 7)
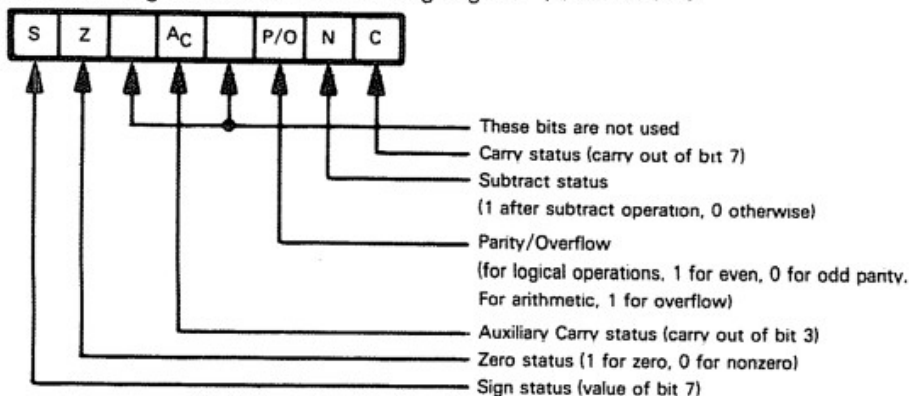
In the individual instruction descriptions, the effect of instruction execution on status is illustrated as follows:

S   Z A_C P/O   N   C

| ? | | 1 | X | 0 | X |

Modified to reflect results of execution
Unconditionally reset to 0
Unconditionally set to 1
Unchanged
Unknown
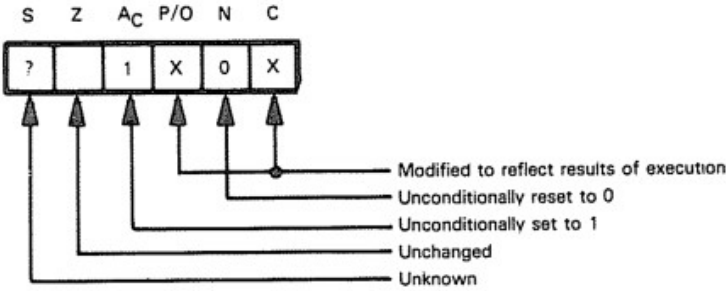
An X identifies a status that is set or reset. A 0 identifies a status that is always cleared. A 1 identifies a status that is always set. A blank means the status does not change. A question mark (?) means the status is not known.

STATUS CHANGES WITH INSTRUCTION EXECUTION

Table 3-4 A Summary of the Z80 Instruction Set

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	Status C	Z	S	P/O	AC	N	Operation Performed:
	IN	A,(port)	DB yy	2	10							[A] ← [port] Input to Accumulator from directly addressed I/O port. Address Bus: A0-A7: port A8-A15: [A]
	IN	reg,(C)	ED 01ddd000	2	11		X	X	P	X	0	[reg] ← [[C]] Input to register from I/O port addressed by the contents of C ** If second byte is 70_{16} only the flags will be affected
	INIR		ED B2	2	20/15**		1	?	?	?	1	Repeat until [B] = 0: [[HL]] ← [[C]] [B] ← [B] - 1 [HL] ← [HL] + 1 Transfer a block of data from I/O port addressed by contents of C to memory location addressed by contents of HL, going from low addresses to high Contents of B serve as a count of bytes remaining to be transferred. **
O/I	INDR		ED BA	2	20/15**		1	?	?	?	1	Repeat until [B] = 0: [[HL]] ← [[C]] [B] ← [B] - 1 [HL] ← [HL] - 1 Transfer a block of data from I/O port addressed by contents of C to memory location addressed by contents of HL, going from high addresses to low Contents of B serve as a count of bytes remaining to be transferred **
	INI		ED A2	2	15		X	?	?	?	1	[[HL]] ← [[C]] [B] ← [B] - 1 [HL] ← [HL] + 1 Transfer a byte of data from I/O port addressed by contents of C to memory location addressed by contents of HL Decrement byte count and increment destination address **

**Address Bus: A0–A7: [C]
A8–A15: [B]

Table 3-4 A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	C	Z	S	P/O	AC	N	Operation Performed
I/O (Continued)	IND		ED AA	2	15		X	?	?	?	1	[[HL]] ← [[C]] [B] ← [B] - 1 [HL] ← [HL] - 1 Transfer a byte of data from I/O port addressed by contents of C to memory location addressed by contents of HL. Decrement both byte count and destination address **
	OUT	(port),A	D3 yy	2	11							[port] ← [A] Output from Accumulator to directly addressed I/O port Address Bus: A0–A7: port A8–A15: [A]
	OUT	(C),reg	ED 01sss001	2	12							[[C]] ← [reg] Output from register to I/O port addressed by the contents of C **
	OTIR		ED B3	2	20/15**		1	?	?	?	1	Repeat until [B] = 0: [[C]] ← [[HL]] [B] ← [B] - 1 [HL] ← [HL] + 1 Transfer a block of data from memory location addressed by contents of HL to I/O port addressed by contents of C, going from low memory to high. Contents of B serve as a count of bytes remaining to be transferred **
	OTDR		ED BB	2	20/15**		1	?	?	?	1	Repeat until [B] = 0: [[C]] ← [[HL]] [B] ← [B] - 1 [HL] ← [HL] - 1 Transfer a block of data from memory location addressed by contents of HL to I/O port addressed by contents of C, going from high memory to low. Contents of B serve as a count of bytes remaining to be transferred **

Table 3-4 A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	Status						Operation Performed
						C	Z	S	P/O	Ac	N	
I/O (Continued)	OUTI		ED A3	2	15		X	?	?	?	1	[[C]] ← [[HL]] [B] ← [B] - 1 [HL] ← [HL] + 1 Transfer a byte of data from memory location addressed by contents of HL to I/O port addressed by contents of C Decrement byte count and increment source address **
	OUTD		ED AB	2	15		X	?	?	?	1	[[C]] ← [[HL]] [B] ← [B] - 1 [HL] ← [HL] - 1 Transfer a byte of data from memory location addressed by contents of HL to I/O port addressed by contents of C Decrement both byte count and source address **
Primary Memory Reference	LD	A,(addr)	3A ppqq	3	13							[A] ← [addr] Load Accumulator from directly addressed memory location
	LD	HL,(addr)	2A ppqq	3	16							[H] ← [addr + 1], [L] ← [addr] Load HL from directly addressed memory
	LD	rp,(addr) xy,(addr)	ED 01xx1011 ppqq 11x11101 2A ppqq	4 4	20 20							[rp(HI)] ← [addr + 1], [rp(LO)] ← [addr] or [xy(HI)] ← [addr + 1], [xy(LO)] ← [addr] Load register pair or Index register from directly addressed memory
	LD	(addr),A	32 ppqq	3	13							[addr] ← [A] Store Accumulator contents in directly addressed memory location
	LD	(addr),HL	22 ppqq	3	16							[addr + 1] ← [H], [addr] ← [L] Store contents of HL to directly addressed memory location
	LD	(addr),rp (addr),xy	ED 01xx0011 ppqq 11x11101 22 ppqq	4 4	20 20							[addr + 1] ← [rp(HI)], [addr] ← [rp(LO)] or [addr + 1] ← [xy(HI)], [addr] ← [xy(LO)] Store contents of register pair or Index register to directly addressed memory.
	LD	A,(BC) A,(DE)	0A 1A	1 1	7 7							[A] ← [[BC]] or [A] ← [[DE]] Load Accumulator from memory location addressed by the contents of the specified register pair

Table 3-4 A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	Status C	Z	S	P/O	AC	N	Operation Performed
Primary Memory Reference (Continued)	LD	reg,(HL)	01ddd110	1	7							[reg] — [[HL]] Load register from memory location addressed by contents of HL
	LD	(BC),A (DE),A	02 12	1 1	7 7							[[BC]] — [A] or [[DE]] — [A] Store Accumulator to memory location addressed by the contents of the specified register pair
	LD	(HL),reg	01110sss	1	7							[[HL]] — [reg] Store register contents to memory location addressed by the contents of HL.
	LD	reg,(xy+disp)	11x11101 01ddd110 disp	3	19							[reg] — [[xy] + disp] Load register from memory location using base relative addressing
	LD	(xy+disp),reg	11x11101 01110sss disp	3	19							[[xy] + disp] — [reg] Store register to memory location addressed relative to contents of Index register.
Block Transfer and Search	LDIR		ED B0	2	20/16**				0	0	0	Repeat until [BC] = 0: [[DE]] — [[HL]] [DE] — [DE] + 1 [HL] — [HL] + 1 [BC] — [BC] - 1 Transfer a block of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE, going from low addresses to high. Contents of BC serve as a count of bytes to be transferred
	LDDR		ED B8	2	20/16**				0	0	0	Repeat until [BC] = 0: [[DE]] — [[HL]] [DE] — [DE] - 1 [HL] — [HL] - 1 [BC] — [BC] - 1 Transfer a block of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE, going from high addresses to low. Contents of BC serve as a count of bytes to be transferred.

Table 3-4 A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	Status						Operation Performed
						C	Z	S	P/O	A_C	N	
Block Transfer and Search (Continued)	LDI		ED A0	2	16				X	0	0	$[[DE]] \leftarrow [[HL]]$ $[DE] \leftarrow [DE] + 1$ $[HL] \leftarrow [HL] + 1$ $[BC] \leftarrow [BC] - 1$ Transfer one byte of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE Increment source and destination addresses and decrement byte count.
	LDD		ED A8	2	16				X	0	0	$[[DE]] \leftarrow [[HL]]$ $[DE] \leftarrow [DE] - 1$ $[HL] \leftarrow [HL] - 1$ $[BC] \leftarrow [BC] - 1$ Transfer one byte of data from the memory location addressed by the contents of HL to the memory location addressed by the contents of DE Decrement source and destination addresses and byte count
	CPIR		ED B1	2	20/16**		X	X	X	X	1	Repeat until $[A] = [[HL]]$ or $[BC] = 0$: $[A] - [[HL]]$ (only flags are affected) $[HL] \leftarrow [HL] + 1$ $[BC] \leftarrow [BC] - 1$ Compare contents of Accumulator with those of memory block addressed by contents of HL, going from low addresses to high. Stop when a match is found or when the byte count becomes zero
	CPDR		ED B9	2	20/16**		X	X	X	X	1	Repeat until $[A] = [[HL]]$ or $[BC] = 0$: $[A] - [[HL]]$ (only flags are affected) $[HL] \leftarrow [HL] - 1$ $[BC] \leftarrow [BC] - 1$ Compare contents of Accumulator with those of memory block addressed by contents of HL, going from high addresses to low Stop when a match is found or when the byte count becomes zero

Table 3-4 A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	C	Z	S	P/O	A_C	N	Operation Performed
Block Transfer and Search (Continued)	CPI		ED A1	2	16		x	x	x	x	1	$[A] - [[HL]]$ (only flags are affected) $[HL] \rightarrow [HL] + 1$ $[BC] \rightarrow [BC] - 1$ Compare contents of Accumulator with those of memory location addressed by contents of HL. Increment address and decrement byte count
	CPD		ED A9	2	16		x	x	x	x	1	$[A] - [[HL]]$ (only flags are affected) $[HL] \rightarrow [HL] - 1$ $[BC] \rightarrow [BC] - 1$ Compare contents of Accumulator with those of memory location addressed by contents of HL. Decrement address and byte count
Secondary Memory Reference	ADD	A,(HL) A,(xy+disp)	86 11x11101 86 disp	1 3	7 19	x	x	x	0	x	0	$[A] \rightarrow [A] + [[HL]]$ or $[A] \rightarrow [A] + [[xy] + disp]$ Add to Accumulator using implied addressing or base relative addressing
	ADC	A,(HL) A,(xy+disp)	8E 11x11101 8E disp	1 3	7 19	x	x	x	0	x	0	$[A] \rightarrow [A] + [[HL]] + C$ or $[A] \rightarrow [A] + [[xy] + disp] + C$ Add with Carry using implied addressing or base relative addressing
	SUB	(HL) (xy + disp)	96 11x11101 96 disp	1 3	7 19	x	x	x	0	x	1	$[A] \rightarrow [A] - [[HL]]$ or $[A] \rightarrow [A] - [[xy] + disp]$ Subtract from Accumulator using implied addressing or base relative addressing
	SBC	A,(HL) A,(xy+disp)	9E 11x11101 9E disp	1 3	7 19	x	x	x	0	x	1	$[A] \rightarrow [A] - [[HL]] - C$ or $[A] \rightarrow [A] - [[xy] + disp] - C$ Subtract with Carry using implied addressing or base relative addressing
	AND	(HL) (xy + disp)	A6 11x11101 A6 disp	1 3	7 19	0	x	x	P	1	0	$[A] \rightarrow [A] \wedge [[HL]]$ or $[A] \rightarrow [A] \wedge [[xy] + disp]$ AND with Accumulator using implied addressing or base relative addressing
	OR	(HL) (xy + disp)	B6 11x11101 B6 disp	1 3	7 19	0	x	x	?	1	0	$[A] \rightarrow [A] \vee [[HL]]$ or $[A] \rightarrow [A] \vee [[xy] + disp]$ OR with Accumulator using implied addressing or base relative addressing

Table 3-4. A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	C	Z	S	P/O	Ac	N	Operation Performed
Secondary Memory Reference (Continued)	XOR	(HL)	AE	1	7	0	×	×	P	1	0	[A] ← [A] ∀ [[HL]] or [A] ← [A] ∀ [[xy] + disp] Exclusive-OR with Accumulator using implied addressing or base relative addressing
		(xy + disp)	11x11101 AE disp	3	19							
	CP	(HL)	BE	1	7	×	×	×	O	×	1	[A] - [[HL]] or [A] - [[xy] + disp] Compare with Accumulator using implied addressing or base relative addressing. Only the flags are affected
		(xy + disp)	11x11101 BE disp	3	19							
	INC	(HL)	34	1	11	×	×	×	O	×	0	[[HL]] ← [[HL]] + 1 or [[xy] + disp] ← [[xy] + disp] + 1 Increment using implied addressing or base relative addressing
		(xy + disp)	11x11101 34 disp	3	23							
	DEC	(HL)	35	1	11	×	×	×	O	×	1	[[HL]] ← [[HL]] - 1 or [[xy] + disp] ← [[xy] + disp] - 1 Decrement using implied addressing or base relative addressing
		(xy + disp)	11x11101 35 disp	3	23							
Memory Shift and Rotate	RLC	(HL)	CB 06	2	15	×	×	×	P	0	0	[[HL]] or [[xy] + disp] Rotate contents of memory location (implied or base relative addressing) left with branch Carry
		(xy + disp)	11x11101 CB disp 06	4	23							
	RL	(HL)	CB 16	2	15	×	×	×	P	0	0	[[HL]] or [[xy] + disp] Rotate contents of memory location left through Carry.
		(xy + disp)	11x11101 CB disp 16	4	23							
	RRC	(HL)	CB 0E	2	15	×	×	×	P	0	0	[[HL]] or [[xy] + disp] Rotate contents of memory location right with branch Carry
		(xy + disp)	11x11101 CB disp 0E	4	23							
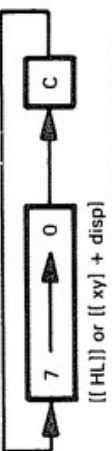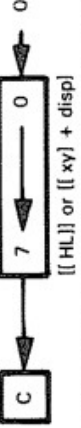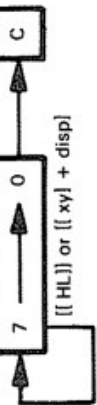
Table 3-4 A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	C	Z	S	P/O	AC	N	Operation Performed
Memory Shift and Rotate (Continued)	RR	(HL)	CB 1E	2	15	X	X	X	P	0	0	[[HL]] or [[xy] + disp]. Rotate contents of memory location right through Carry
		(xy + disp)	11x11101 CB disp / 1E	4	23							
	SLA	(HL)	CB 26	2	15	X	X	X	P	0	0	[[HL]] or [[xy] + disp]. Shift contents of memory location left and clear LSB (Arithmetic Shift)
		(xy + disp)	11x11101 CB disp / 26	4	23							
	SRA	(HL)	CB 2E	2	15	X	X	X	P	0	0	[[HL]] or [[xy] + disp]. Shift contents of memory location right and preserve MSB (Arithmetic Shift)
		(xy + disp)	11x11101 CB disp / 2E	4	23							
	SRL	(HL)	CB 3E	2	15	X	X	X	P	0	0	[[HL]] or [[xy] + disp]. Shift contents of memory location right and clear MSB (Logical Shift)
		(xy + disp)	11x11101 CB disp / 3E	4	23							

3-30

Table 3-4. A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	Status						Operation Performed
						C	Z	S	P/O	A_C	N	
Immediate	LD	reg,data	00ddd110 yy	2	7							[reg] ← data Load immediate into register
	LD	rp data16	00xx0001 yyyy	3	10							[rp] ← data16 or [xy] ← data16 Load 16 bits of immediate data into register pair or Index register
	LD	xy,data16	11x11101 21 yyyy	4	14							
	LD	(HL),data	36 yy	2	10							[[HL]] ← data or [[xy] + disp] ← data
		(xy + disp), data	11x11101 36 disp yy	4	19							Load immediate into memory location using implied or base relative addressing
Jump	JP	label	C3 ppqq	3	10							[PC] ← label Jump to instruction at address represented by label
	JR	disp	18 (disp-2)	2	12							[PC] ← [PC] + 2 + (disp-2) Jump relative to present contents of Program Counter
	JP	(HL)	E9	1	4							[PC] ← [HL] or [PC] ← [xy] Jump to address contained in HL or Index register
		(xy)	11x11101 E9	2	8							
Subroutine Call and Return	CALL	label	CD ppqq	3	17							[[SP] - 1] ← [PC(HI)] [[SP] - 2] ← [PC(LO)] [SP] ← [SP] - 2 [PC] ← label Jump to subroutine starting at address represented by label
	CALL	cond,label	11ccc100 ppqq	3	10/17							Jump to subroutine if condition is satisfied; otherwise, continue in sequence.
	RET		C9	1	10							[PC(LO)] ← [[SP]] [PC(HI)] ← [[SP] + 1] [SP] ← [SP] + 2 Return from subroutine
	RET	cond	11ccc000	1	5/11							Return from subroutine if condition is satisfied; otherwise, continue in sequence

Table 3-4. A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	Status						Operation Performed
						C	Z	S	P/O	AC	N	
Immediate Operate	ADD	A,data	C6 yy	2	7	X	X	X	O	X	0	$[A] \leftarrow [A]$ + data Add immediate to Accumulator
	ADC	A,data	CE yy	2	7	X	X	X	O	X	0	$[A] \leftarrow [A]$ + data + C Add immediate with Carry
	SUB	data	D6 yy	2	7	X	X	X	O	X	1	$[A] \leftarrow [A]$ - data Subtract immediate from Accumulator
	SBC	A,data	DE yy	2	7	X	X	X	O	X	1	$[A] \leftarrow [A]$ - data - C Subtract immediate with Carry
	AND	data	E6 yy	2	7	0	X	X	P	1	0	$[A] \leftarrow [A] \wedge$ data AND immediate with Accumulator
	OR	data	F6 yy	2	7	0	X	X	P	1	0	$[A] \leftarrow [A] \vee$ data OR immediate with Accumulator
	XOR	data	EE yy	2	7	0	X	X	P	1	0	$[A] \leftarrow [A] \veebar$ data Exclusive-OR immediate with Accumulator
	CP	data	FE yy	2	7	X	X	X	O	X	1	$[A]$ - data Compare immediate data with Accumulator contents; only the flags are affected
Jump on Condition	JP	cond,label	11ccc010 ppqq	3	10							If cond, then $[PC] \leftarrow$ label Jump to instruction at address represented by label if the condition is true
	JR	C,disp	38 (disp-2)	2	7/12							If C = 1, then $[PC] \leftarrow [PC] + 2 + (disp - 2)$ Jump relative to contents of Program Counter if Carry flag is set
	JR	NC,disp	30 (disp-2)	2	7/12							If C = 0, then $[PC] \leftarrow [PC] + 2 + (disp -2)$ Jump relative to contents of Program Counter if Carry flag is reset
	JR	Z,disp	28 (disp-2)	2	7/12							If Z = 1, then $[PC] \leftarrow [PC] + 2 + (disp -2)$ Jump relative to contents of Program Counter if Zero flag is set
	JR	NZ,disp	20 (disp-2)	2	7/12							If Z = 0, then $[PC] \leftarrow [PC] + 2 + (disp -2)$ Jump relative to contents of Program Counter if Zero flag is reset.
	DJNZ	disp	10 (disp-2)	2	8/13							$[B] \leftarrow [B]$ - 1 If $[B] \neq 0$, then $[PC] \leftarrow [PC] + 2 + (disp -2)$ Decrement contents of B and Jump relative to contents of Program Counter if result is not 0.

Table 3-4. A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	C	Z	S	P/O	AC	N	Operation Performed
Register-Register Move	LD	dst,src	01dddsss	1	4							[dst] ← [src] Move contents of source register to destination register. Register designations src and dst may each be A, B, C, D, E, H or L
	LD	A,I	ED 57	2	9		X	X	I	0	0	[A] ← [I] Move contents of Interrupt Vector register to Accumulator.
	LD	A,R	ED 5F	2	9		X	X	I	0	0	[A] ← [R] Move contents of Refresh register to Accumulator
	LD	I,A	ED 47	2	9							[I] ← [A] Load Interrupt Vector register from Accumulator
	LD	R,A	ED 4F	2	9							[R] ← [A] Load Refresh register from Accumulator
	LD	SP,HL	F9	1	6							[SP] ← [HL] Move contents of HL to Stack Pointer
	LD	SP,xy	11x11101 F9	2	10							[SP] ← [xy] Move contents of Index register to Stack Pointer
	EX	DE,HL	EB	1	4							[DE] ⟷ [HL] Exchange contents of DE and HL
	EX	AF,AF'	08	1	4							[AF] ⟷ [AF'] Exchange program status and alternate program status
	EXX		D9	1	4							[BC] ⟷ [BC'], [DE] ⟷ [DE'], [HL] ⟷ [HL'] Exchange register pairs and alternate register pairs

Table 3-4. A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	Status C	Z	S	P/O	A_C	N	Operation Performed
Register-Register Operate	ADD	A,reg	10000rrr	1	4	x	x	x	O	x	0	$[A] \leftarrow [A] + [reg]$ Add contents of register to Accumulator.
	ADC	A,reg	10001rrr	1	4	x	x	x	O	x	0	$[A] \leftarrow [A] + [reg] + C$ Add contents of register and Carry to Accumulator
	SUB	reg	10010rrr	1	4	x	x	x	O	x	1	$[A] \leftarrow [A] - [reg]$ Subtract contents of register from Accumulator
	SBC	A,reg	10011rrr	1	4	x	x	x	O	x	1	$[A] \leftarrow [A] - [reg] - C$ Subtract contents of register and Carry from Accumulator
	AND	reg	10100rrr	1	4	O	x	x	P	1	0	$[A] \leftarrow [A] \wedge [reg]$ AND contents of register with contents of Accumulator
	OR	reg	10110rrr	1	4	O	x	x	P	1	0	$[A] \leftarrow [A] \vee [reg]$ OR contents of register with contents of Accumulator
	XOR	reg	10101rrr	1	4	O	x	x	P	1	0	$[A] \leftarrow [A] \veebar [reg]$ Exclusive-OR contents of register with contents of Accumulator
	CP	reg	10111rrr	1	4	x	x	x	O	x	1	$[A] - [reg]$ Compare contents of register with contents of Accumulator Only the flags are affected
	ADD	HL,rp	00xx1001	1	11	x				?	0	$[HL] \leftarrow [HL] + [rp]$ 16-bit add register pair contents to contents of HL
	ADC	HL,rp	ED 01xx1010	2	15	x	x	x	O	?	0	$[HL] \leftarrow [HL] + [rp] + C$ 16-bit add with Carry register pair contents to contents of HL
	SBC	HL,rp	ED 01xx0010	2	15	x	x	x	O	?	1	$[HL] \leftarrow [HL] - [rp] - C$ 16-bit subtract with Carry register pair contents from contents of HL
	ADD	IX,pp	DD 00xx1001	2	15	x				?	0	$[IX] \leftarrow [IX] + [pp]$ 16-bit add register pair contents to contents of Index register IX (pp = BC, DE, IX, SP)
	ADD	IY,rr	FD 00xx1001	2	15	x				?	0	$[IY] \leftarrow [IY] + [rr]$ 16-bit add register pair contents to contents of Index register IY (rr = BC, DE, IY, SP)

Table 3-4. A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	C	Z	S	P/O	AC	N	Operation Performed
Register Operate	DAA		27	1	4	X	X	X	P	X	1	Decimal adjust Accumulator, assuming that Accumulator contents are the sum or difference of BCD operands
	CPL		2F	1	4					1	1	$[A] \leftarrow [\bar{A}]$ Complement Accumulator (ones complement)
	NEG		ED 44	2	8	X	X	X	O	X	1	$[A] \leftarrow [\bar{A}] + 1$ Negate Accumulator (twos complement)
	INC	reg	00rrr100	1	4		X	X	O	X	0	$[reg] \leftarrow [reg] + 1$ Increment register contents
	INC	rp	00xx0011	1	6							$[rp] \leftarrow [rp] + 1$ or $[xy] \leftarrow [xy] + 1$
		xy	11x11101 23	2	10							Increment contents of register or Index register.
	DEC	reg	00rrr101	1	4		X	X	O	X	1	$[reg] \leftarrow [reg] - 1$ Decrement register contents
	DEC	rp	00xx1011	1	6							$[rp] \leftarrow [rp] - 1$ or $[xy] \leftarrow [xy] - 1$
		xy	11x11101 2B	2	10							Decrement contents of register pair or Index register
Register Shift and Rotate	RLCA		07	1	4	X				0	0	Rotate Accumulator left with branch Carry.
	RLA		17	1	4	X				0	0	Rotate Accumulator left through Carry.
	RRCA		0F	1	4	X				0	0	Rotate Accumulator right with branch Carry.

Table 3-4. A Summary of the Z80 Instruction Set (Continued)

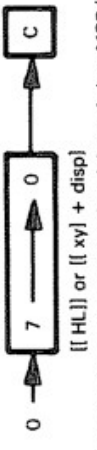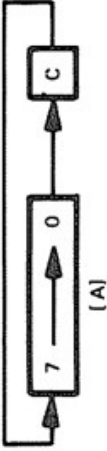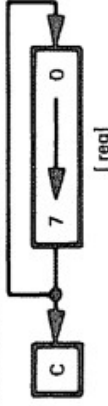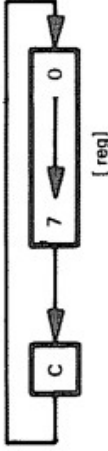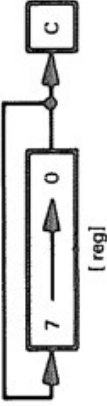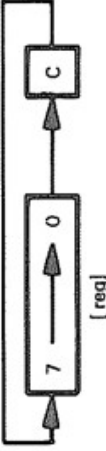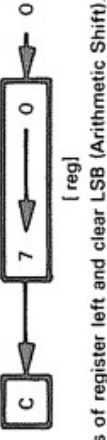| Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	C	Z	S	P/O	Ac	N	Operation Performed
Register Shift and Rotate (Continued)	RRA		1F	1	4	X				0	0	Rotate Accumulator right through Carry. [A]
	RLC	reg	CB 00000rrr	2	8	X	X	X	P	0	0	Rotate contents of register left with branch Carry [reg]
	RL	reg	CB 00010rrr	2	8	X	X	X	P	0	0	Rotate contents of register left through Carry. [reg]
	RRC	reg	CB 00001rrr	2	8	X	X	X	P	0	0	Rotate contents of register right through Carry. [reg]
	RR	reg	CB 00011rrr	2	8	X	X	X	P	0	0	Rotate contents of register right with branch Carry [reg]
	SLA	reg	CB 00100rrr	2	8	X	X	X	P	0	0	Shift contents of register left and clear LSB (Arithmetic Shift). [reg]

Table 3-4. A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	C	Z	S	P/O	Ac	N	Operation Performed
Register Shift and Rotate (Continued)	SRA	reg	CB 00101rrr	2	8	X	X	X	P	0	0	Shift contents of register right and preserve MSB (Arithmetic Shift).
	SRL	reg	CB 00111rrr	2	8	X	X	X	P	0	0	Shift contents of register right and clear MSB (Logical Shift).
	RLD		ED 6F	2	18	X	X	X	P	0	0	Rotate one BCD digit left between the Accumulator and memory location (implied addressing) Contents of the upper half of the Accumulator are not affected.
	RRD		ED 67	2	18	X	X	X	P	0	0	Rotate one BCD digit right between the Accumulator and memory location (implied addressing) Contents of the upper half of the Accumulator are not affected.

Table 3-4. A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	C	Z	S	P/O	Ac	N	Operation Performed
Bit Manipulation	BIT	b,reg	CB 01bbbrrr	2	8		X	?	?	1	0	Z ← $\overline{reg(b)}$; Zero flag contains complement of the selected register bit
	BIT	b,(HL) / b,(xy+displ)	CB 01bbb110 / 11x11101 CB disp 01bbb110	2 / 4	12 / 20		X	?	?	1	0	Z ← $\overline{[[HL]](b)}$ or Z ← $\overline{[[xy]+displ(b)}$; Zero flag contains complement of selected bit of the memory location (implied addressing or base relative addressing)
	SET	b,reg	CB 11bbbrrr	2	8							reg(b) ← 1 ; Set indicated register bit
	SET	b,(HL) / b,(xy+displ)	CB 11bbb110 / 11x11101 CB disp 11bbb110	2 / 4	15 / 23							[[HL]](b) ← 1 or [[xy]+displ(b) ← 1 ; Set indicated bit of memory location (implied addressing or base relative addressing)
	RES	b,reg	CB 10bbbrrr	2	8							reg(b) ← 0 ; Reset indicated register bit
	RES	b,(HL) / b,(xy+displ)	CB 10bbb110 / 11x11101 CB disp 10bbb110	2 / 4	15 / 23							[[HL]](b) ← 0 or [[xy]+displ(b) ← 0 ; Reset indicated bit in memory location (implied addressing or base relative addressing)
Stack	PUSH	pr / xy	11xx0101 / 11x11101 E5	1 / 2	11 / 15							[[SP]-1] ← [pr(HI)] ; [[SP]-2] ← [pr(LO)] ; [SP] ← [SP]-2 ; Put contents of register pair or Index register on top of Stack and decrement Stack Pointer
	POP	pr / xy	11xx0001 / 11x11101 E1	1 / 2	10 / 14							[pr(LO)] ← [[SP]] ; [pr(HI)] ← [[SP]+1] ; [SP] ← [SP]+2 ; Put contents of top of Stack in register pair or Index register and increment Stack Pointer.
	EX	(SP),HL / (SP),xy	E3 / 11x11101 E3	1 / 2	19 / 23							[H] ←→ [[SP]+1] ; [L] ←→ [[SP]] ; Exchange contents of HL or Index register and top of Stack

Table 3-4. A Summary of the Z80 Instruction Set (Continued)

Type	Mnemonic	Operand	Object Code	Bytes	Clock Cycles	C	Z	S	P/O	A_C	N	Operation Performed
Interrupt	DI		F3	1	4							Disable interrupts
	EI		FB	1	4							Enable interrupts
	RST	n	11xxx111	1	11							[[SP]-1] — [PC(HI)] [[SP]-2] — [PC(LO)] [SP] — [SP]-2 [PC] — $(8 \cdot n)_{16}$ Restart at designated location.
	RETI		ED 4D	2	14							Return from interrupt
	RETN		ED 45	2	14							Return from nonmaskable interrupt.
	IM	0	ED 46	2	8							Set interrupt mode 0, 1, or 2
		1	ED 56	2	8							
		2	ED 5E	2	8							
Status	SCF		37	1	4	1				0	0	C — 1 Set Carry flag
	CCF		3F	1	4	X				?	0	C — $\overline{C}$ Complement Carry flag
	NOP		00	1	4							No operation — volatile memories are refreshed
	HALT		76	1	4							CPU halts, executes NOPs to refresh volatile memories

**Execution time shown is for one iteration.

3-39

Table 3-5. Instruction Object Codes in Numerical Order

OBJECT CODE	INSTRUCTION		OBJECT CODE	INSTRUCTION	
00	NOP		39	ADD	HL,SP
01 yyyy	LD	BC,data16	3A ppqq	LD	A,(addr)
02	LD	(BC),A	3B	DEC	SP
03	INC	BC	3C	INC	A
04	INC	B	3D	DEC	A
05	DEC	B	3E yy	LD	A,data
06 yy	LD	B,data	3F	CCF	
07	RLCA		4 0sss	LD	B,reg
08	EX	AF,AF'	46	LD	B,(HL)
09	ADD	HL,BC	4 1sss	LD	C,reg
0A	LD	A,(BC)	4E	LD	C,(HL)
0B	DEC	BC	5 0sss	LD	D,reg
0C	INC	C	56	LD	D,(HL)
0D	DEC	C	5 1sss	LD	E,reg
0E yy	LD	C,data	5E	LD	E,(HL)
0F	RRCA		6 0sss	LD	H,reg
10 disp-2	DJNZ	disp	66	LD	H,(HL)
11 yyyy	LD	DE,data16	6 1sss	LD	L,reg
12	LD	(DE),A	6E	LD	L,(HL)
13	INC	DE	7 0sss	LD	(HL),reg
14	INC	D	76	HALT	
15	DEC	D	7 1sss	LD	A,reg
16 yy	LD	D,data	7E	LD	A,(HL)
17	RLA		8 0rrr	ADD	A,reg
18 disp-2	JR	disp	86	ADD	A,(HL)
19	ADD	HL,DE	8 1rrr	ADC	A,reg
1A	LD	A,(DE)	8E	ADC	A,(HL)
1B	DEC	DE	9 0rrr	SUB	reg
1C	INC	E	96	SUB	(HL)
1D	DEC	E	9 1rrr	SBC	A,reg
1E yy	LD	E,data	9E	SBC	A,(HL)
1F	RRA		A 0rrr	AND	reg
20 disp-2	JR	NZ,disp	A6	AND	(HL)
21 yyyy	LD	HL,data16	A 1rrr	XOR	reg
22 ppqq	LD	(addr),HL	AE	XOR	(HL)
23	INC	HL	B 0rrr	OR	reg
24	INC	H	B6	OR	(HL)
25	DEC	H	B 1rrr	CP	reg
26 yy	LD	H,data	BE	CP	(HL)
27	DAA		C0	RET	NZ
28 disp-2	JR	Z,disp	C1	POP	BC
29	ADD	HL,HL	C2 ppqq	JP	NZ,addr
2A ppqq	LD	HL,(addr)	C3 ppqq	JP	addr
2B	DEC	HL	C4 ppqq	CALL	NZ,addr
2C	INC	L	C5	PUSH	BC
2D	DEC	L	C6 yy	ADD	A,data
2E	LD	L,data	C7	RST	00H
2F	CPL		C8	RET	Z
30 disp-2	JR	NC,disp	C9	RET	
31 yyyy	LD	SP,data16	CA ppqq	JP	Z,addr
32 ppqq	LD	(addr),A	CB 0 0rrr	RLC	reg
33	INC	SP	CB 06	RLC	(HL)
34	INC	(HL)	CB 0 1rrr	RRC	reg
35	DEC	(HL)	CB 0E	RRC	(HL)
36 yy	LD	(HL),data	CB 1 0rrr	RL	reg
37	SCF		CB 16	RL	(HL)
38	JR	C,disp	CB 1 1rrr	RR	reg

Table 3-5. Instruction Object Codes in Numerical Order (Continued)

OBJECT CODE	INSTRUCTION		OBJECT CODE	INSTRUCTION	
CB 1E	RR	(HL)	DD CB disp 10bbb110	RES	b,(IX + disp)
CB 2 0rrr	SLA	reg	DD CB disp 11bbb110	SET	b,(IX + disp)
CB 26	SLA	(HL)	DD E1	POP	IX
CB 2 1rrr	SRA	reg	DD E3	EX	(SP),IX
CB 2E	SRA	(HL)	DD E5	PUSH	IX
CB 3 1rrr	SRL	reg	DD E9	JP	(IX)
CB 3E	SRL	(HL)	DD F9	LD	SP,IX
CB 01bbbrrr	BIT	b,reg	DE yy	SBC	A,data
CB 01bbb110	BIT	b,(HL)	DF	RST	18H
CB 10bbbrrr	RES	b,reg	E0	RET	PO
CB 10bbb110	RES	b,(HL)	E1	POP	HL
CB 11bbbrrr	SET	b,reg	E2 ppqq	JP	PO,addr
CB 11bbb110	SET	b,(HL)	E3	EX	(SP),HL
CC ppqq	CALL	Z,addr	E4 ppqq	CALL	PO,addr
CD ppqq	CALL	addr	E5	PUSH	HL
CE yy	ADC	A,data	E6 yy	AND	data
CF	RST	08H	E7	RST	20H
D0	RET	NC	E8	RET	PE
D1	POP	DE	E9	JP	(HL)
D2 ppqq	JP	NC,addr	EA ppqq	JP	PE,addr
D3 yy	OUT	(port),A	EB	EX	DE,HL
D4 ppqq	CALL	NC,addr	EC ppqq	CALL	PE,addr
D5	PUSH	DE	ED 01ddd000	IN	reg,(C)
D6 yy	SUB	data	ED 01sss001	OUT	(C),reg
D7	RST	10H	ED 01xx 2	SBC	HL,rp
D8	RET	C	ED 01xx 3 ppqq	LD	(addr),rp
D9	EXX		ED 44	NEG	
DA ppqq	JP	C,addr	ED 45	RETN	
DB yy	IN	A,(port)	ED 010nn110	IM	m
DC ppqq	CALL	C,addr	ED 47	LD	I,A
DD 00xx 9	ADD	IX,pp	ED 01xx A	ADC	HL,rp
DD 21 yyyy	LD	IX,data16	ED 01xx B ppqq	LD	rp,(addr)
DD 22 ppqq	LD	(addr),IX	ED 4D	RETI	
DD 23	INC	IX	ED 4F	LD	R,A
DD 2A ppqq	LD	IX,(addr)	ED 57	LD	A,I
DD 2B	DEC	IX	ED 5F	LD	A,R
DD 34 disp	INC	(IX + disp)	ED 67	RRD	
DD 35 disp	DEC	(IX + disp)	ED 6F	RLD	
DD 36 disp yy	LD	(IX + disp),data	ED A0	LDI	
DD 01ddd110 disp	LD	reg,(IX + disp)	ED A1	CPI	
DD 1 0sss disp	LD	(IX + disp),reg	ED A2	INI	
DD 86 disp	ADD	A,(IX + disp)	ED A3	OUTI	
DD 8E disp	ADC	A,(IX + disp)	ED A8	LDD	
DD 96 disp	SUB	(IX + disp)	ED A9	CPD	
DD 9E disp	SBC	A,(IX + disp)	ED AA	IND	
DD A6 disp	AND	(IX + disp)	ED AB	OUTD	
DD AE disp	XOR	(IX + disp)	ED B0	LDIR	
DD B6 disp	OR	(IX + disp)	ED B1	CPIR	
DD BE disp	CP	(IX + disp)	ED B2	INIR	
DD CB disp 06	RLC	(IX + disp)	ED B3	OTIR	
DD CB disp 0E	RRC	(IX + disp)	ED B8	LDDR	
DD CB disp 16	RL	(IX + disp)	ED B9	CPDR	
DD CB disp 1E	RR	(IX + disp)	ED BA	INDR	
DD CB disp 26	SLA	(IX + disp)	ED BB	OTDR	
DD CB disp 2E	SRA	(IX + disp)	EE yy	XOR	data
DD CB disp 3E	SRL	(IX + disp)	EF	RST	28H
DD CB disp 01bbb110	BIT	b,(IX + disp)			

Table 3-5. Instruction Object Codes in Numerical Order (Continued)

OBJECT CODE	INSTRUCTION	
F0	RET	P
F1	POP	AF
F2 ppqq	JP	P.addr
F3	DI	
F4 ppqq	CALL	P.addr
F5	PUSH	AF
F6 yy	OR	data
F7	RST	30H
F8	RET	M
F9	LD	SP,HL
FA ppqq	JP	M,addr
FB	EI	
FC ppqq	CALL	M,addr
FD 00xx 9	ADD	IY,rr
FD 21 yyyy	LD	IY,data16
FD 22 ppqq	LD	(addr),IY
FD 23	INC	IY
FD 2A ppqq	LD	IY,(addr)
FD 2B	DEC	IY
FD 34 disp	INC	(IY + disp)
FD 35 disp	DEC	(IY + disp)
FD 36 disp yy	LD	(IY + disp),data
FD 01ddd110 disp	LD	reg,(IY + disp)
FD 7 0sss disp	LD	(IY + disp),reg
FD 86 disp	ADD	A,(IY + disp)

OBJECT CODE	INSTRUCTION	
FD 8E disp	ADC	A,(IY + disp)
FD 96 disp	SUB	(IY + disp)
FD 9E disp	SBC	A,(IY + disp)
FD A6 disp	AND	(IY + disp)
FD AE disp	XOR	(IY + disp)
FD B6 disp	OR	(IY + disp)
FD BE disp	CP	(IY + disp)
FD CB disp 06	RLC	(IY + disp)
FD CB disp 0E	RRC	(IY + disp)
FD CB disp 16	RL	(IY + disp)
FD CB disp 1E	RR	(IY + disp)
FD CB disp 26	SLA	(IY + disp)
FD CB disp 2E	SRA	(IY + disp)
FD CB disp 3E	SRL	(IY + disp)
FD CB disp 01bbb110	BIT	b,(IY + disp)
FD CB disp 10bbb110	RES	b,(IY + disp)
FD CB disp 11bbb110	SET	b,(IY + disp)
FD E1	POP	IY
FD E3	EX	(SP),IY
FD E5	PUSH	IY
FD E9	JP	(IY)
FD F9	LD	SP,IY
FE yy	CP	data
FF	RST	38H

ADC A,data — ADD IMMEDIATE WITH CARRY TO ACCUMULATOR

$$\underbrace{\text{ADC A,}}_{\text{CE}} \quad \underbrace{\text{data}}_{\text{yy}}$$
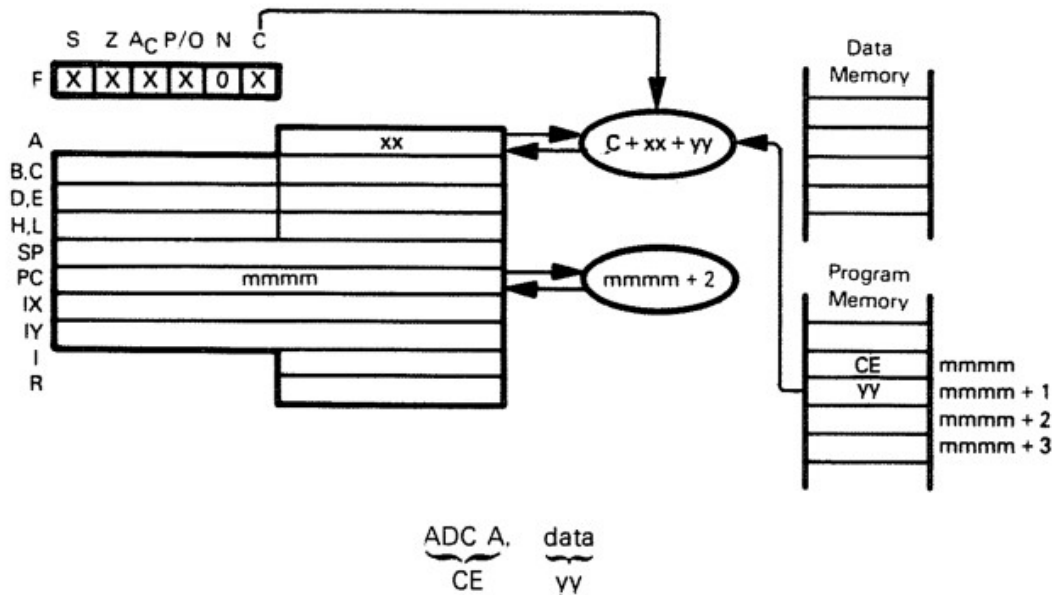
Add the contents of the next program memory byte and the Carry status to the Accumulator.

Suppose $xx=3A_{16}$, $yy=7C_{16}$, and Carry=0. After the instruction

$$\text{ADC} \quad \text{A,7CH}$$

has executed, the Accumulator will contain $B6_{16}$:
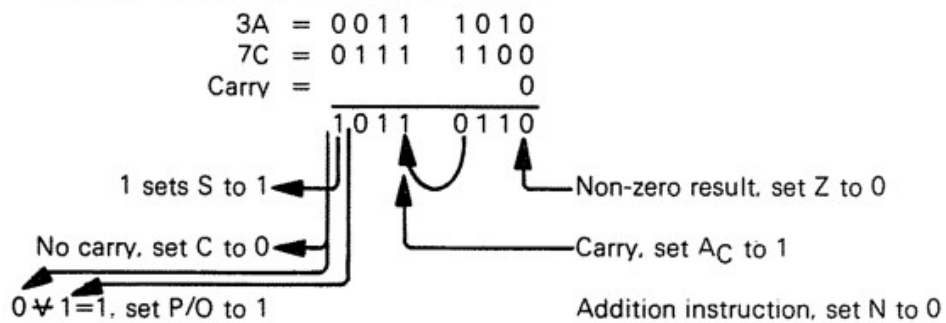
The ADC instruction is frequently used in multibyte addition for the second and subsequent bytes.

ADC A,reg — ADD REGISTER WITH CARRY TO ACCUMULATOR
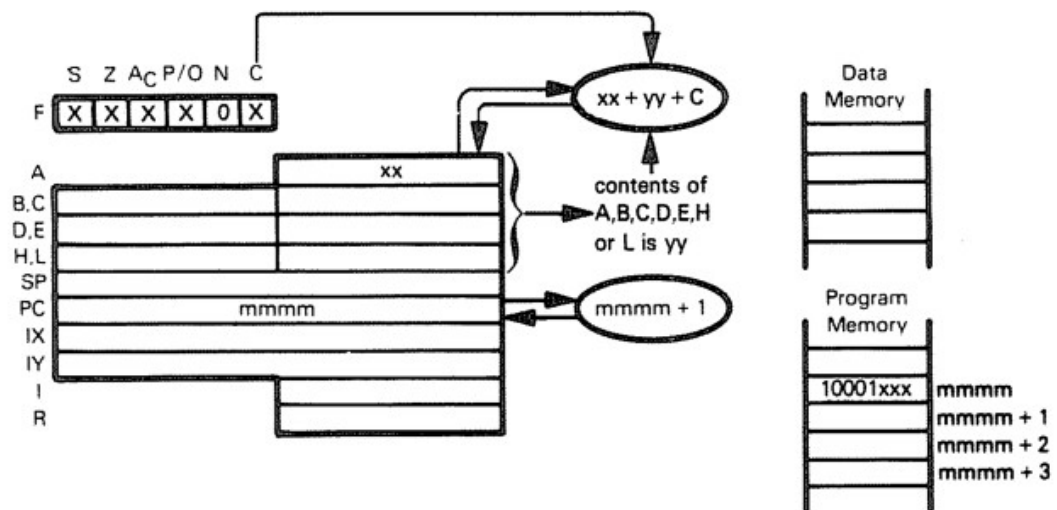

```
ADC A,    reg
⌣⌣⌣⌣⌣    ⌣⌣⌣
10001     xxx
          000  for reg=B
          001  for reg=C
          010  for reg=D
          011  for reg=E
          100  for reg=H
          101  for reg=L
          111  for reg=A
```

Add the contents of Register A, B, C, D, E, H or L and the Carry status to the Accumulator.

Suppose $xx=E3_{16}$, Register E contains $A0_{16}$, and Carry=1. After the instruction

ADC   A,E

has executed, the Accumulator will contain 84_{16}:
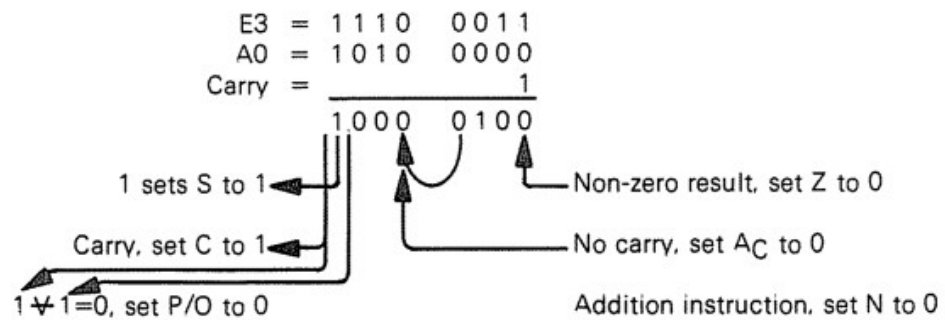
```
   E3  =  1 1 1 0   0 0 1 1
   A0  =  1 0 1 0   0 0 0 0
Carry  =                  1
          1 0 0 0   0 1 0 0
```

1 sets S to 1 ◄——
Carry, set C to 1 ◄——
$1 ⊻ 1=0$, set P/O to 0

Non-zero result, set Z to 0
No carry, set A_C to 0
Addition instruction, set N to 0

The ADC instruction is most frequently used in multibyte addition for the second and subsequent bytes.

ADC A,(HL) — ADD MEMORY AND CARRY TO
ADC A,(IX+disp)   ACCUMULATOR
ADC A,(IY+disp)
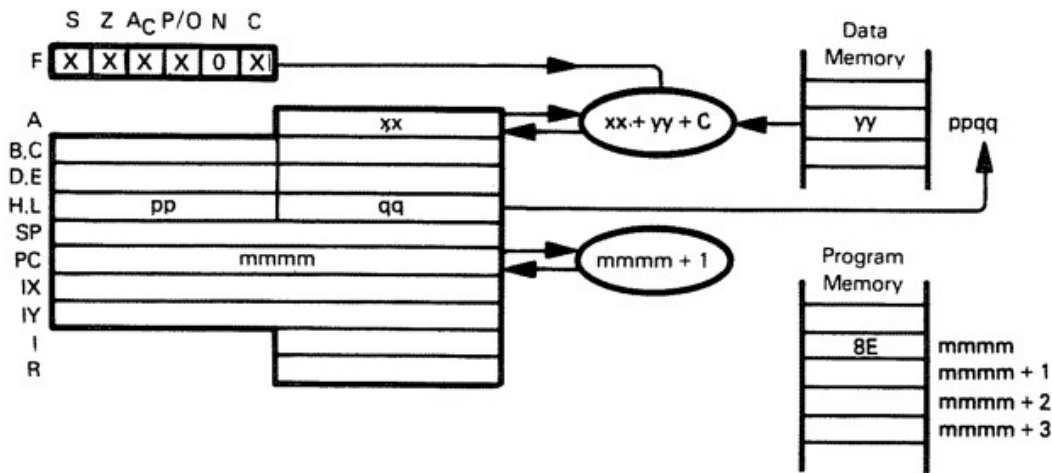
The illustration shows execution of ADC A,(HL):

$$\underbrace{\text{ADC A,(HL)}}_{\text{8E}}$$

Add the contents of memory location (specified by the contents of the HL register pair) and the Carry status to the Accumulator.

Suppose $xx=E3_{16}$, $yy=A0_{16}$, and Carry=1. After the instruction

ADC   A,(HL)

has executed, the Accumulator will contain 84_{16}:
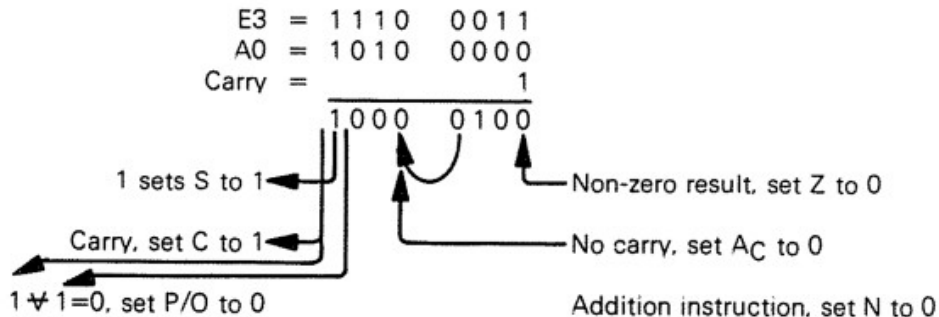
$$\underbrace{\text{ADC A,(IX+disp)}}$$

DD 8E   d

Add the contents of memory location (specified by the sum of the contents of the IX register and the displacement digit d) and the Carry to the Accumulator.

$$\underbrace{\text{ADC A,(IY+disp)}}$$

FD 8E   d

This instruction is identical to ADC A,(IX+disp), except that it uses the IY register instead of the IX register.

The ADC instruction is most frequently used in multibyte addition for the second and subsequent bytes.

ADC HL,rp — ADD REGISTER PAIR WITH CARRY TO H AND L
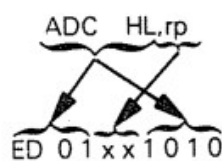
S   Z   A$_C$ P/O N   C

F | X | X | X | X | 0 | X |

A
B,C
D,E
H,L   xx   xx
SP
PC   mmmm
IX
IY
I
R

BC, DE, HL or SP
contain yyyy

xxxx + yyyy
+ C

mmmm + 2

Data Memory

Program Memory

ED	mmmm
01xx1010	mmmm + 1
	mmmm + 2
	mmmm + 3

ADC   HL,rp

ED 0 1 x x 1 0 1 0

00 for rp is register pair BC
01 for rp is register pair DE
10 for rp is register pair HL
11 for rp is Stack Pointer

Add the 16-bit value from either the BC, DE, HL register pair or the Stack Pointer, and the Carry status, to the HL register pair.

Suppose HL contains A536$_{16}$, BC contains 1044$_{16}$, and Carry=1. After execution of

ADC   HL,BC

the HL register pair will contain:

A536 = 1010 0101 0011 0110
1044 = 0001 0000 0100 0100
Carry =   1
         1011 0101 0111 1011
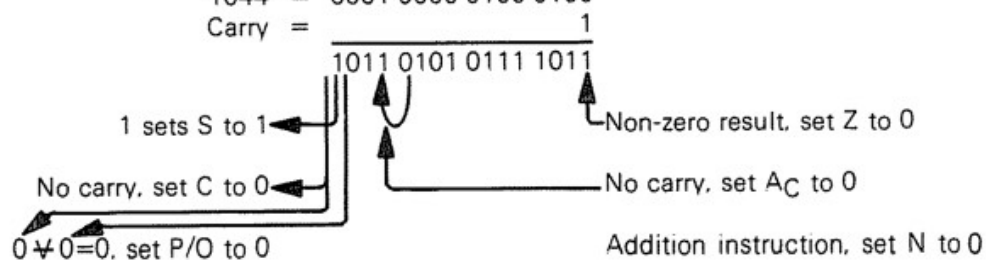
1 sets S to 1

No carry, set C to 0

0 ⩛ 0=0, set P/O to 0

Non-zero result, set Z to 0

No carry, set A$_C$ to 0

Addition instruction, set N to 0

The ADC instruction is most frequently used in multibyte addition for the second and subsequent bytes.

ADD A,data — ADD IMMEDIATE TO ACCUMULATOR
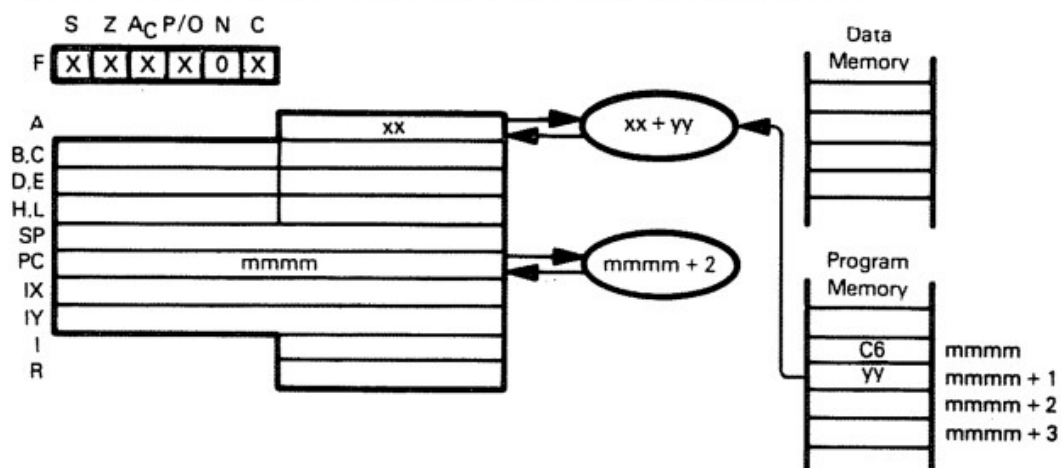
S Z A$_C$P/O N C

F | X | X | X | X | 0 | X |

A   xx   →← xx + yy ←

B,C
D,E
H,L
SP
PC   mmmm   →← mmmm + 2
IX
IY
I
R

Data Memory

Program Memory

C6	mmmm
yy	mmmm + 1
	mmmm + 2
	mmmm + 3

$$\underbrace{\text{ADD A,}}_{\text{C6}}\ \underbrace{\text{data}}_{\text{yy}}$$

Add the contents of the next program memory byte to the Accumulator.

Suppose xx=3A$_{16}$, yy=7C$_{16}$, and Carry=0. After the instruction

        ADD   A,7CH

has executed, the Accumulator will contain B6$_{16}$:

```
3A  =  0 0 1 1   1 0 1 0
7C  =  0 1 1 1   1 1 0 0
       1,0 1 1   0 1 1 0
```

1 sets S to 1 ◄─────────────────────────────── Non-zero result, set Z to 0

No carry, set C to 0 ◄────────────────── Carry, set A$_C$ to 1

0 ⩒ 1=1: set P/O to 1   Addition instruction, set N to 0
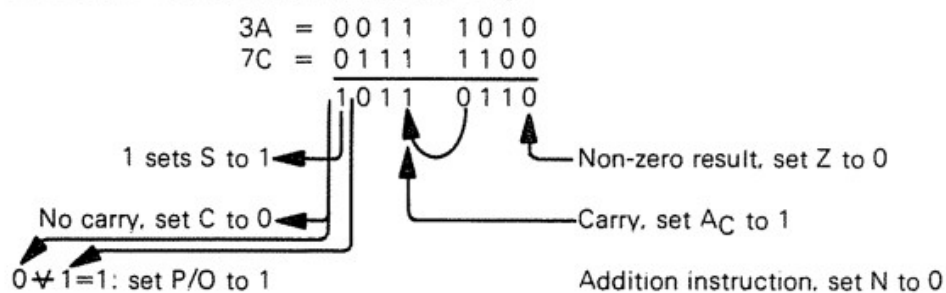
This is a routine data manipulation instruction.

ADD A,reg — ADD CONTENTS OF REGISTER TO ACCUMULATOR

```
     S  Z AC P/O N  C
  F [X][X][X][X][0][X]                              xx + yy

  A                          |        xx       |      contents of
  B,C                        |                 |      A,B,C,D,E,
  D,E                        |                 |      H or L is yy
  H,L
  SP
  PC                         |      mmmm       |     mmmm + 1
  IX
  IY
  I                                                  10000xxx  mmmm
  R                                                            mmmm + 1
                                                               mmmm + 2
                                                               mmmm + 3
```

Data Memory

Program Memory
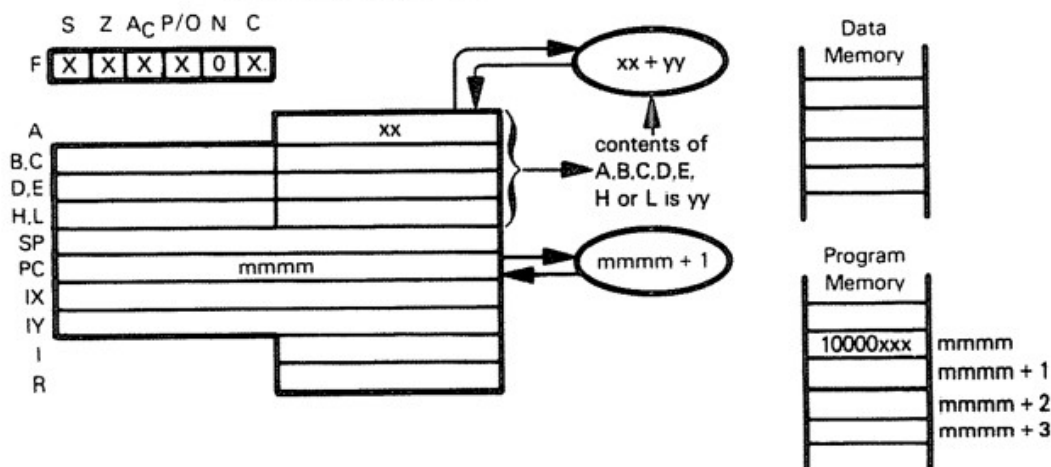
```
        ADD    reg
       10000   xxx
               000   for reg=B
               001   for reg=C
               010   for reg=D
               011   for reg=E
               100   for reg=H
               101   for reg=L
               111   for reg=A
```

Add the contents of Register A, B, C, D, E, H or L to the Accumulator.

Suppose $xx=E3_{16}$, Register E contains $A0_{16}$. After execution of

$$ADD \ A,E$$

the Accumulator will contain 83_{16}:

```
        E3 = 1110  0011
        A0 = 1010  0000
             1,000  0011
```

1 sets S to 1 ◄

Carry, set C to 1 ◄

1 ⊻ 1=0, set P/O to 0

Non-zero result, set Z to 0

No carry, set AC to 0

Addition instruction, set N to 0
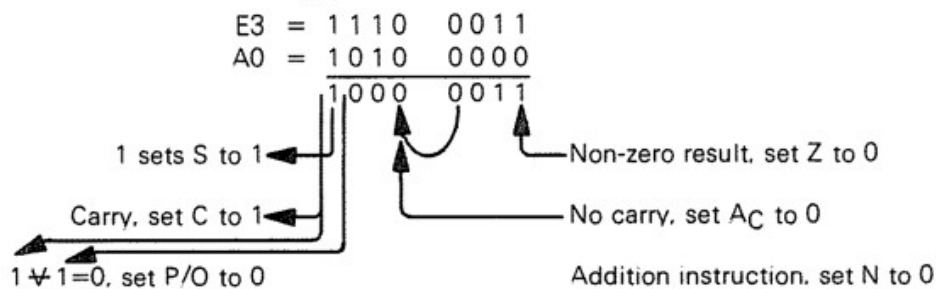
This is a routine data manipulation instruction

ADD A,(HL) — ADD MEMORY TO ACCUMULATOR
ADD A,(IX+disp)
ADD A,(IY+disp)
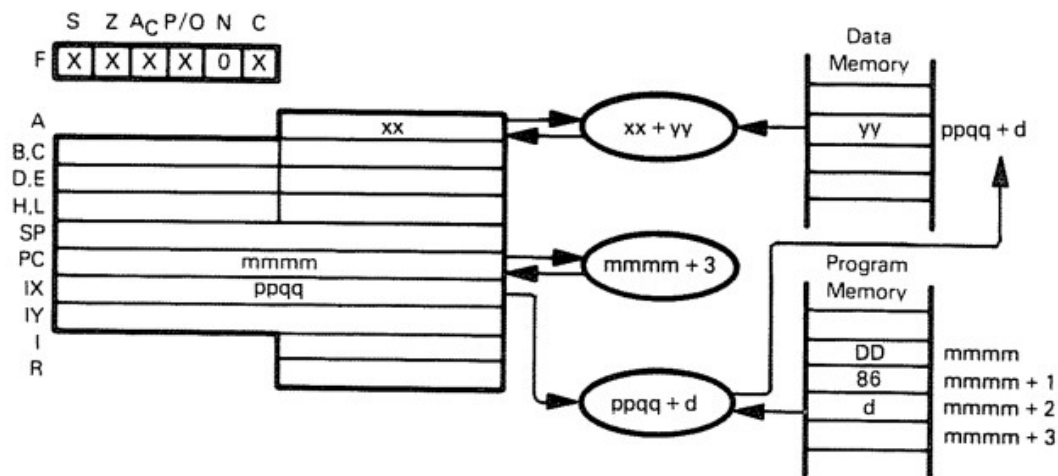
The illustration shows execution of ADD A,(IX+disp).

$$\underbrace{\text{ADD A,(IX+disp)}}$$

$$\text{DD} \quad 86 \quad \text{d}$$

Add the contents of memory location (specified by the sum of the contents of the IX register and the displacement digit d) to the contents of the Accumulator.

Suppose $ppqq=4000_{16}$, $xx=1A_{16}$, and memory location $400F_{16}$ contains 50_{16}. After the instruction

$$\text{ADD A,(IX+0FH)}$$

has executed, the Accumulator will contain $6A_{16}$.

```
1A  =  0001   1010
50  =  0101   0000
       0 110   1010
```

0 sets S to 0 ◄————
                                        ————— Non-zero result, set Z to 0
No carry, set C to 0 ◄————
                                        ————— No carry, set A_C to 0
$0 \not\forall 0=0$; set P/O to 0
                            Addition instruction, set N to 0

$$\underbrace{\text{ADD A,(IY+disp)}}$$

$$\text{FD} \quad 86 \quad \text{d}$$
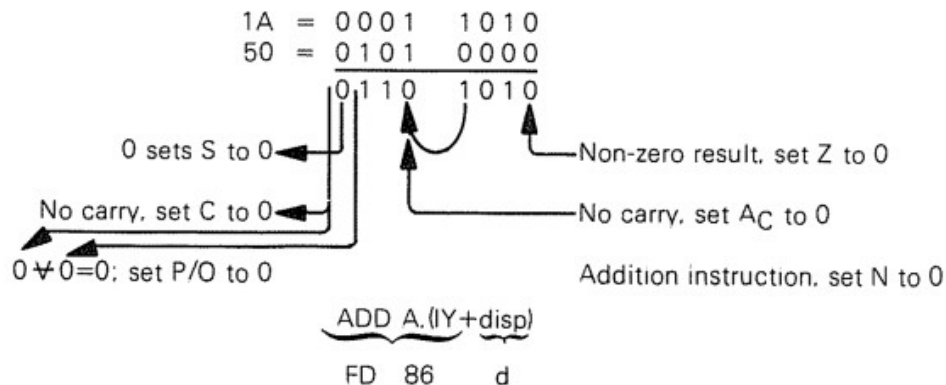
This instruction is identical to ADD A,(IX+disp), except that it uses the IY register instead of the IX register.

$$\underbrace{\text{ADD A,(HL)}}$$

$$86$$

This version of the instruction adds the contents of memory location, specified by the contents of the HL register pair, to the Accumulator.

The ADD instruction is a routine data manipulation instruction.

ADD HL,rp — ADD REGISTER PAIR TO H AND L
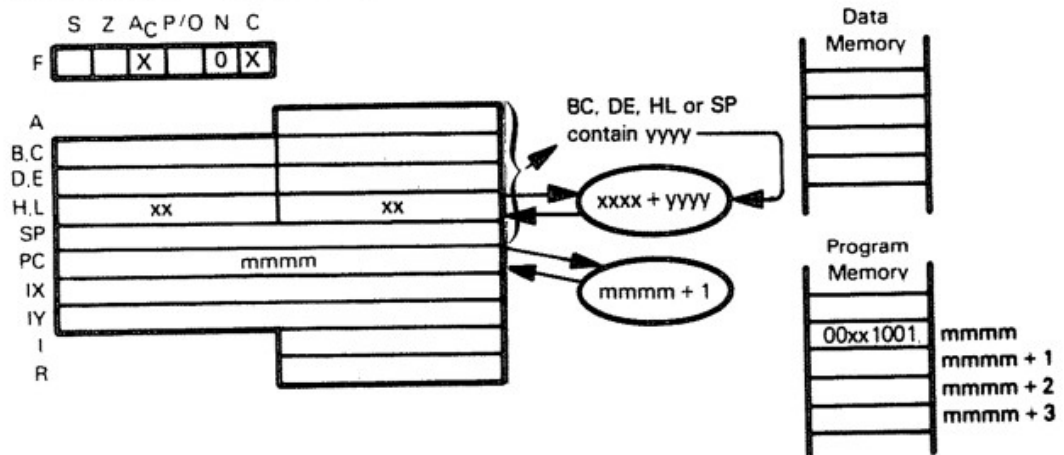

```
ADD  HL,rp
```



```
00  xx  1001
```
00 for rp is register pair BC
01 for rp is register pair DE
10 for rp is register pair HL
11 for rp is Stack Pointer

Add the 16-bit value from either the BC, DE, HL register pair or the Stack Pointer to the HL register pair.

Suppose HL contains $034A_{16}$ and BC contains $214C_{16}$. After the instruction

ADD HL,BC

has executed, the HL register pair will contain 2496_{16}.

```
034A  =  0000 0011 0100 1010
214C  =  0010 0001 0100 1100
         ─────────────────────
         0010 0100 1001 0110
```

No carry, set C to 0 ◄─────────┘ └───────────── No carry, set A_C to 0

Addition instruction, set N to 0

The ADD HL,HL instruction is equivalent to a 16-bit left shift.

ADD xy,rp — ADD REGISTER PAIR TO INDEX REGISTER
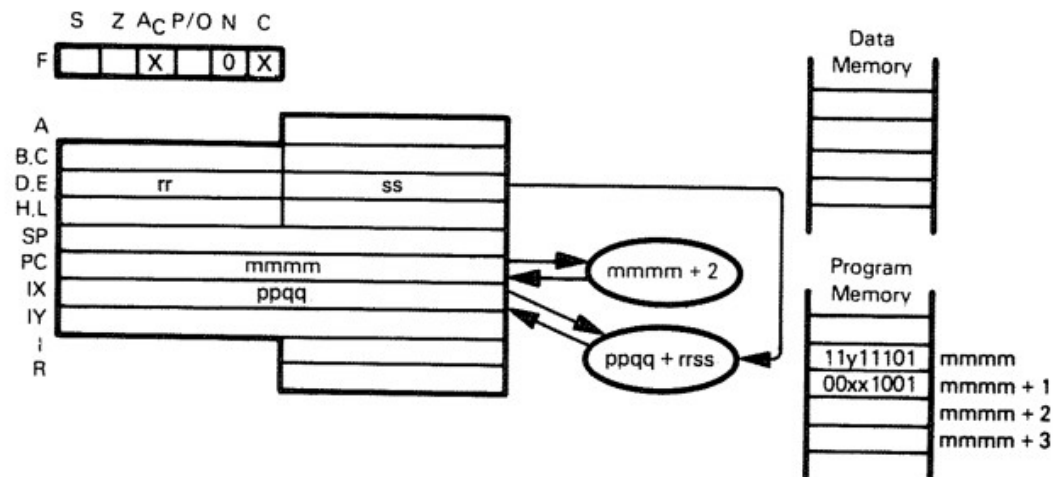
The illustration shows execution of ADD IX,DE.

0 for Index register=IX   00 for rp is register pair BC
1 for Index register=IY   01 for rp is register pair DE
                                       10 for rp is specified Index register
                                       11 for rp is Stack Pointer
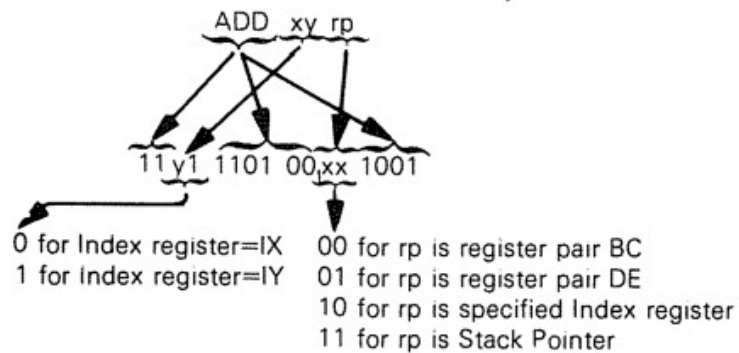
Add the contents of the specified register pair to the contents of the specified Index register.

Suppose IY contains $4FF0_{16}$ and BC contains $000F_{16}$. After the instruction

ADD IY,BC

has executed, Index Register IY will contain $4FFF_{16}$.

AND data — AND IMMEDIATE WITH ACCUMULATOR
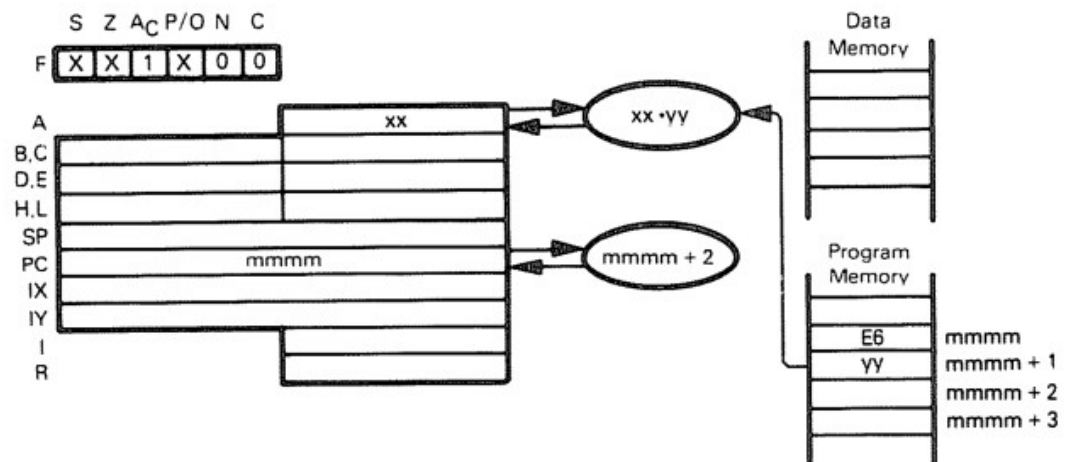
$$\underbrace{AND}_{E6} \quad \underbrace{data}_{yy}$$

AND the contents of the next program memory byte to the Accumulator.

Suppose xx=3A$_{16}$. After the instruction

$$AND \quad 7CH$$

has executed, the Accumulator will contain 38$_{16}$.

```
3A  =  0011   1010
7C  =  0111   1100
       0011   1000
```

0 sets S to 0 ←——┘

——Three 1 bits, set P/O to 0

——Non-zero result, set Z to 0

This is a routine logical instruction; it is often used to turn bits "off". For example, the instruction

$$AND \quad 7FH$$

will unconditionally set the high order Accumulator bit to 0.

AND reg — AND REGISTER WITH ACCUMULATOR
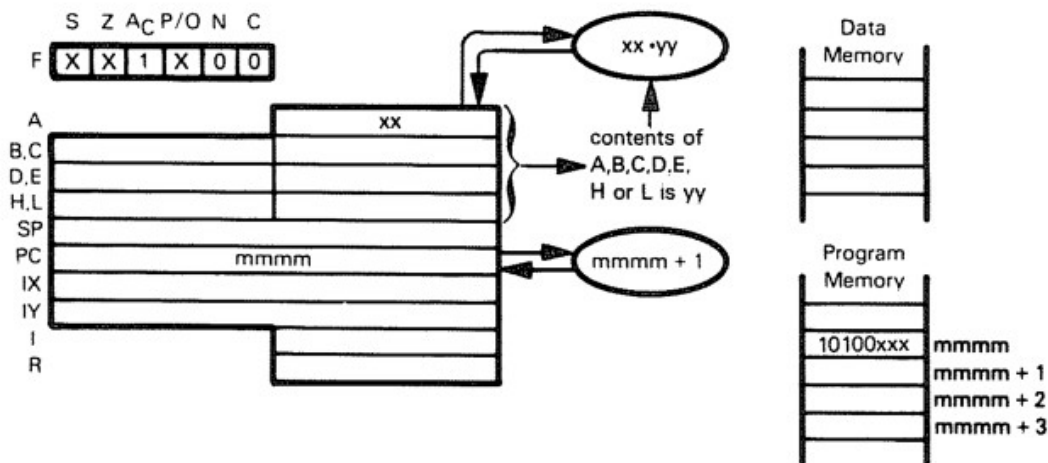

```
         AND    reg
        10100   xxx
                000   for reg=B
                001   for reg=C
                010   for reg=D
                011   for reg=E
                100   for reg=H
                101   for reg=L
                111   for reg=A
```

AND the Accumulator with the contents of Register A. B, C, D, E, H or L. Save the result in the Accumulator.

Suppose $xx=E3_{16}$, and Register E contains $A0_{16}$. After the instruction

<p style="text-align:center">AND E</p>

has executed, the Accumulator will contain $A0_{16}$.

```
E3  =  1 1 1 0   0 0 1 1
A0  =  1 0 1 0   0 0 0 0
       1 0 1 0   0 0 0 0
```

1 sets S to 1 ◄─────┘   └── Two 1 bits, set P/O to 1

└── Non-zero result, set Z to 0

AND is a frequently used logical instruction.

AND (HL) — AND MEMORY WITH ACCUMULATOR
AND (IX+disp)
AND (IY+disp)
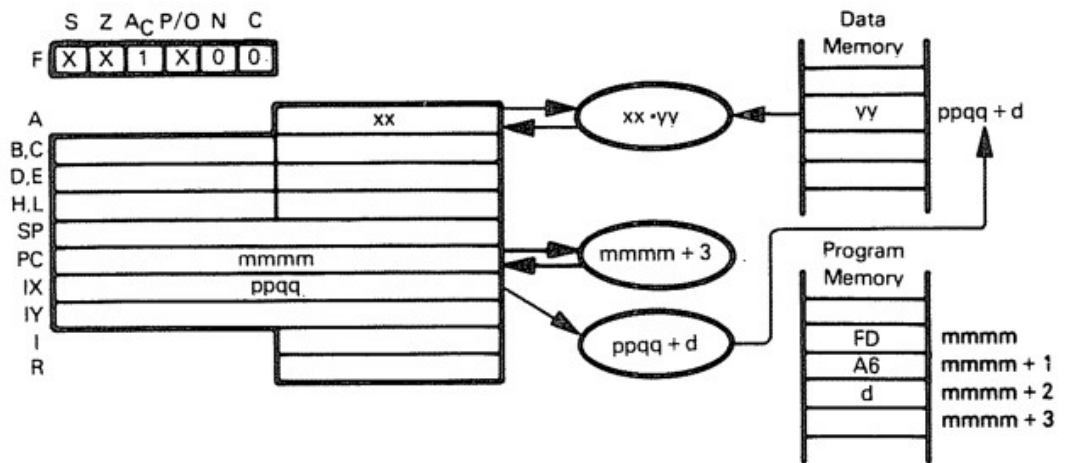
The illustration shows execution of AND (IY+disp).

$$\underbrace{\text{AND (IY+disp)}}\ \underbrace{}$$
$$\text{FD}\quad\text{A6}\qquad\text{d}$$
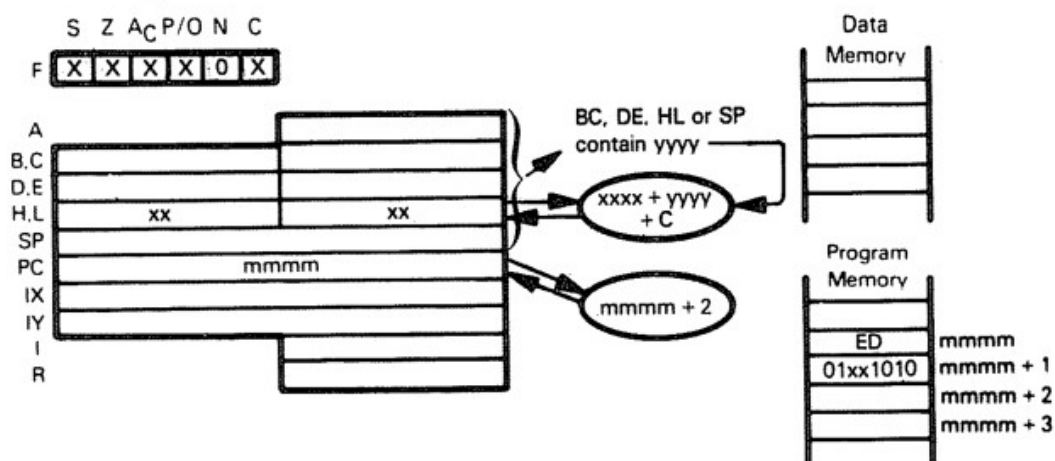
AND the contents of memory location (specified by the sum of the contents of the IY register and the displacement digit d) with the Accumulator.

Suppose $xx=E3_{16}$, $ppqq=4000_{16}$, and memory location $400F_{16}$ contains $A0_{16}$. After the instruction

$$\text{AND (IY+0FH)}$$

has executed, the Accumulator will contain $A0_{16}$.

$$
\begin{array}{rcll}
E3 & = & 1110 & 0111 \\
A0 & = & 1010 & 0000 \\
\hline
   &   & 1010 & 0000 \\
\end{array}
$$

1 sets S to 1 ◀——

——Two 1 bits, set P/O to 1

——Non-zero result, set Z to 0

$$\underbrace{\text{AND (IX+disp)}}\ \underbrace{}$$
$$\text{DD}\quad\text{A6}\qquad\text{d}$$

This instruction is identical to AND (IY+disp), except that it uses the IX register instead of the IY register.

$$\underbrace{\text{AND (HL)}}$$
$$\text{A6}$$

AND the contents of the memory location (specified by the contents of the HL register pair) with the Accumulator.

AND is a frequently used logical instruction.

BIT b,reg — TEST BIT b IN REGISTER reg
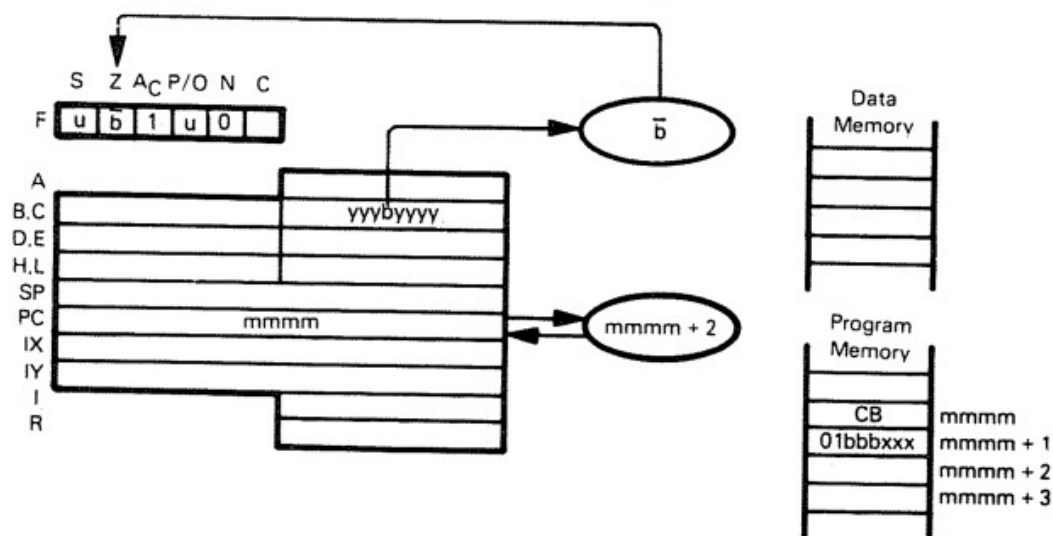
	BIT	b,	reg	
	CB 01	bbb	xxx	

Bit Tested			Register
0	000	000	B
1	001	001	C
2	010	010	D
3	011	011	E
4	100	100	H
5	101	101	L
6	110	111	A
7	111		

Place complement of indicated register's specified bit in Z flag of F register.

Suppose Register C contains 1110 1111. The instruction BIT 4.C will then set the Z flag to 1. while bit 4 in Register C remains 0. Bit 0 is the least significant bit.

3-55

BIT b,(HL) — TEST BIT b OF INDICATED MEMORY POSITION
BIT b,(IX+disp)
BIT b,(IY+disp)
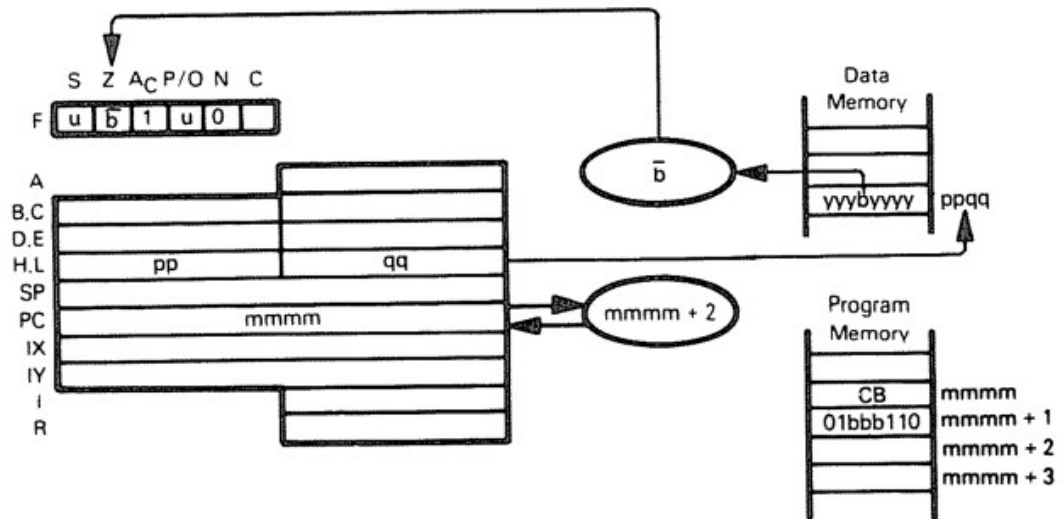
The illustration shows execution of BIT 4,(HL). Bit 0 is the least significant bit.

BIT	b,	(HL)
CB 01	bbb	110

Bit Tested	bbb
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Test indicated bit within memory position specified by the contents of Register HL, and place bit's complement in Z flag of the F register.

Suppose HL contains 4000H and bit 3 in memory location 4000H contains 1. The instruction

BIT 3,(HL)

will then set the Z flag to 0, while bit 3 in memory location 4000H remains 1.

BIT b,(IX+disp)

DD CB d 01 bbb 110
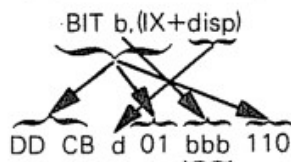
bbb is the same as in BIT b,(HL)

Examine specified bit within memory location indicated by the sum of Index Register IX and disp. Place the complement in the Z flag of the F register.

Suppose Index Register IX contains 4000H and bit 4 of memory location 4004H is 0. The instruction

BIT 4,(IX+4H)

will then set the Z flag to 1. while bit 4 of memory location 4004H remains 0.

BIT b,(IY+disp)

FD CB d 01 bbb 110
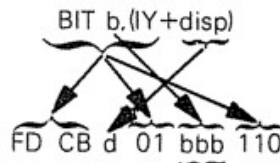
bbb is the same as in BIT b,(HL)

This instruction is identical to BIT b,(IX+disp), except that it uses the IY register instead of the IX register.

CALL label — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND

CALL   label
 CD   ppqq
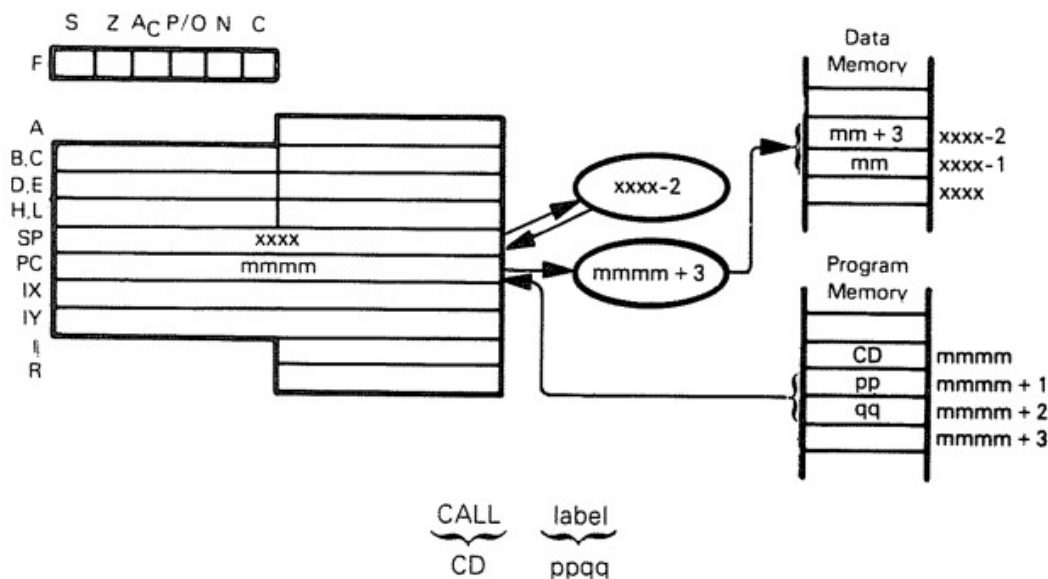
Store the address of the instruction following the CALL on the top of the stack: the top of the stack is a data memory byte addressed by the Stack Pointer. Then subtract 2 from the Stack Pointer in order to address the new top of stack. Move the 16-bit address contained in the second and third CALL instruction object program bytes to the Program Counter. The second byte of the CALL instruction is the low-order half of the address, and the third byte is the high-order byte.

Consider the instruction sequence:

```
        CALL    SUBR
        AND     7CH
        -
        -
        -
SUBR
```

After the instruction has executed, the address of the AND instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

CALL condition,label — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND IF CONDITION IS SATISFIED
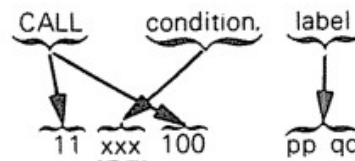
		Condition	Relevant Flag
000	NZ	Non-Zero	Z
001	Z	Zero	Z
010	NC	Non-Carry	C
011	C	Carry	C
100	PO	Parity Odd	P/O
101	PE	Parity Even	P/O
110	P	Sign Positive	S
111	M	Sign Negative	S

This instruction is identical to the CALL instruction. except that the identified subroutine will be called only if the condition is satisfied: otherwise. the instruction sequentially following the CALL condition instruction will be executed.

Consider the instruction sequence:
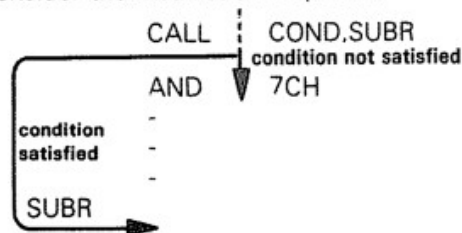
If the condition is not satisfied. the AND instruction will be executed after the CALL COND.SUBR instruction has executed. If the condition is satisfied. the address of the AND instruction is saved at the top of the stack, and the Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

CCF — COMPLEMENT CARRY FLAG

S Z A_C P/O N C

$$\underbrace{\text{CCF}}_{\text{3F}}$$
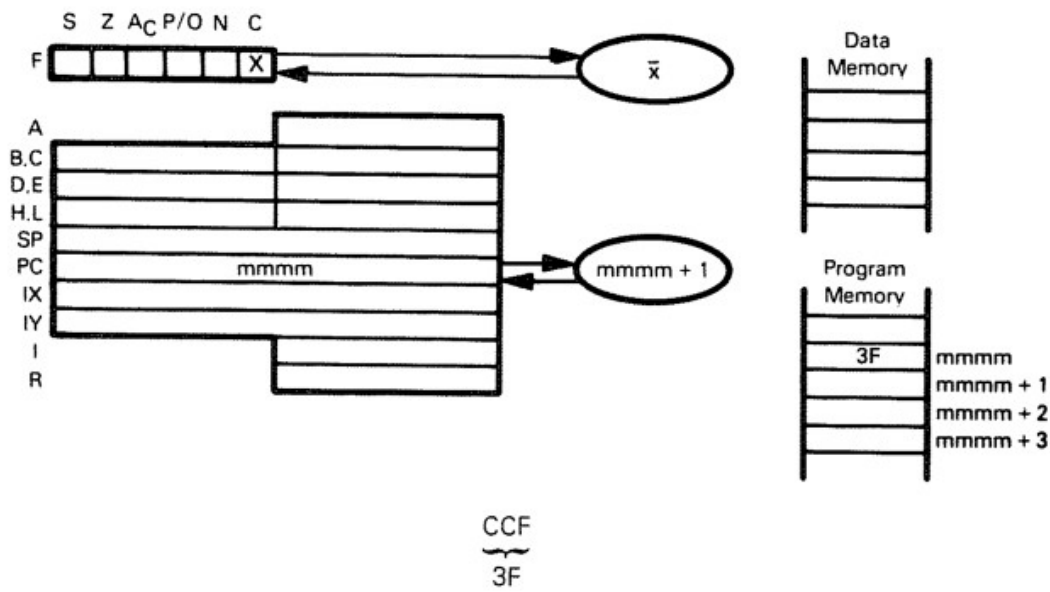
Complement the Carry flag. No other status or register contents are affected.

CP data — COMPARE IMMEDIATE DATA WITH ACCUMULATOR

$$\underbrace{CP}_{FE} \quad \underbrace{data}_{yy}$$
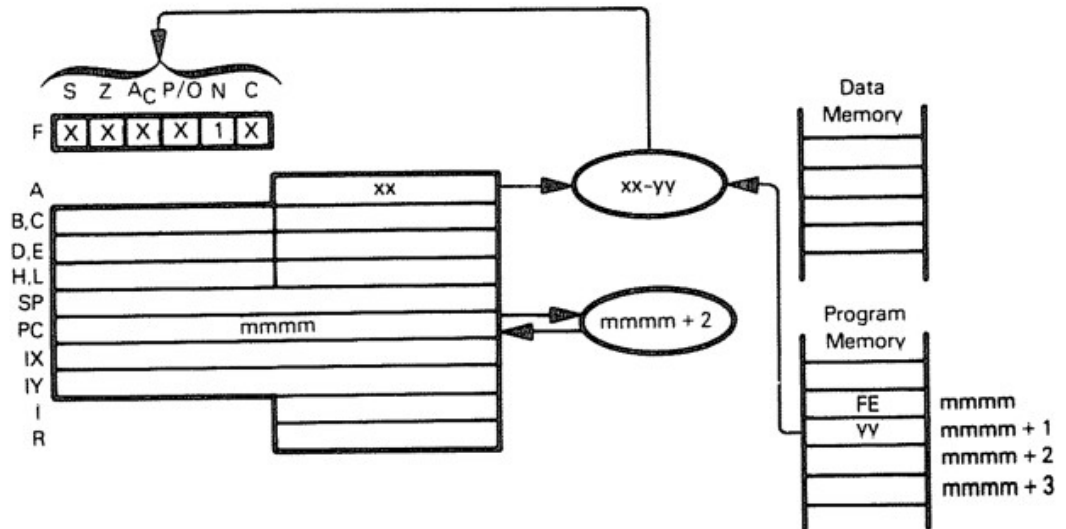
Subtract the contents of the second object code byte from the contents of the Accumulator, treating both numbers as simple binary data. Discard the result; i.e., leave the Accumulator alone, but modify the status flags to reflect the result of the subtraction.

Suppose $xx=E3_{16}$ and the second byte of the CP instruction object code contains $A0_{16}$. After the instruction

CP 0A0H

has executed, the Accumulator will still contain $E3_{16}$, but statuses will be modified as follows:
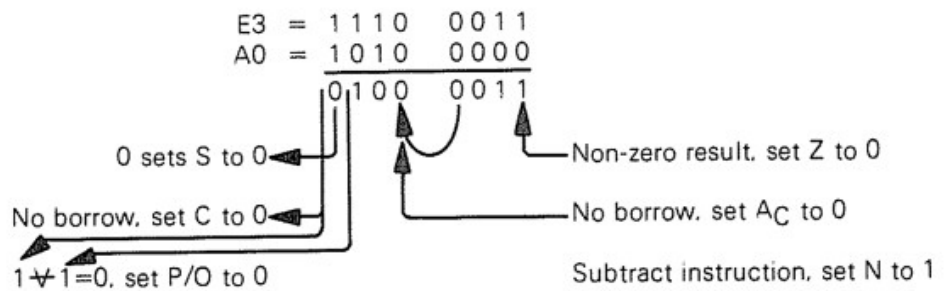
Notice that the resulting carry is complemented.

CP reg — COMPARE REGISTER WITH ACCUMULATOR
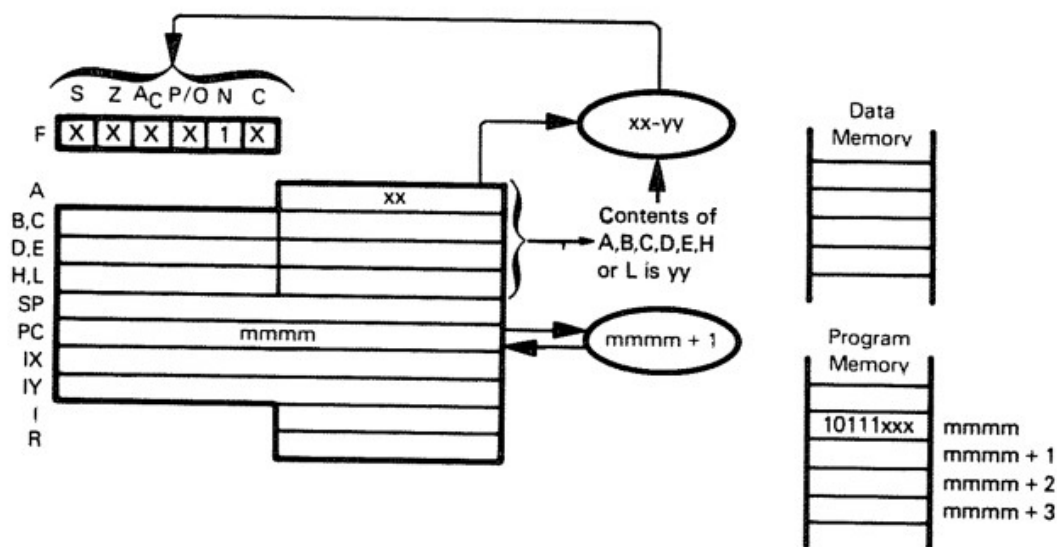

```
CP      reg
~~~~~   ~~~
10111   xxx
        ~~~
        000   for reg=B
        001   for reg=C
        010   for reg=D
        011   for reg=E
        100   for reg=H
        101   for reg=L
        111   for reg=A
```

Subtract the contents of Register A, B, C, D, E, H or L from the contents of the Accumulator, treating both numbers as simple binary data. Discard the result; i.e., leave the Accumulator alone, but modify status flags to reflect the result of the subtraction.

Suppose $xx=E3_{16}$ and Register B contains $A0_{16}$. After the instruction

$$CP \quad B$$

has executed, the Accumulator will still contain $E3_{16}$, but statuses will be modified as follows:

```
E3 = 1110  0011
A0 = 1010  0000
    0 1 0 0  0 0 1 1
```

0 sets S to 0 ◄─────────────── ─────── Non-zero result, set Z to 0

No borrow, set C to 0 ◄─────── ─────── No borrow, set A_C to 0

1 ⊻ 1=0, set P/O to 0   Subtract instruction, set N to 1

Notice that the resulting carry is complemented.

CP (HL) — COMPARE MEMORY WITH ACCUMULATOR
CP (IX+disp)
CP (IY+disp)
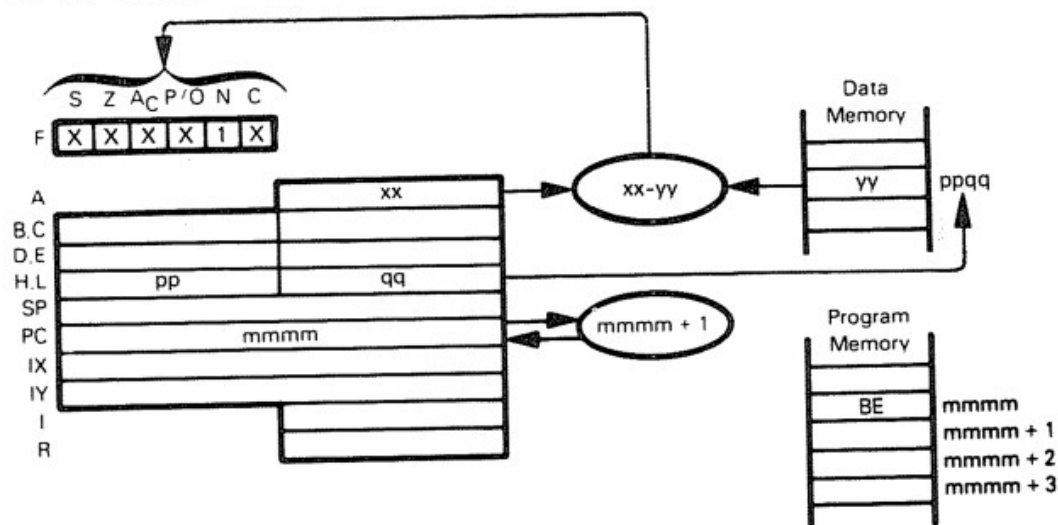
The illustration shows execution of CP (HL):

$$\underbrace{\text{CP (HL)}}_{\text{BE}}$$

Subtract the contents of memory location (specified by the contents of the HL register pair) from the contents of the Accumulator. treating both numbers as simple binary data. Discard the result: i.e.. leave the Accumulator alone. but modify status flags to reflect the result of the subtraction.

Suppose $xx=E3_{16}$ and $yy=A0_{16}$. After execution of

$$\text{CP (HL)}$$

the Accumulator will still contain $E3_{16}$. but statuses will be modified as follows:

```
E3 = 1110   0011
A0 = 0110   0000
    0,1 0 0   0 0 1 1
```

0 sets S to 0 ◄——

No borrow, set C to 0 ◄——

$1 \not\vee 1 = 0$, set P/O to 0

Non-zero result. set Z to 0

No borrow. set A_C to 0

Subtract instruction. set N to 1

Notice that the resulting carry is complemented.

$$\underbrace{\text{CP}}_{\text{DD}} \; \underbrace{\text{(IX+disp)}}_{\text{BE} \quad \text{d}}$$

Subtract the contents of memory location (specified by the sum of the contents of the IX register and the displacement value d) from the contents of the Accumulator, treating both numbers as simple binary data. Discard the result; i.e., leave the Accumulator alone, but modify status flags to reflect the result of the subtraction.

$$\underbrace{CP \ (IY+disp)}$$

$$\underbrace{FD \ BE} \quad \underbrace{d}$$

This instruction is identical to CP (IX+disp), except that it uses the IY register instead of the IX register.

CPD — COMPARE ACCUMULATOR WITH MEMORY. DECREMENT ADDRESS AND BYTE COUNTER
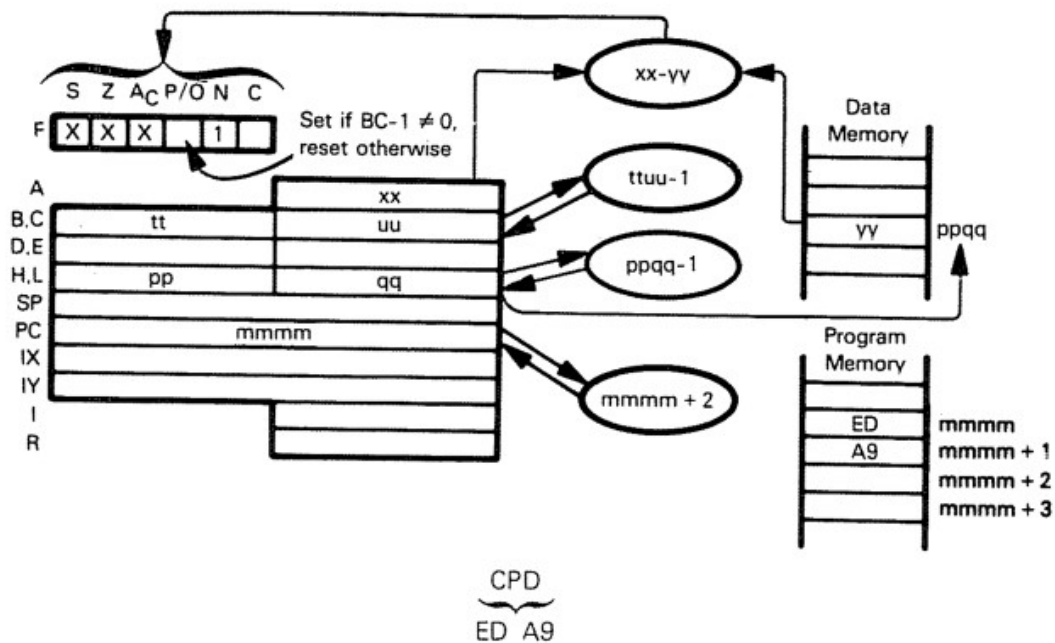
$$\underbrace{CPD}$$

$$\underbrace{ED \ A9}$$

Compare the contents of the Accumulator with the contents of memory location (specified by the HL register pair). If A is equal to memory, set Z flag. Decrement the HL and BC register pairs. (BC is used as the Byte Counter.)

Suppose xx=E3$_{16}$, ppqq=4000$_{16}$, BC contains 0001$_{16}$, and yy=A0$_{16}$. After the instruction

CPD

has executed, the Accumulator will still contain E3$_{16}$, but statuses will be modified as follows:

$$\begin{array}{rcl}
E3 & = & 1110 \quad 0011 \\
A0 & = & 1010 \quad 0000 \\
\hline
 & & 0100 \quad 0011
\end{array}$$

0 sets S to 0 ◄———

———Non-zero result. set Z to 0

———No borrow, set A$_C$ to 0

The P/O flag will be reset because BC-1=0

Subtract instruction involved, set N to 1

Carry not affected.
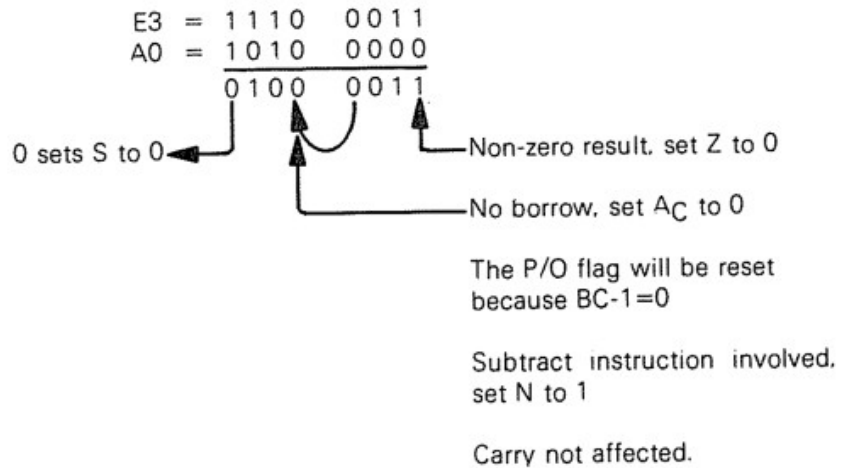
The HL register pair will contain 3FFF$_{16}$, and BC=0.

CPDR — COMPARE ACCUMULATOR WITH MEMORY. DECREMENT ADDRESS AND BYTE COUNTER. CONTINUE UNTIL MATCH IS FOUND OR BYTE COUNTER IS ZERO

CPDR
⌣
ED B9

This instruction is identical to CPD, except that it is repeated until a match is found or the byte counter is zero. After each data transfer, interrupts will be recognized and two refresh cycles will be executed.

Suppose the HL register pair contains 5000$_{16}$, the BC register pair contains 00FF$_{16}$, the Accumulator contains F9$_{16}$, and memory has contents as follows:

Location	Contents
5000$_{16}$	AA$_{16}$
4FFF$_{16}$	BC$_{16}$
4FFE$_{16}$	19$_{16}$
4FFD$_{16}$	7A$_{16}$
4FFC$_{16}$	F9$_{16}$
4FFB$_{16}$	DD$_{16}$

After execution of

CPDR

the P/O flag will be 1. the Z flag will be 1, the HL register pair will contain 4FFB$_{16}$, and the BC register pair will contain 00FA$_{16}$.

CPI — COMPARE ACCUMULATOR WITH MEMORY.
DECREMENT BYTE COUNTER.
INCREMENT ADDRESS
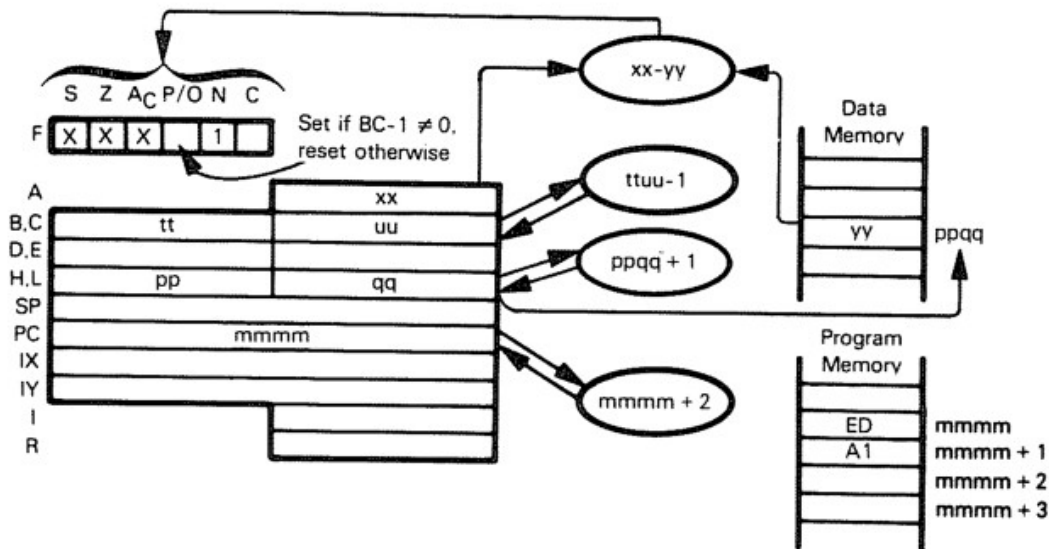
CPI
⌣
ED A1

Compare the contents of the Accumulator with the contents of memory location (specified by the HL register pair). If A is equal to memory, set the Z flag. Increment the HL register pair and decrement the BC register pair (BC is used as Byte Counter).

Suppose $xx=E3_{16}$, $ppqq=4000_{16}$, BC contains 0032_{16}, and $yy=E3_{16}$. After the instruction

CPI

has executed, the Accumulator will still contain $E3_{16}$, but statuses will be modified as follows:

$$
\begin{array}{rl}
E3 = & 1111 \quad 0011 \\
-E3 = & 0000 \quad 1101 \\
\hline
& 0000 \quad 0000
\end{array}
$$

0 sets S to 0 ◄——

—— Result is 0, set Z to 1

—— No borrow, set A_C to 0

The P/O flag will be set because BC-1 ≠ 0.

Subtract instruction involved, set N to 1.

Carry not affected.

The HL register pair will contain 4001_{16}, and BC will contain 0031_{16}.

CPIR — COMPARE ACCUMULATOR WITH MEMORY. DECREMENT BYTE COUNTER. INCREMENT ADDRESS. CONTINUE UNTIL MATCH IS FOUND OR BYTE COUNTER IS ZERO

CPIR

ED B1

This instruction is identical to CPI, except that it is repeated until a match is found or the byte counter is zero. After each data transfer interrupts will be recognized and two refresh cycles will be executed.

Suppose the HL register pair contains 4500_{16}, the BC register pair contains $00FF_{16}$, the Accumulator contains $F9_{16}$, and memory has contents as follows:

Location	Contents
4500_{16}	AA_{16}
4501_{16}	15_{16}
4502_{16}	$F9_{16}$

After execution of

CPIR

the P/O flag will be 1, and the Z flag will be 1. The HL register pair will contain 4503_{16}, and the BC register pair will contain $00FC_{16}$.

CPL — COMPLEMENT THE ACCUMULATOR

S Z A$_C$ P/O N C

```
F [  |  | 1 |  | 1 |  ]
```

A   [xx   ] ◄ (x̄x̄)
B,C
D,E
H,L
SP
PC   [mmmm   ] ◄ (mmmm + 1)
IX
IY
I
R

Data Memory

Program Memory

2F   mmmm
      mmmm + 1
      mmmm + 2
      mmmm + 3

CPL
‿
2F
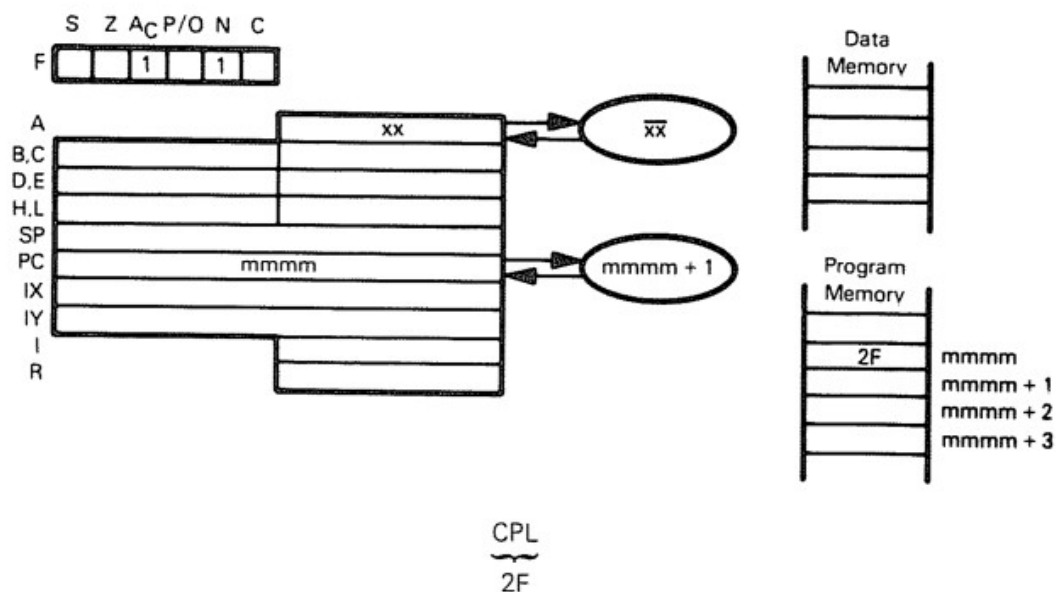
Complement the contents of the Accumulator. No other register's contents are affected.

Suppose the Accumulator contains $3A_{16}$. After the instruction

CPL

has executed, the Accumulator will contain $C5_{16}$.

$$3A = 0011 \quad 1010$$
$$Complement = 1100 \quad 0101$$

This is a routine logical instruction. You need not use it for binary subtraction; there are special subtract instructions (SUB, SBC).

3-67

DAA — DECIMAL ADJUST ACCUMULATOR
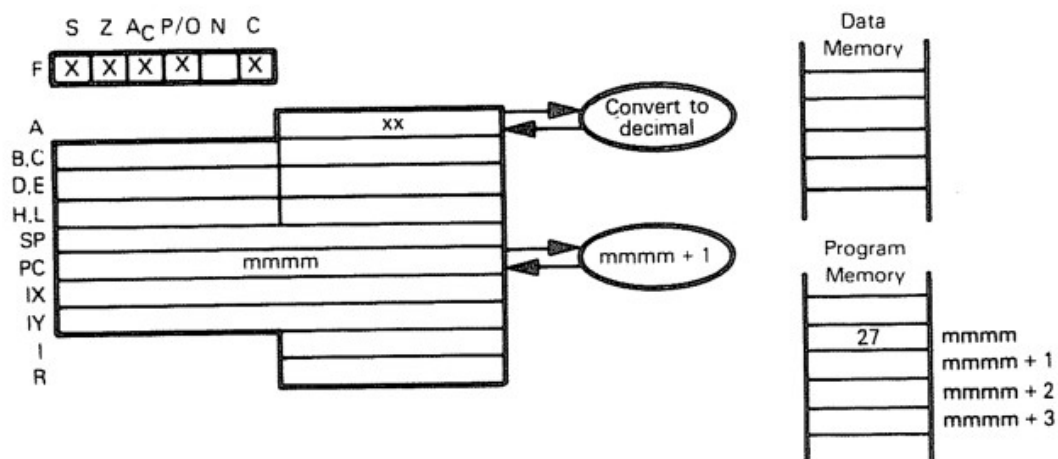
$$\underbrace{\text{DAA}}_{27}$$

Convert the contents of the Accumulator to binary-coded decimal form. This instruction should only be used after adding or subtracting two BCD numbers; i.e., look upon ADD DAA or ADC DAA or INC DAA or SUB DAA or SBC DAA or DEC DAA or NEG DAA as compound, decimal arithmetic instructions which operate on BCD sources to generate BCD answers.

Suppose the Accumulator contains 39_{16} and the B register contains 47_{16}. After the instructions

```
ADD  B
DAA
```

have executed, the Accumulator will contain 86_{16}, not 80_{16}.

Z80 CPU logic uses the values in the Carry and Auxiliary Carry, as well as the Accumulator contents, in the Decimal Adjust operation.

DEC reg — DECREMENT REGISTER CONTENTS


```
DEC   reg
 00   xxx   101

      000   for reg=B
      001   for reg=C
      010   for reg=D
      011   for reg=E
      100   for reg=H
      101   for reg=L
      111   for reg=A
```
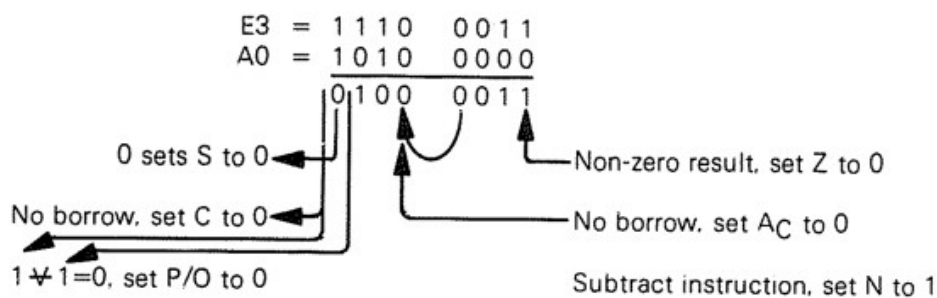
Subtract 1 from the contents of the specified register.

Suppose Register A contains 50_{16}. After execution of

DEC A

Register A will contain $4F_{16}$.

DEC rp — DECREMENT CONTENTS OF SPECIFIED REGISTER
DEC IX   PAIR
DEC IY
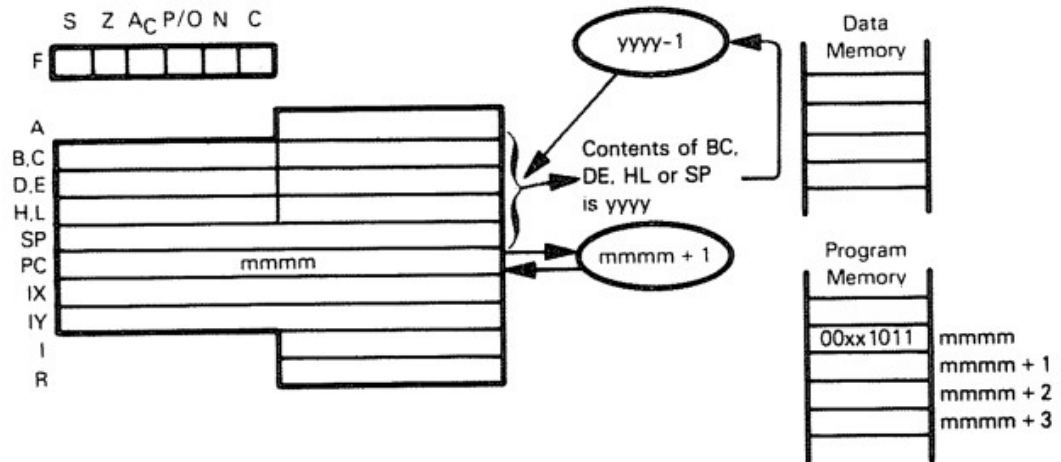
The illustration shows execution of DEC rp:
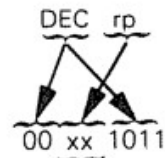
DEC   rp

00 xx 1011

00 for rp is register pair BC
01 for rp is register pair DE
10 for rp is register pair HL
11 for rp is Stack Pointer

Subtract 1 from the 16-bit value contained in the specified register pair. No status flags are affected.

Suppose the H and L registers contain $2F00_{16}$. After the instruction

DEC HL

has executed, the H and L registers will contain $2EFF_{16}$.

DEC IX

DD 2B

Subtract 1 from the 16-bit value contained in the IX register.

DEC IY

FD 2B

Subtract 1 from the 16-bit value contained in the IY register.

Neither DEC rp, DEC IX nor DEC IY affects any of the status flags. This is a defect in the Z80 instruction set, inherited from the 8080. Whereas the DEC reg instruction is used in iterative instruction loops that use a counter with a value of 256 or less, the DEC rp (DEC IX or DEC IY) instruction must be used if the counter value is more than 256. Since the DEC rp instruction sets no status flags, other instructions must be added to simply

test for a zero result. This is a typical loop form:

```
        LD      DE,DATA     :LOAD INITIAL 16-BIT COUNTER VALUE
LOOP    -                   :FIRST INSTRUCTION OF LOOP
        -
        -
        DEC     DE          :DECREMENT COUNTER
        LD      A,D         :TO TEST FOR ZERO, MOVE D TO A
        OR      E           :THEN OR A WITH E
        JP      NZ,LOOP     :RETURN IF NOT ZERO
```

DEC (HL) — DECREMENT MEMORY CONTENTS
DEC (IX+disp)
DEC (IY+disp)
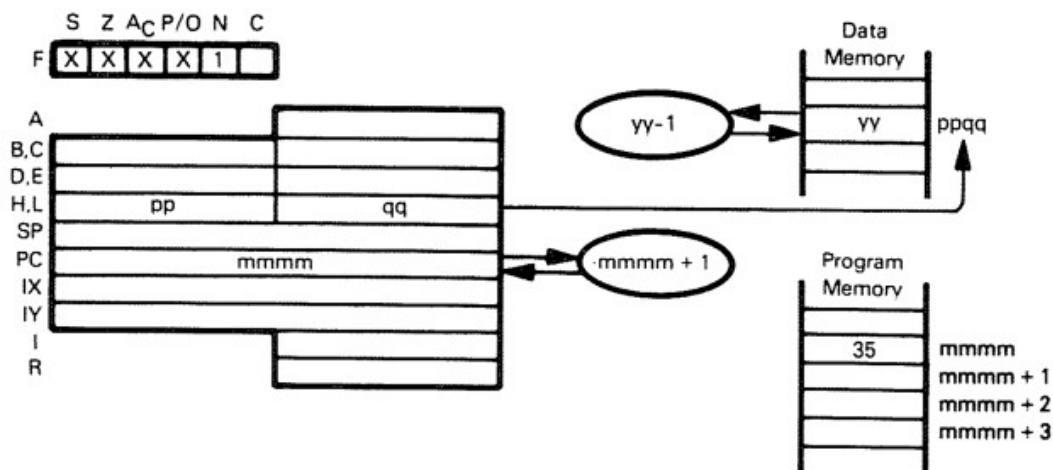
The illustration shows execution of DEC (HL):

$$\underbrace{DEC\ (HL)}_{35}$$

Subtract 1 from the contents of memory location (specified by the contents of the HL register pair).

Suppose $ppqq=4500_{16}$, $yy=5F_{16}$. After execution of

$$DEC\ (HL)$$

memory location 4500_{16} will contain $5E_{16}$.
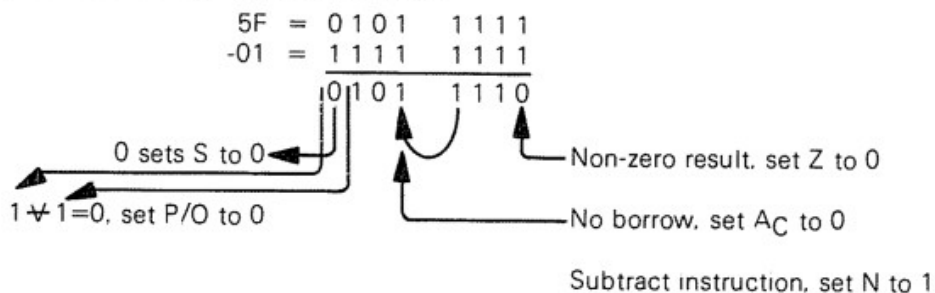
$$\underbrace{\text{DEC (IX+disp)}}_{} \underbrace{}_{}$$

$$\text{DD 35   d}$$

Subtract 1 from the contents of memory location (specified by the sum of the contents of the IX register and the displacement value d).

$$\underbrace{\text{DEC (IY+disp)}}_{} \underbrace{}_{}$$

$$\text{FD 35   d}$$

This instruction is identical to DEC (IX+disp), except that it uses the IY register instead of the IX register.

DI — DISABLE INTERRUPTS

$$\underbrace{\text{DI}}_{\text{F3}}$$
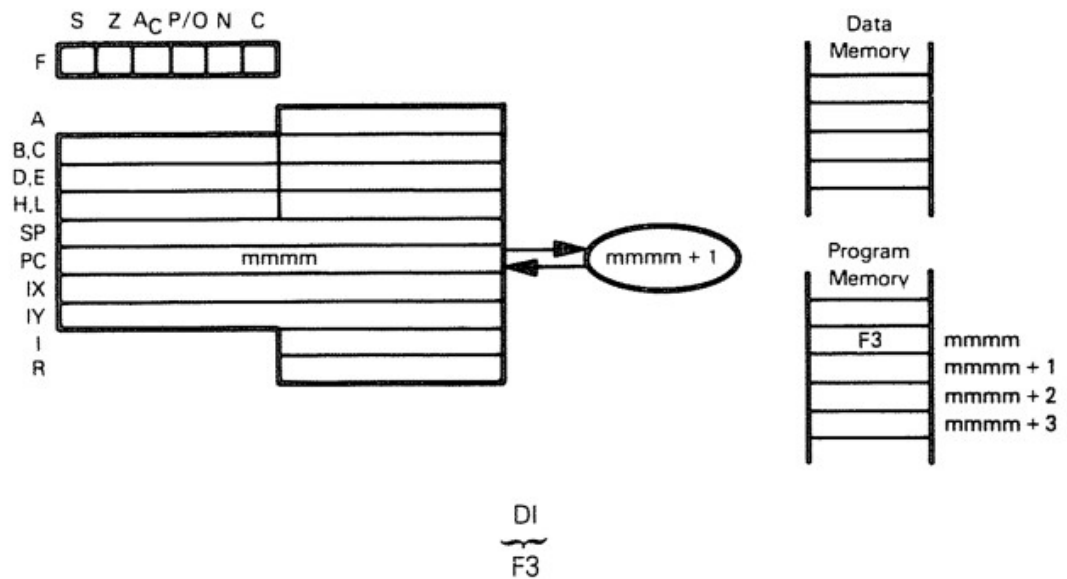
When this instruction is executed, the maskable interrupt request is disabled and the INT input to the CPU will be ignored. Remember that when an interrupt is acknowledged, the maskable interrupt is automatically disabled.

The maskable interrupt request remains disabled until it is subsequently enabled by an EI instruction.

No registers or flags are affected by this instruction.

DJNZ disp — JUMP RELATIVE TO PRESENT CONTENTS OF PROGRAM COUNTER IF REG B IS NOT ZERO

$$\underbrace{\text{DJNZ}}_{10} \quad \underbrace{\text{disp}}_{\text{dd-2}}$$
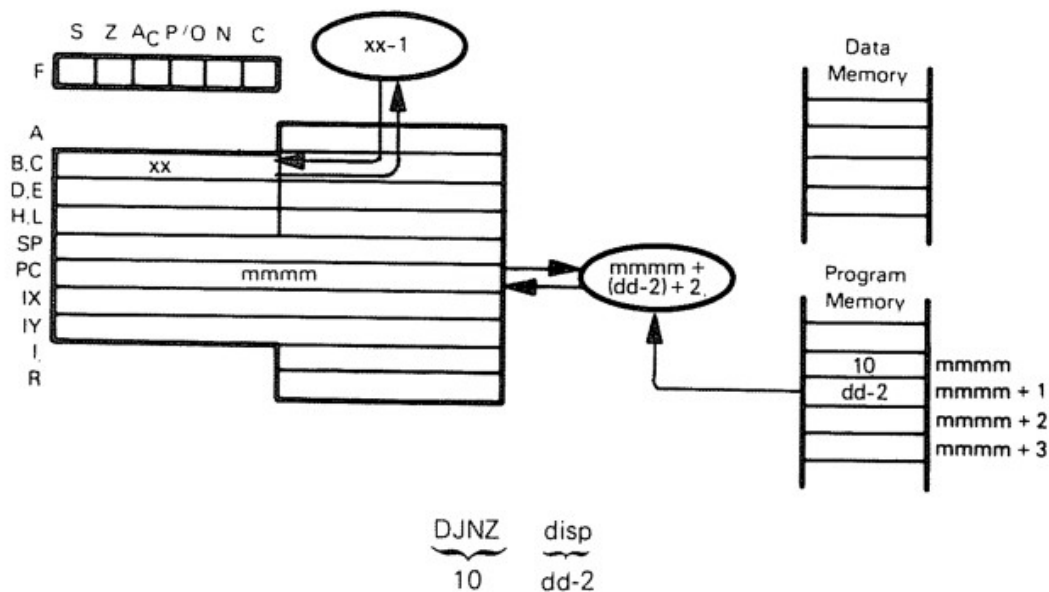
Decrement Register B. If remaining contents are not zero, add the contents of the DJNZ instruction object code second byte and 2 to the Program Counter. The jump is measured from the address of the instruction operation code, and has a range of -126 to +129 bytes. The Assembler automatically adjusts for the twice-incremented PC.

If the contents of B are zero after decrementing, the next sequential instruction is executed.

The DJNZ instruction is extremely useful for any program loop operation, since the one instruction replaces the typical "decrement-then-branch on condition" instruction sequence.

EI — ENABLE INTERRUPTS
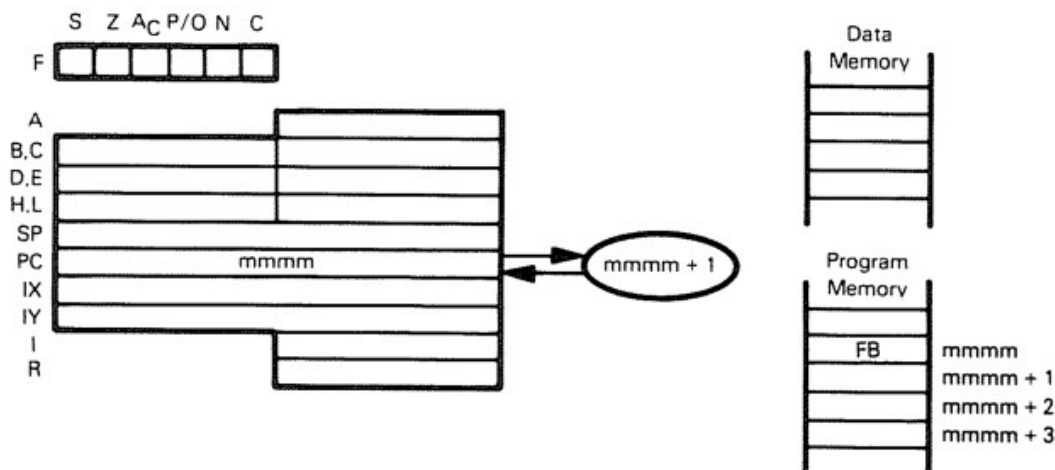
Execution of this instruction causes interrupts to be enabled. but not until one more instruction executes.

Most interrupt service routines end with the two instructions:

```
EI              ;ENABLE INTERRUPTS
RET             :RETURN TO INTERRUPTED PROGRAM
```

If interrupts are processed serially. then for the entire duration of the interrupt service routine all maskable interrupts are disabled — which means that in a multi-interrupt application there is a significant possibility for one or more interrupts to be pending when any interrupt service routine completes execution.

If interrupts were acknowledged as soon as the EI instructions had executed, then the Return instruction would not be executed. Under these circumstances, returns would stack up one on top of the other — and unnecessarily consume stack memory space. This may be illustrated as follows:
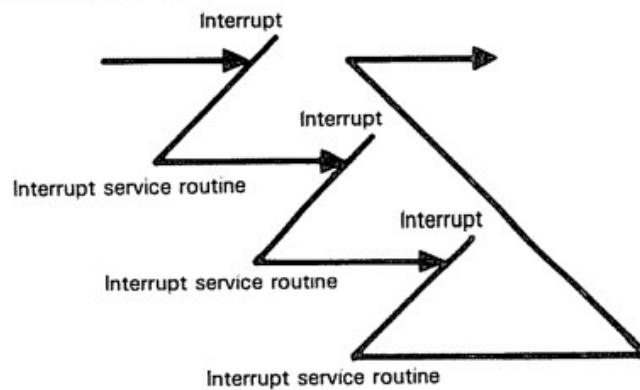
By inhibiting interrupts for one more instruction following execution of EI, the Z80 CPU ensures that the RET instruction gets executed in the sequence:

```
-
-
-
EI              ;ENABLE INTERRUPTS
RET             ;RETURN FROM INTERRUPT
```

It is not uncommon for interrupts to be kept disabled while an interrupt service routine is executing. Interrupts are processed serially:

3-74
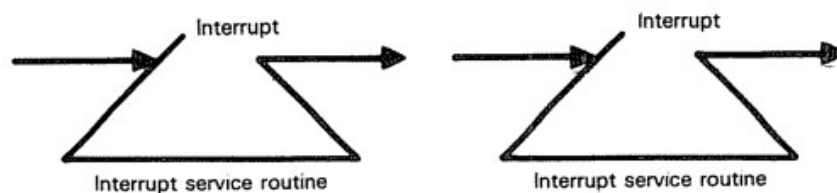
EX AF,AF' — EXCHANGE PROGRAM STATUS AND ALTERNATE PROGRAM STATUS
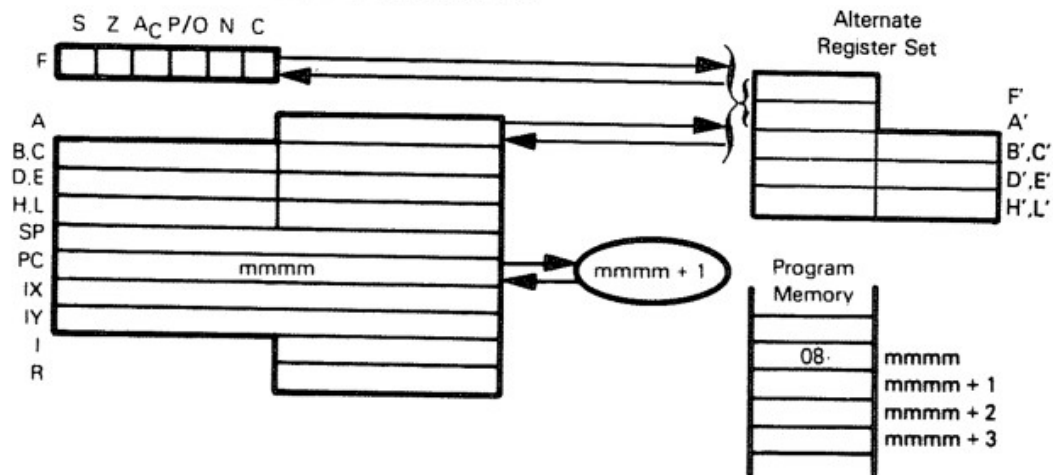
The two-byte contents of register pairs AF and A'F' are exchanged.

Suppose AF contains $4F99_{16}$ and A'F' contains $10AA_{16}$. After execution of

EX AF,AF'

AF will contain $10AA_{16}$ and AF' will contain $4F99_{16}$.

EX DE,HL — EXCHANGE DE AND HL CONTENTS
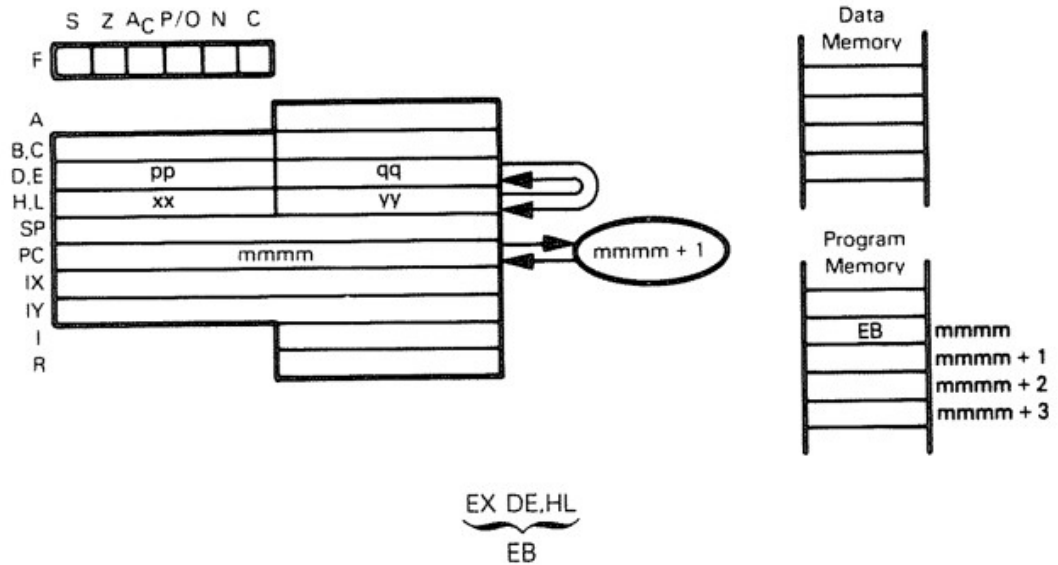
$$\underbrace{\text{EX DE,HL}}_{\text{EB}}$$

The D and E registers' contents are swapped with the H and L registers' contents.

Suppose $pp=03_{16}$, $qq=2A_{16}$, $xx=41_{16}$ and $yy=FC_{16}$. After the instruction

EX DE,HL

has executed, H will contain 03_{16}, L will contain $2A_{16}$, D will contain 41_{16} and E will contain FC_{16}.

The two instructions:

EX DE,HL
LD A,(HL)

are equivalent to:

LD A,(DE)

but if you want to load data addressed by the D and E register into the B register,

EX DE,HL
LD B,(HL)

has no single instruction equivalent.

EX (SP),HL — EXCHANGE CONTENTS OF REGISTER AND
EX (SP),IX   TOP OF STACK
EX (SP),IY
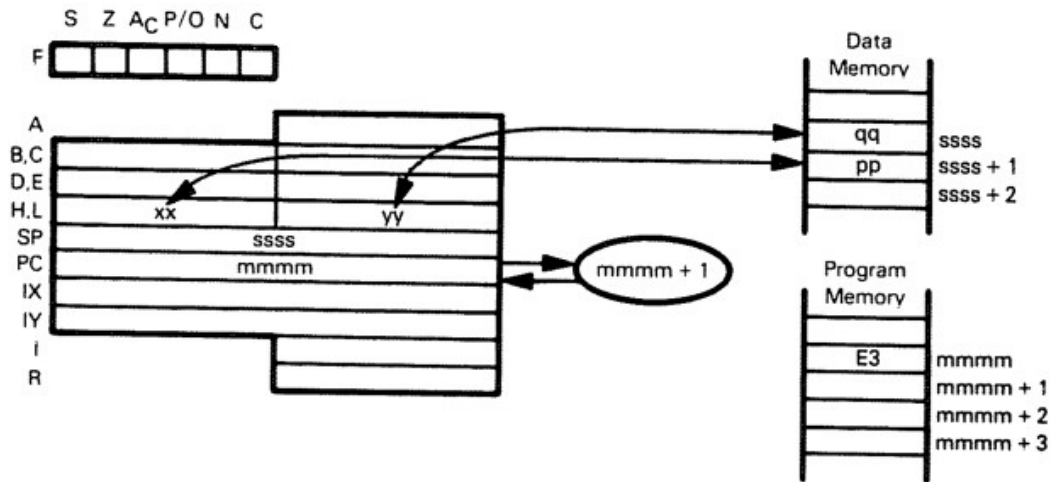
The illustration shows execution of EX (SP),HL.

$$\underbrace{\text{EX (SP),HL}}_{\text{E3}}$$

Exchange the contents of the L register with the top stack byte. Exchange the contents of the H register with the byte below the stack top.

Suppose $xx=21_{16}$, $yy=FA_{16}$, $pp=3A_{16}$, $qq=E2_{16}$. After the instruction

EX (SP),HL

has executed, H will contain $3A_{16}$, L will contain $E2_{16}$ and the two top stack bytes will contain FA_{16} and 21_{16} respectively.

The EX (SP),HL instruction is used to access and manipulate data at the top of the stack.

$$\underbrace{\text{EX (SP),IX}}_{\text{DD E3}}$$

Exchange the contents of the IX register's low-order byte with the top stack byte. Exchange the IX register's high-order byte with the byte below the stack top.

$$\underbrace{\text{EX (SP),IY}}_{\text{FD E3}}$$

This instruction is identical to EX (SP),IX, but uses the IY register instead of the IX register.

EXX — EXCHANGE REGISTER PAIRS AND ALTERNATE REGISTER PAIRS
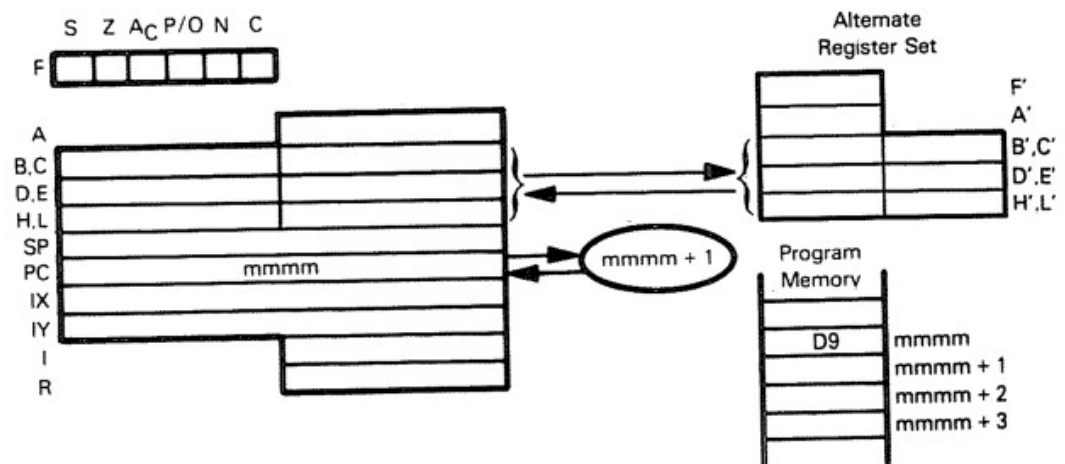
$$\underbrace{\text{EXX}}_{\text{D9}}$$

The contents of register pairs BC, DE and HL are swapped with the contents of register pairs B'C', D'E', and H'L'.

Suppose register pairs BC, DE and HL contain 4901_{16}, $5F00_{16}$ and 7251_{16} respectively, and register pairs B'C', D'E', H'L' contain 0000_{16}, $10FF_{16}$ and 3333_{16} respectively. After the execution of

EXX

the registers will have the following contents:

BC: 0000_{16}; DE: $10FF_{16}$; HL: 3333_{16};
B'C': 4901_{16}; D'E': $5F00_{16}$; H'L': 7251_{16}

This instruction can be used to exchange register banks to provide very fast interrupt response times.

HALT


```
        S  Z AC P/O N  C
   F   [  ][ ][ ][ ][ ][ ]

   A
  B,C
  D,E
  H,L
   SP
   PC          mmmm              mmmm + 1
   IX
   IY
   I
   R
```

Data Memory

Program Memory

```
        76      mmmm
                mmmm + 1
                mmmm + 2
                mmmm + 3
```
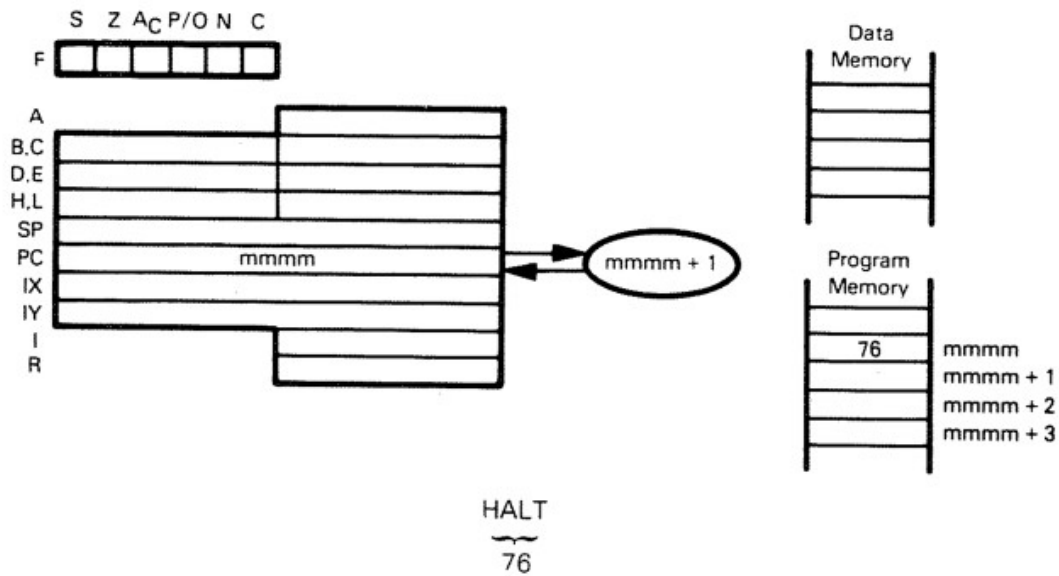
HALT
⏝
76

When the HALT instruction is executed, program execution ceases. The CPU requires an interrupt or a reset to restart execution. No registers or statuses are affected; however, memory refresh logic continues to operate.

IM 0 — INTERRUPT MODE 0
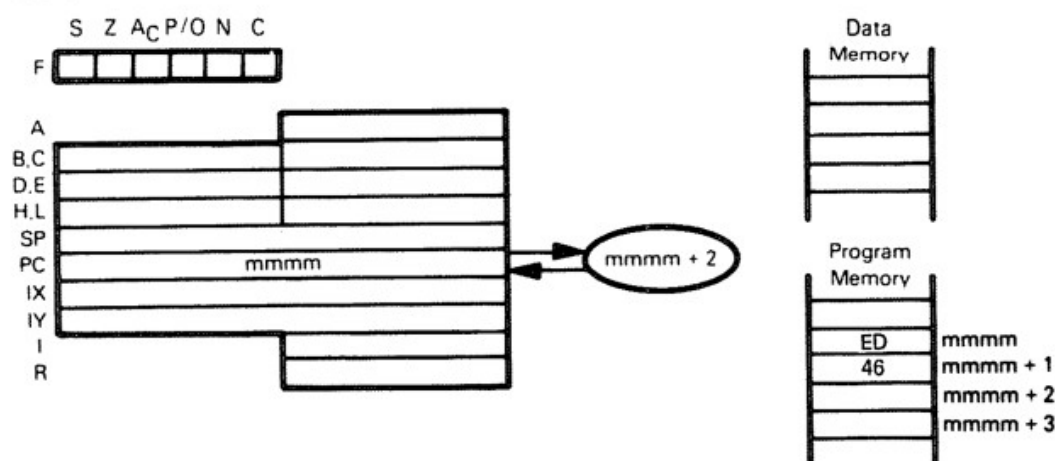
$$\underbrace{\text{IM 0}}_{\text{ED 46}}$$

This instruction places the CPU in interrupt mode 0. In this mode, the interrupting device will place an instruction on the Data Bus and the CPU will then execute that instruction. No registers or statuses are affected.

IM 1 — INTERRUPT MODE 1

$$\underbrace{\text{IM 1}}_{\text{ED 56}}$$

This instruction places the CPU in interrupt mode 1. In this mode, the CPU responds to an interrupt by executing a restart (RST) to location 0038_{16}.

IM 2 — INTERRUPT MODE 2

$$\underbrace{\text{IM 2}}_{\text{ED 5E}}$$

This instruction places the CPU in interrupt mode 2. In this mode, the CPU performs an indirect call to any specified location in memory. A 16-bit address is formed using the contents of the Interrupt Vector (I) register for the upper eight bits, while the lower eight bits are supplied by the interrupting device. Refer to Chapter 12 for a full description of interrupt modes. No registers or statuses are affected by this instruction.

IN A,(port) — INPUT TO ACCUMULATOR
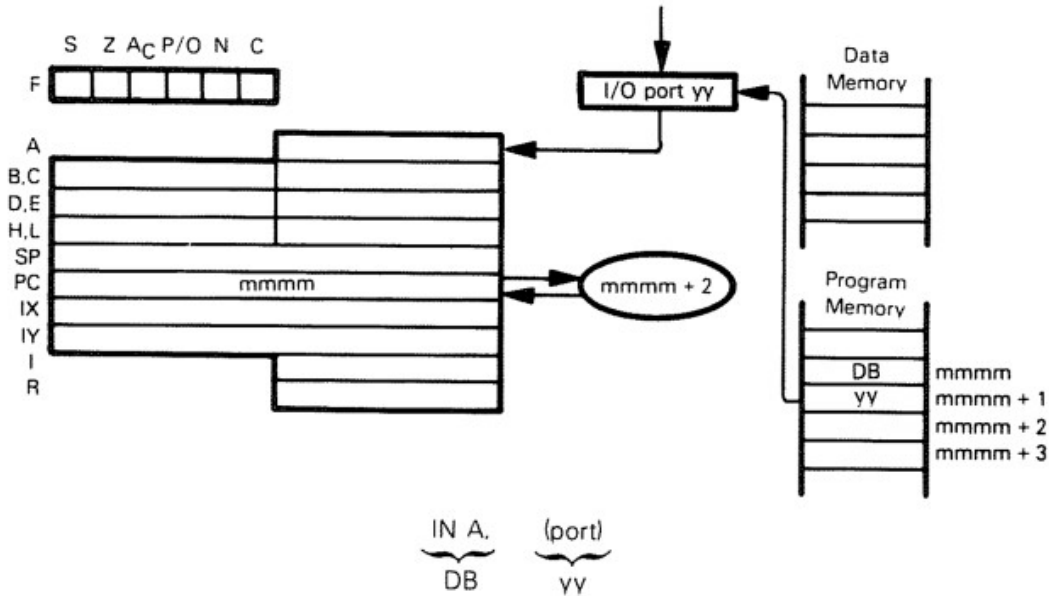
Load a byte of data into the Accumulator from the I/O port (identified by the second IN instruction object code byte).

Suppose 36_{16} is held in the buffer of I/O port $1A_{16}$. After the instruction

IN A,(1AH)

has executed, the Accumulator will contain 36_{16}.

The IN instruction does not affect any statuses.

Use of the IN instruction is very hardware dependent. Valid I/O port addresses are determined by the way in which I/O logic has been implemented. It is also possible to design a microcomputer system that accesses external logic using memory reference instructions with specific memory addresses.

INC reg — INCREMENT REGISTER CONTENTS

```
S  Z  Ac P/O  N  C
F  X  X  X   X   0
```

yy + 1

Contents of A, B, C, D, E, H or L is yy

Data Memory

```
A
B,C
D,E
H,L
SP
PC        mmmm
IX
IY
I
R
```

mmmm + 1

Program Memory

```
00xxx100   mmmm
           mmmm + 1
           mmmm + 2
           mmmm + 3
```

INC reg

00 xxx 100

```
000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A
```
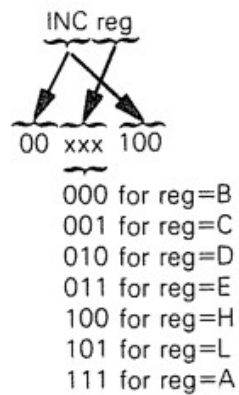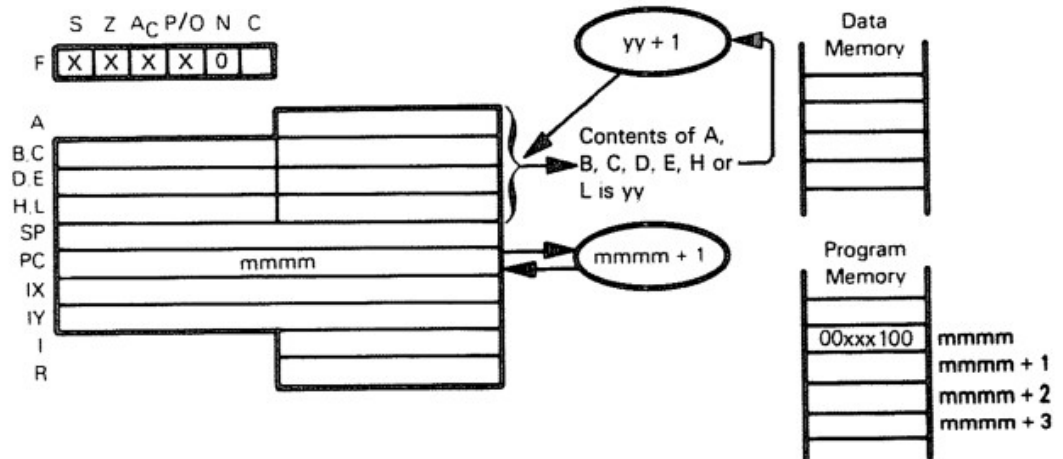
Add 1 to the contents of the specified register.

Suppose Register E contains $A8_{16}$. After execution of

$$INC \ E$$

Register E will contain $A9_{16}$.

INC rp — INCREMENT CONTENTS OF SPECIFIED REGISTER PAIR
INC IX
INC IY
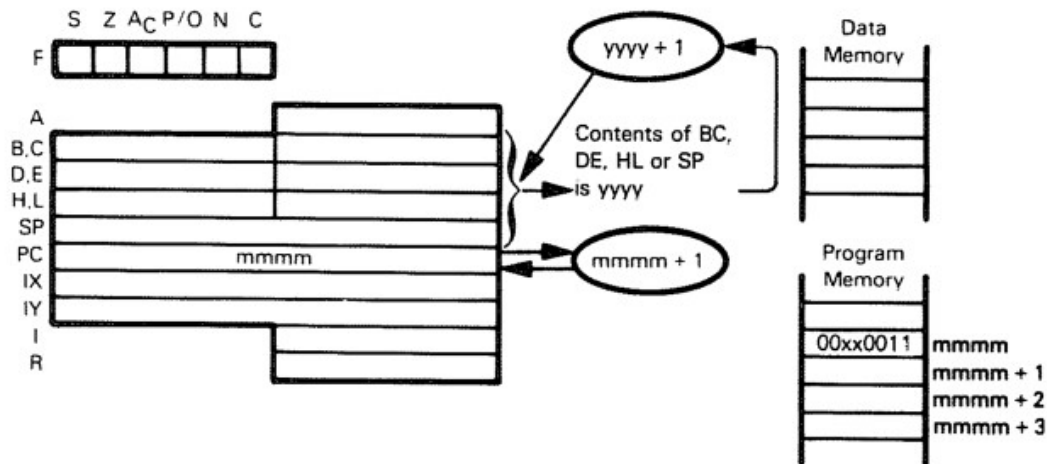
The illustration shows execution of INC rp:
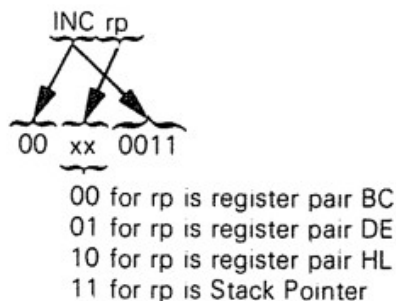
INC rp

00 xx 0011

00 for rp is register pair BC
01 for rp is register pair DE
10 for rp is register pair HL
11 for rp is Stack Pointer

Add 1 to the 16-bit value contained in the specified register pair. No status flags are affected.

Suppose the D and E registers contain $2F7A_{16}$. After the instruction

INC DE

has executed, the D and E registers will contain $2F7B_{16}$.

INC IX

DD 23

Add 1 to the 16-bit value contained in the IX register.

INC IY

FD 23

Add 1 to the 16-bit value contained in the IY register.

Just like the DEC rp, DEC IX and DEC IY, neither INC rp, INC IX nor INC IY affects any status flags. This is a defect in the Z80 instruction set inherited from the 8080.

INC (HL) — INCREMENT MEMORY CONTENTS
INC (IX+disp)
INC (IY+disp)
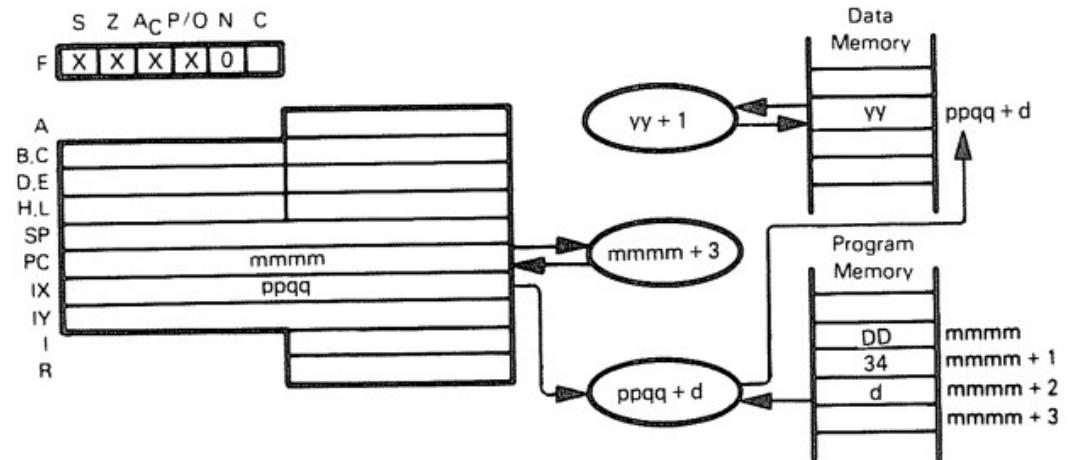
The illustration shows execution of INC (IX+d):

$$\underbrace{\text{INC (IX+disp)}}_{} \quad \underbrace{}_{}$$

DD 34   d

Add 1 to the contents of memory location (specified by the sum of the contents of Register IX and the displacement value d).

Suppose ppqq=4000_{16} and memory location $400F_{16}$ contains 36_{16}. After execution of the instruction

INC (IX+0FH)

memory location $400F_{16}$ will contain 37_{16}.
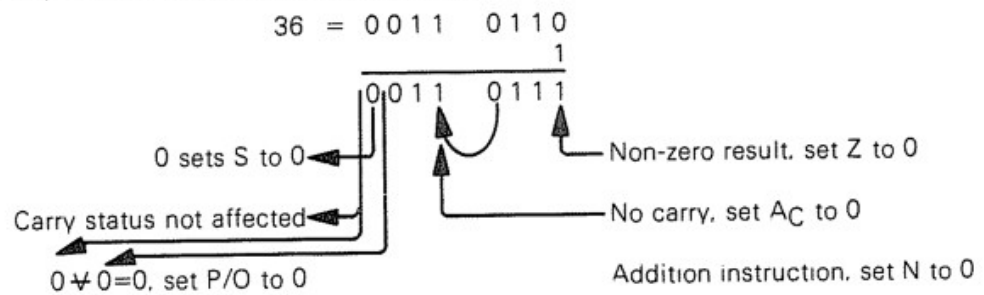
$$\underbrace{\text{INC (IY+disp)}}_{}$$

FD 34   d

This instruction is identical to INC (IX+disp), except that it uses the IY register instead of the IX register.

$$\underbrace{\text{INC (HL)}}_{34}$$

Add 1 to the contents of memory location (specified by the contents of the HL register pair).

IND — INPUT TO MEMORY AND DECREMENT POINTER
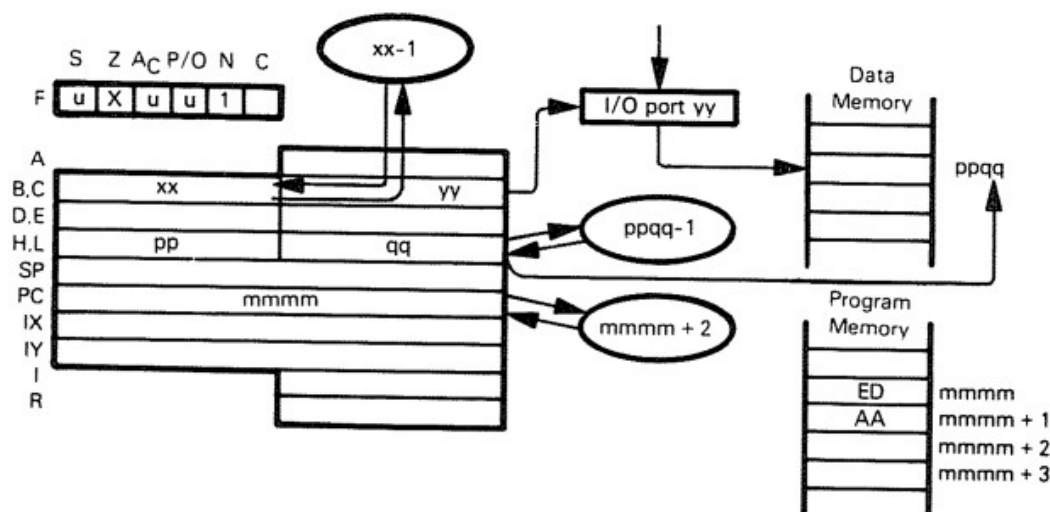
$$\underbrace{\text{IND}}_{\text{ED AA}}$$

Input from I/O port (addressed by Register C) to memory location (specified by HL). Decrement Registers B and HL.

Suppose $xx=05_{16}$, $yy=15_{16}$, $ppqq=2400_{16}$, and 19_{16} is held in the buffer of I/O port 15_{16}. After the instruction

IND

has executed, memory location 2400_{16} will contain 19_{16}. The B register will contain 04_{16} and the HL register pair $23FF_{16}$.

INDR — INPUT TO MEMORY AND DECREMENT POINTER UNTIL BYTE COUNTER IS ZERO

$$\underbrace{\text{INDR}}_{\text{ED BA}}$$

INDR is identical to IND, but is repeated until Register B=0.

Suppose Register B contains 03_{16}, Register C contains 15_{16}, and HL contains 2400_{16}. The following sequence of bytes is available at I/O port 15_{16}:

$$17_{16}, 59_{16} \text{ and } AE_{16}$$

After the execution of

INDR

the HL register pair will contain $23FD_{16}$ and Register B will contain zero, and memory locations will have contents as follows:

Location	Contents
2400	17_{16}
23FF	59_{16}
23FE	AE_{16}

This instruction is extremely useful for loading blocks of data from an input device into memory.

INI — INPUT TO MEMORY AND INCREMENT POINTER
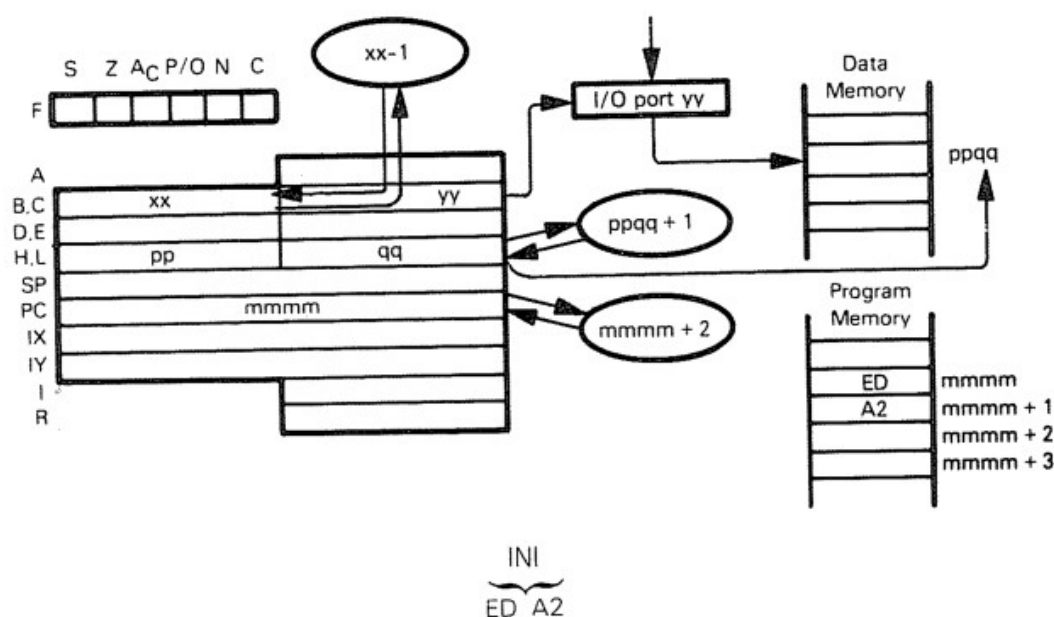
$$\underbrace{\text{INI}}_{\text{ED A2}}$$

Input from I/O port (addressed by Register C) to memory location (specified by HL). Decrement Register B; increment register pair HL.

Suppose $xx=05_{16}$, $yy=15_{16}$, $ppqq=2400_{16}$, and 19_{16} is held in the buffer of I/O port 15_{16}.

After the instruction

<div align="center">INI</div>

has executed, memory location 2400_{16} will contain 19_{16}. The B register will contain 04_{16} and the HL register pair 2401_{16}.

INIR — INPUT TO MEMORY AND INCREMENT POINTER UNTIL BYTE COUNTER IS ZERO

$$\underbrace{\text{INIR}}_{\text{ED B2}}$$

INIR is identical to INI, but is repeated until Register B=0.

Suppose Register B contains 03_{16}, Register C contains 15_{16}, and HL contains 2400_{16}. The following sequence of bytes is available at I/O port 15_{16}:

<div align="center">17_{16}, 59_{16} and AE_{16}</div>

After the execution of

<div align="center">INIR</div>

the HL register pair will contain 2403_{16} and Register B will contain zero, and memory locations will have contents as follows:

Location	Contents
2400	17_{16}
2401	59_{16}
2402	AE_{16}

This instruction is extremely useful for loading blocks of data from a device into memory.

IN reg,(C) — INPUT TO REGISTER

```
   S  Z A C P/O N  C
F  X  X  0  X  0                           I/O port yy
```

```
A
B,C                              yy                    Register
D,E                                                  A, B, C, D, E,
H,L                                                     H or L
SP
PC              mmmm                                  mmmm + 2
IX
IY
I
R
```

Data Memory

Program Memory

```
ED        mmmm
01xxx000  mmmm + 1
          mmmm + 2
          mmmm + 3
```

```
       IN  reg, (C)


ED 01 xxx 000
```
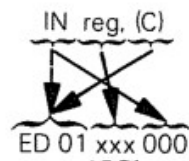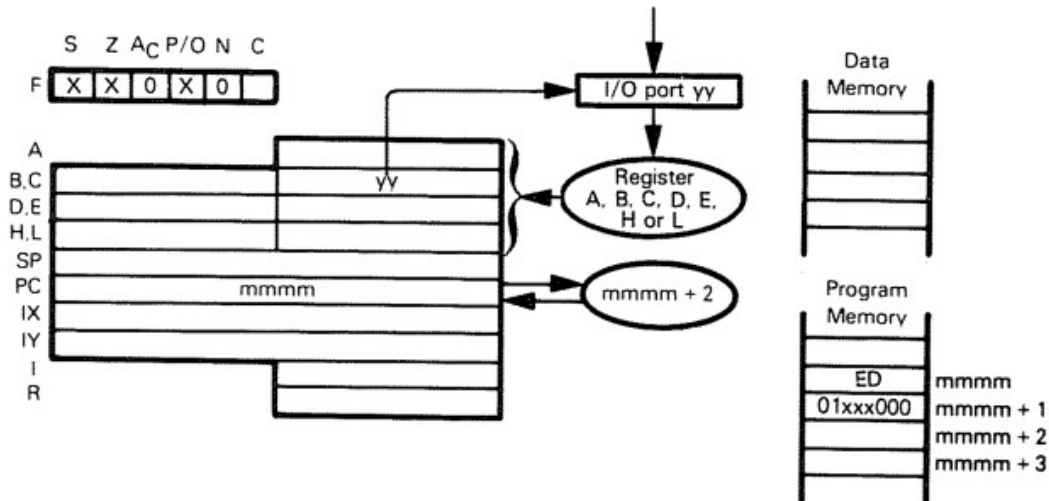
000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A
110 for setting of status flags without
    changing registers

Load a byte of data into the specified register (reg) from the I/O port (identified by the contents of the C register).

Suppose 42_{16} is held in the buffer of I/O port 36_{16}, and Register C contains 36_{16}. After the instruction

IN D,(C)

has executed, the D register will contain 42_{16}.

During the execution of the instruction, the contents of Register B are placed on the top half of the Address Bus, making it possible to extend the number of addressable I/O ports.

JP label — JUMP TO THE INSTRUCTION IDENTIFIED IN THE OPERAND

S   Z A_C P/O N   C
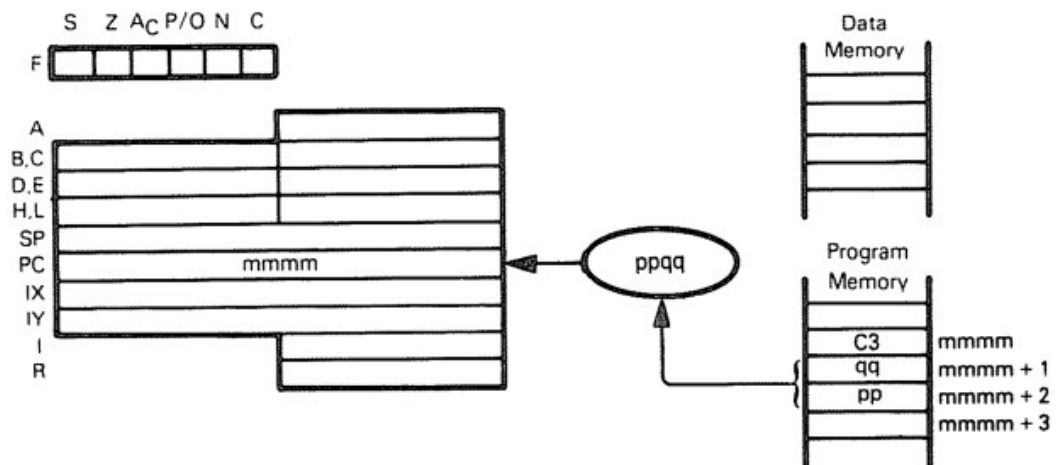

```
JP   label
C3   ppqq
```

Load the contents of the Jump instruction object code second and third bytes into the Program Counter; this becomes the memory address for the next instruction to be executed. The previous Program Counter contents are lost.

In the following sequence:

```
       JP    NEXT
       AND   7FH
       -
       -
NEXT   CPL
```

The CPL instruction will be executed after the JP instruction. The AND instruction will never be executed, unless a Jump instruction somewhere else in the instruction sequence jumps to this instruction.

JP condition,label — JUMP TO ADDRESS IDENTIFIED IN THE OPERAND IF CONDITION IS SATISIFED

JP cond, label

11 cc 010 ppqq
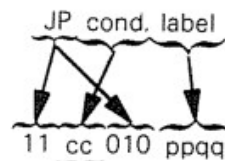
		Condition	Relevant Flag
000	NZ	Non-Zero	Z
001	Z	Zero	Z
010	NC	No Carry	C
011	C	Carry	C
100	PO	Parity Odd	P/O
101	PE	Parity Even	P/O
110	P	Sign Positive	S
111	M	Sign Negative	S

This instruction is identical to the JP instruction. except that the jump will be performed only if the condition is satisfied: otherwise. the instruction sequentially following the JP condition instruction will be executed.

Consider the instruction sequence

```
                  JP      COND.LABEL
                          condition not satisfied
          AND     7CH
condition
satisfied          -
                   -
     LABEL  OR       B
```
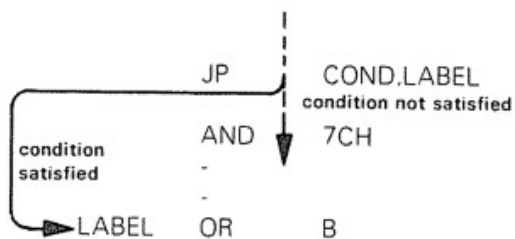
After the JP cond.label instruction has executed. if the condition is satisfied then the OR instruction will be executed. If the condition is not satisfied, the AND instruction. being the next sequential instruction. is executed.

JP (HL) — JUMP TO ADDRESS SPECIFIED BY CONTENTS
JP (IX)   OF 16-BIT REGISTER
JP (IY)
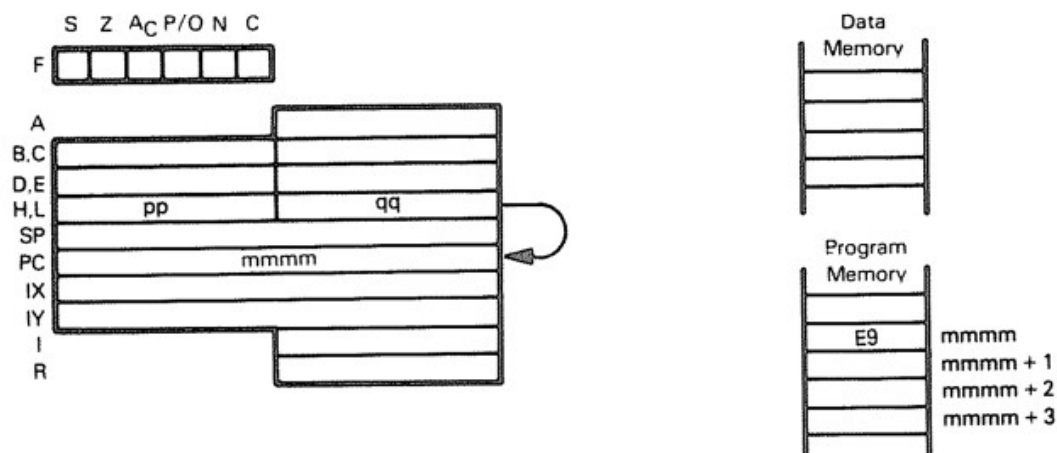
The illustration shows execution of JP (HL):

$$\underbrace{JP \ (HL)}_{E9}$$

The contents of the HL register pair are moved to the Program Counter; therefore, an implied addressing jump is performed.

The instruction sequence

```
LD      H.ADDR
JP      (HL)
```

has exactly the same net effect as the single instruction

```
JP      ADDR
```

Both specify that the instruction with label ADDR is to be executed next.

The JP (HL) instruction is useful when you want to increment a return address for a subroutine that has multiple returns.

Consider the following call to subroutine SUB:

```
CALL    SUB     ;CALL SUBROUTINE
JP      ERR     :ERROR RETURN
                :GOOD RETURN
```

Using RET to return from SUB would return execution of JP ERR; therefore, if SUB executes without detecting error conditions. return as follows:

```
POP     HL      :POP RETURN ADDRESS TO HL
INC     HL      :ADD 3 TO RETURN ADDRESS
INC     HL
INC     HL
JP      (HL)    :RETURN
```

$$\underbrace{JP \ (IX)}_{DD \ E9}$$

This instruction is identical to the JP (HL) instruction. except that it uses the IX register

instead of the HL register pair.

$$\underbrace{\text{JP (IY)}}_{\text{FD E9}}$$

This instruction is identical to the JP (HL) instruction. except that it uses the IY register instead of the HL register pair.

JR C,disp — JUMP RELATIVE TO CONTENTS OF PROGRAM COUNTER IF CARRY IS SET

$$\underbrace{\text{JR C,}}\;\underbrace{\text{disp}}$$
$$\text{38}\qquad\text{dd-2}$$

This instruction is identical to the JR disp instruction. except that the jump is only executed if the Carry status equals 1. otherwise, the next instruction is executed.

In the following instruction sequence:

```
        4000    JR      C,$+8
                                c=0
        4002    AND         7FH
  C=1           -
                -
     4008    OR          B
```
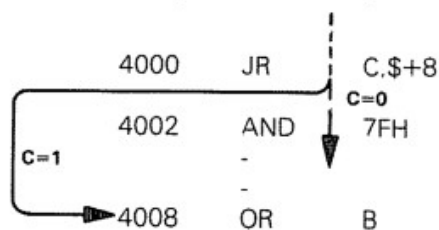
After the JR C,$+8 instruction. the OR instruction is executed if the Carry status equals 1. The AND instruction is executed if the Carry status equals 0.

JR disp — JUMP RELATIVE TO PRESENT CONTENTS OF PROGRAM COUNTER

JR disp

18 dd-2
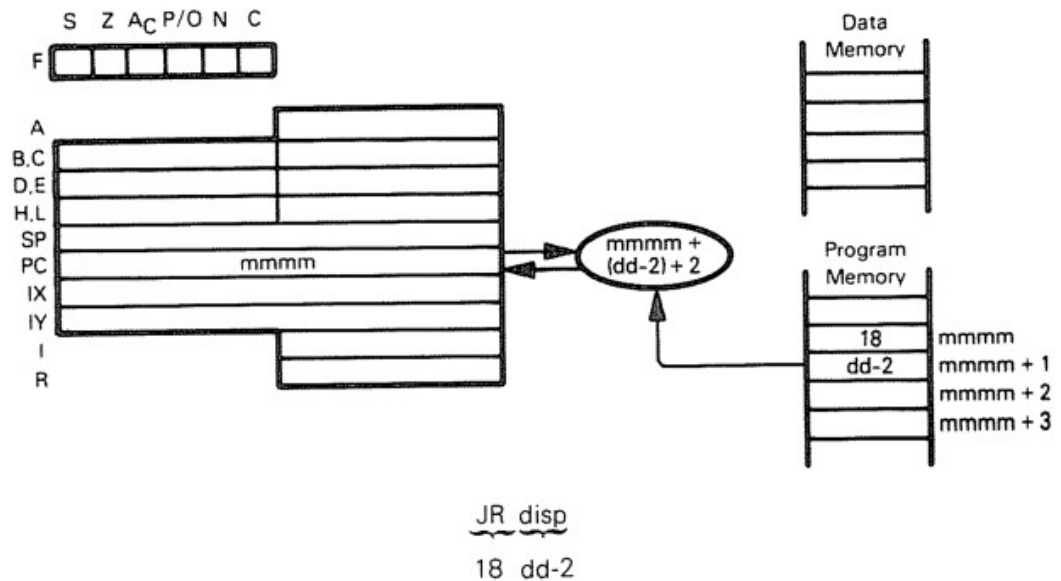
Add the contents of the JR instruction object code second byte, the contents of the Program Counter, and 2. Load the sum into the Program Counter. The jump is measured from the address of the instruction operation code, and has a range of -126 to +129 bytes. The Assembler automatically adjusts for the twice-incremented PC.

The following assembly language statement is used to jump four steps forward from address 4000_{16}.

JR $+4

Result of this instruction is shown below:

Location	Instruction	
4000	18	
4001	02	
4002	-	
4003	-	
4004	-	◄——— new PC value

JR NC,disp — JUMP RELATIVE TO CONTENTS OF PROGRAM COUNTER IF CARRY FLAG IS RESET

JR NC,disp

30   dd-2

This instruction is identical to the JR disp instruction. except that the jump is only executed if the Carry status equals 0; otherwise, the next instruction is executed.

In the following instruction sequence:

```
          4000    ADD      A,7FH
          4001
C=0       4002              C=1
          4003    JR       NC,$-3
          4005    OR       B
```
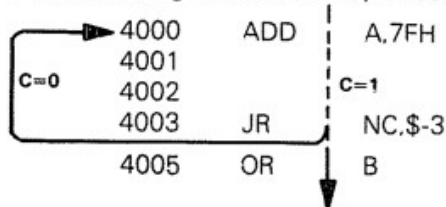
After the JR NC,$-3 instruction. the OR instruction is executed if the Carry status equals 1. The ADD instruction is executed if the Carry status equals 0.

JR NZ,disp — JUMP RELATIVE TO CONTENTS OF PROGRAM COUNTER IF ZERO FLAG IS RESET

JR NZ,disp

20   dd-2

This instruction is identical to the JR disp instruction. except that the jump is only executed if the Zero status equals 0: otherwise. the next instruction is executed.

In the following instruction sequence:

```
          4000    JR       NZ,$+6
          4002    AND      7FH
Z=0       4004    -                Z=1
          4005    -
          4006    OR       B
```
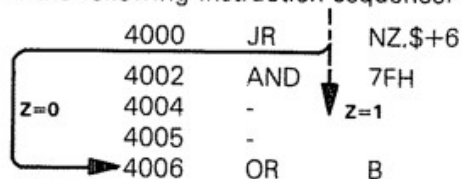
After the JR NZ,$+6 instruction. the OR instruction is executed if the Zero status equals 0. The AND instruction is executed if the Zero status equals 1.

JR Z,disp — JUMP RELATIVE TO CONTENTS OF PROGRAM COUNTER IF ZERO FLAG IS SET

$$\underbrace{\text{JR Z,disp}}_{\text{28 \quad dd-2}}$$

This instruction is identical to the JR disp instruction, except that the jump is only executed if the Zero status equals 1; otherwise, the next instruction is executed.

In the following instruction sequence:

```
            4000      JR        Z,$+6
            4002      AND       7FH
  z=1       4004      -            z=0
            4005      -
         ┌► 4006      OR        B
```
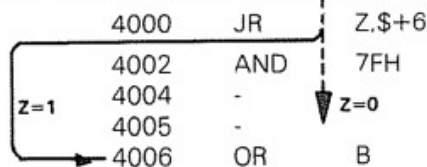
After the JR Z,$+6 instruction, the OR instruction is executed if the Zero status equals 1. The AND instruction is executed if the Zero status equals 0.

LD A,I — MOVE CONTENTS OF INTERRUPT VECTOR OR
LD A,R   REFRESH REGISTER TO ACCUMULATOR
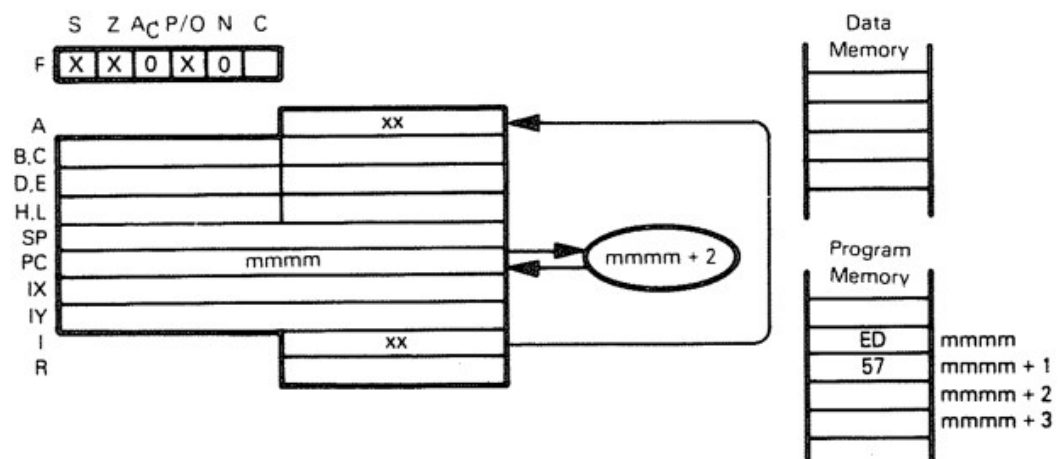
The illustration shows execution of LD A,I:

$$\underbrace{\text{LD A,I}}_{\text{ED \quad 57}}$$

Move the contents of the Interrupt Vector register to the Accumulator, and reflect interrupt enable status in Parity/Overflow flag.

Suppose the Interrupt Vector register contains $7F_{16}$, and interrupts are disabled. After execution of

LD A,I

Register A will contain $7F_{16}$, and P/O will be 0.

$$\underbrace{\text{LD A,R}}_{\text{ED \quad 5F}}$$

Move the contents of the Refresh register to the Accumulator. The value of the interrupt flip-flop will appear in the Parity/Overflow flag.

LD A,(addr) — LOAD ACCUMULATOR FROM MEMORY USING DIRECT ADDRESSING
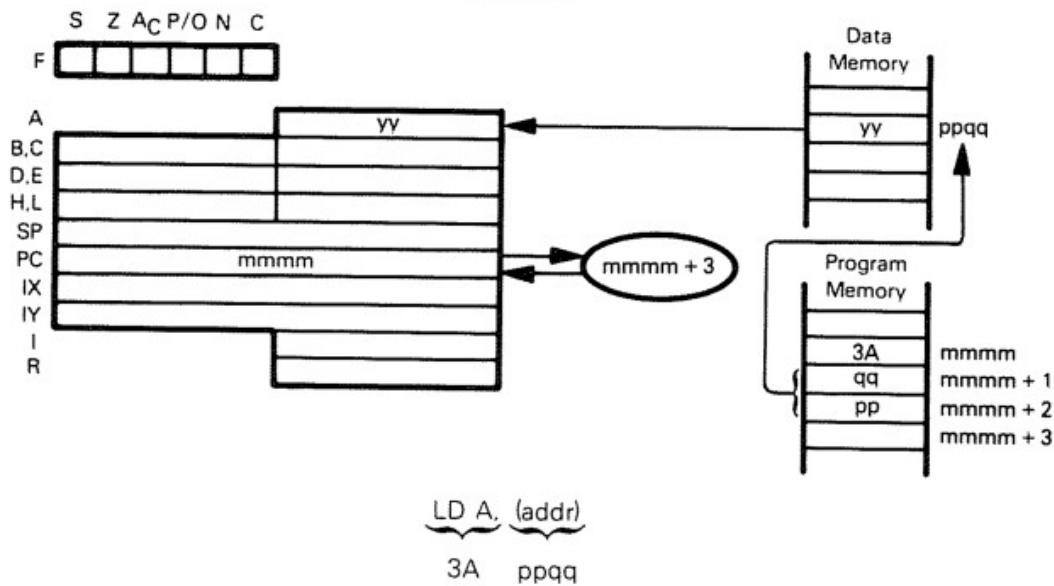
$$\underbrace{\text{LD A, (addr)}}$$

$$\text{3A} \quad \text{ppqq}$$

Load the contents of the memory byte (addressed directly by the second and third bytes of the LD A,(addr) instruction object code) into the Accumulator. Suppose memory byte $084A_{16}$ contains 20_{16}. After the instruction

```
label    EQU      084AH
 -
 -
 -
LD       A,(label)
```

has executed, the Accumulator will contain 20_{16}.

Remember that EQU is an assembler directive rather than an instruction; it tells the Assembler to use the 16-bit value $084A_{16}$ wherever the label appears.

The instruction

```
LD       A,(label)
```

is equivalent to the two instructions

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

When you are loading a single value from memory, the LD A,(label) instruction is preferred; it uses one instruction and three object program bytes to do what the LD HL,label, LD A,(HL) combination does in two instructions and four object program bytes. Also, the LD HL,label, LD A,(HL) combination uses the H and L registers, which LD A,(label) does not.

LD A,(rp) — LOAD ACCUMULATOR FROM MEMORY LOCATION
ADDRESSED BY REGISTER PAIR


```
LD A,(rp)
```

000 x 1010

0 if register pair=BC
1 if register pair=DE
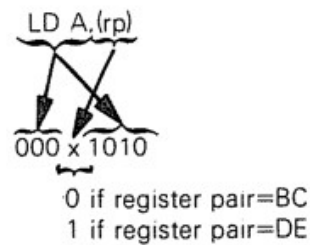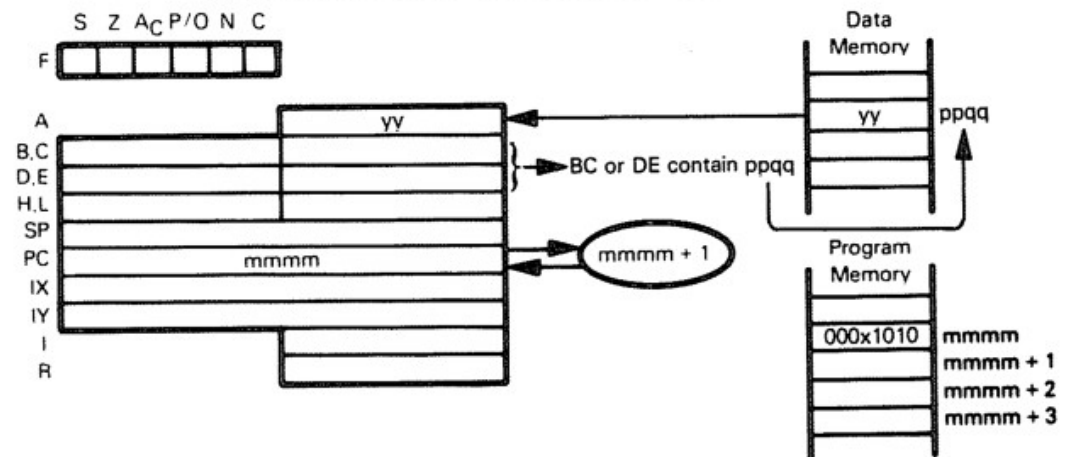
Load the contents of the memory byte (addressed by the BC or DE register pair) into the Accumulator.

Suppose the B register contains 08_{16}, the C register contains $4A_{16}$, and memory byte $084A_{16}$ contains $3A_{16}$. After the instruction

        LD A,(BC)

has executed, the Accumulator will contain $3A_{16}$.

Normally, the LD A,(rp) and LD rp,data will be used together, since the LD rp,data instruction loads a 16-bit address into the BC or DE registers as follows:

     LD   BC,084AH
     LD   A,(BC)

LD dst,src — MOVE CONTENTS OF SOURCE REGISTER TO DESTINATION REGISTER

```
S  Z AcP/O N  C
F [  ][  ][  ][  ][  ][  ]
```

A
B,C
D,E
H,L
SP
PC   mmmm
IX
IY
I
R

Register A, B, C, D, E, H or L

Register A, B, C D, E, H, L

mmmm + 1

Data Memory

Program Memory

01dddsss	mmmm
	mmmm + 1
	mmmm + 2
	mmmm + 3

LD dst, src

01 ddd sss
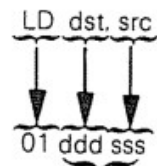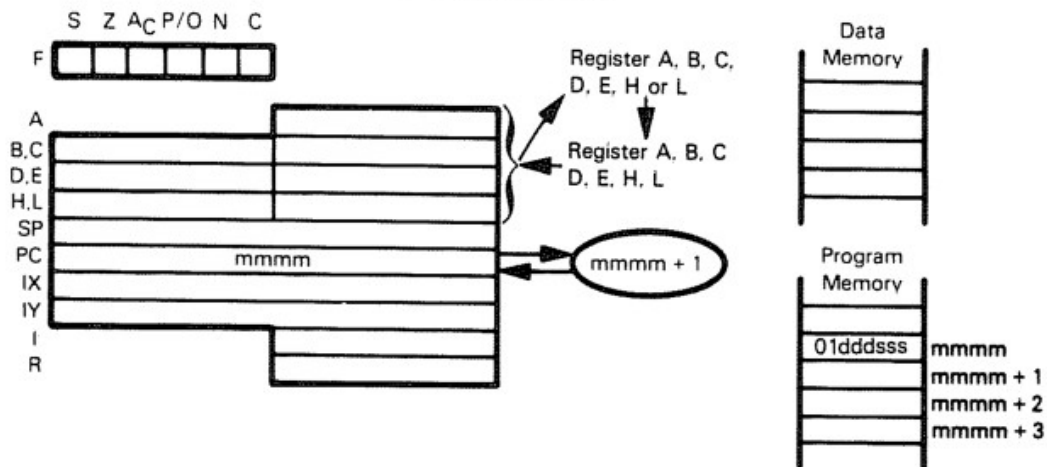
```
000 for dst or src=B
001 for dst or src=C
010 for dst or src=D
011 for dst or src=E
100 for dst or src=H
101 for dst or src=L
111 for dst or src=A
```

The contents of any designated register are loaded into any other register.

For example:

LD A,B

loads the contents of Register B into Register A.

LD L,D

loads the contents of Register D into Register L.

LD C,C

does nothing, since the C register has been specified as both the source and the destination.

LD HL, (addr) — LOAD REGISTER PAIR OR INDEX REGISTER
LD rp, (addr)   FROM MEMORY USING DIRECT ADDRESSING
LD IX, (addr)
LD IY, (addr)
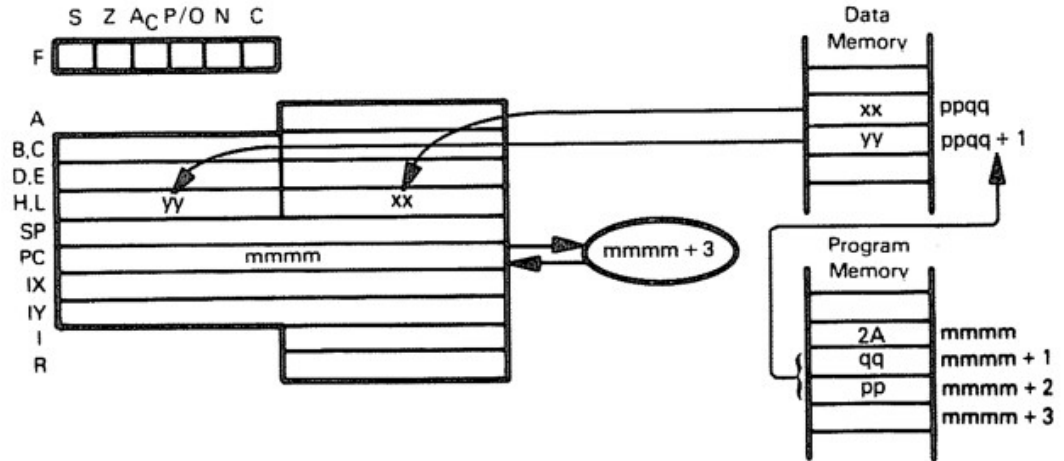
The illustration shows execution of LD HL(ppqq):

LD HL.addr
2A   ppqq

Load the HL register pair from directly addressed memory location.

Suppose memory location 4004_{16} contains AD_{16} and memory location 4005_{16} contains 12_{16}. After the instruction

LD HL,(4004H)

has executed, the HL register pair will contain $12AD_{16}$.
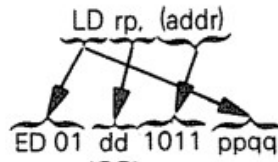
LD rp, (addr)

ED 01 dd 1011 ppqq

00 for rp is register pair BC
01 for rp is register pair DE
10 for rp is register pair HL
11 for rp is Stack Pointer

Load register pair from directly addressed memory.

Suppose memory location $49FF_{16}$ contains BE_{16} and memory location $4A00_{16}$ contains 33_{16}. After the instruction

LD DE,(49FFH)

has executed, the DE register pair will contain $33BE_{16}$.

LD IX,(addr)

DD 2A ppqq

Load IX register from directly addressed memory.

Suppose memory location D111$_{16}$ contains FF$_{16}$ and memory location D112$_{16}$ contains 56$_{16}$. After the instruction

<p style="text-align:center">LD IX,(D111H)</p>

has executed. the IX register will contain 56FF$_{16}$.

<p style="text-align:center">LD IY,(addr)
⏝ ⏝
FD 2A ppqq</p>

Load IY register from directly addressed memory.

Affects IY register instead of IX. Otherwise identical to LD IX(addr).

LD I,A — LOAD INTERRUPT VECTOR OR REFRESH
LD R,A   REGISTER FROM ACCUMULATOR
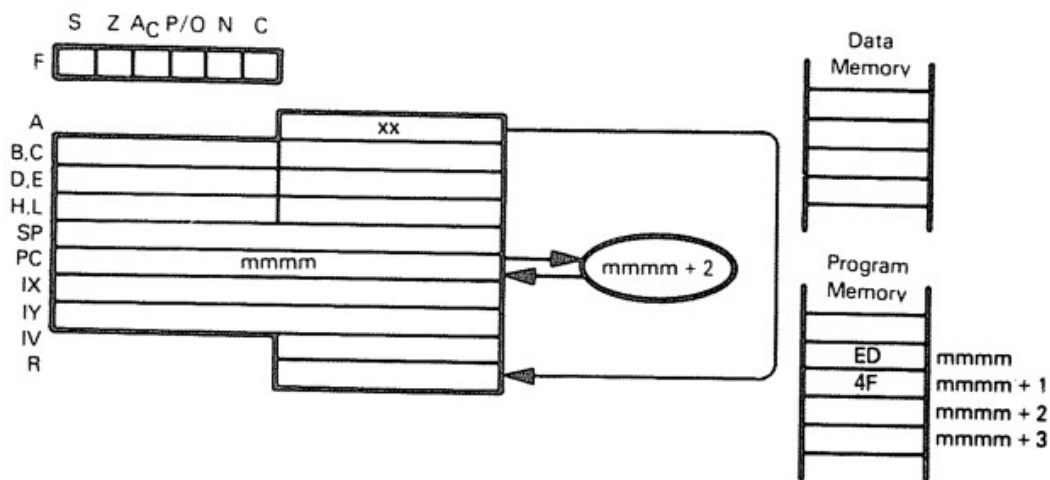
The illustration shows execution of LD R,A:

<p style="text-align:center">LD R,A
⏝
ED 4F</p>

Load Refresh register from Accumulator.

Suppose the Accumulator contains 7F$_{16}$. After the instruction

<p style="text-align:center">LD R,A</p>

has executed. the Refresh register will contain 7F$_{16}$.

<p style="text-align:center">LD I,A
⏝
ED 47</p>

Load Interrupt Vector register from Accumulator.

<p style="text-align:center">3-99</p>

LD reg,data — LOAD IMMEDIATE INTO REGISTER

LD reg,data

```
00 xxx 110 yy
```

000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A
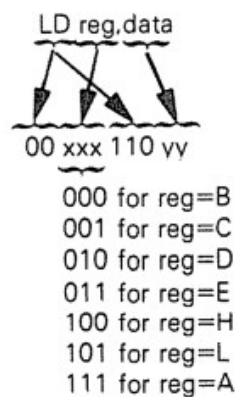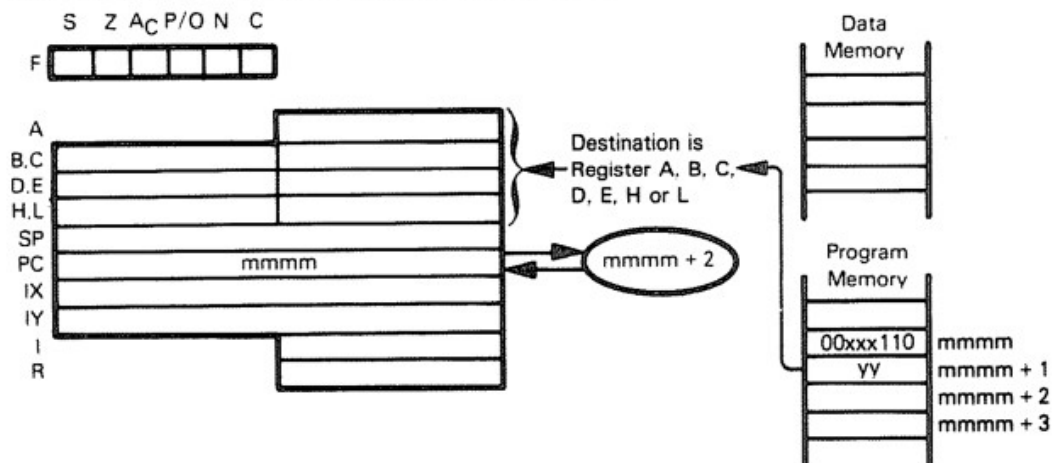
Load the contents of the second object code byte into one of the registers.

When the instruction

$$LD \ \ A,2AH$$

has executed, $2A_{16}$ is loaded into the Accumulator.

LD rp,data — LOAD 16 BITS OF DATA IMMEDIATE INTO
LD IX,data   REGISTER
LD IY,data
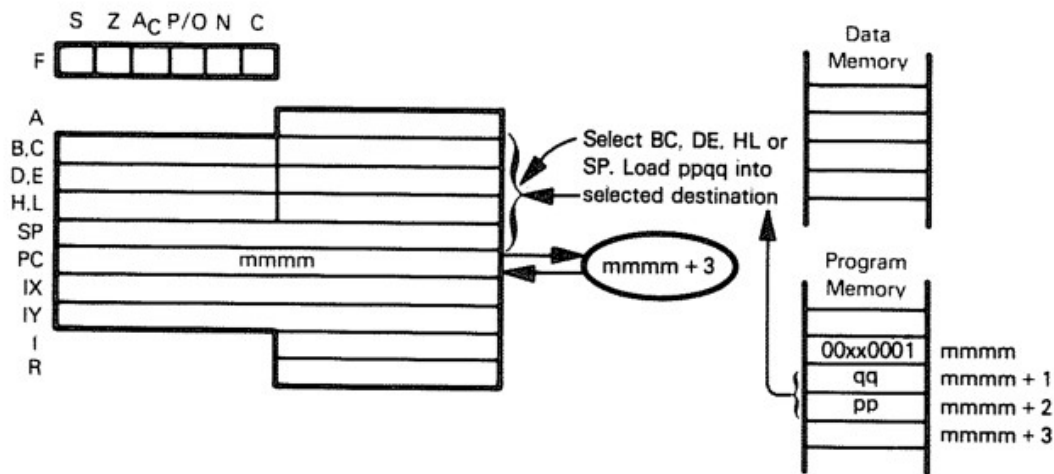
The illustration shows execution of LD rp,data:

LD rp, data

00 xx 0001 ppqq

00 for rp is register pair BC
01 for rp is register pair DE
10 for rp is register pair HL
11 for rp is Stack Pointer
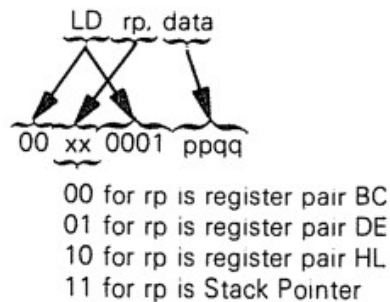
Load the contents of the second and third object code bytes into the selected register pair. After the instruction

LD SP,217AH

has executed, the Stack Pointer will contain $217A_{16}$.

LD IX, data

DD 21 ppqq

Load the contents of the second and third object code bytes into the Index register IX.

LD IY, data

FD 21 ppqq

Load the contents of the second and third object code bytes into the Index Register IY.

Notice that the LD rp,data instruction is equivalent to two LD reg,data instructions.

For example:

LD   HL,032AH

is equivalent to

LD   H,03H
LD   L,2AH

LD reg,(HL) — LOAD REGISTER FROM MEMORY
LD reg,(IX+disp)
LD reg,(IY+disp)
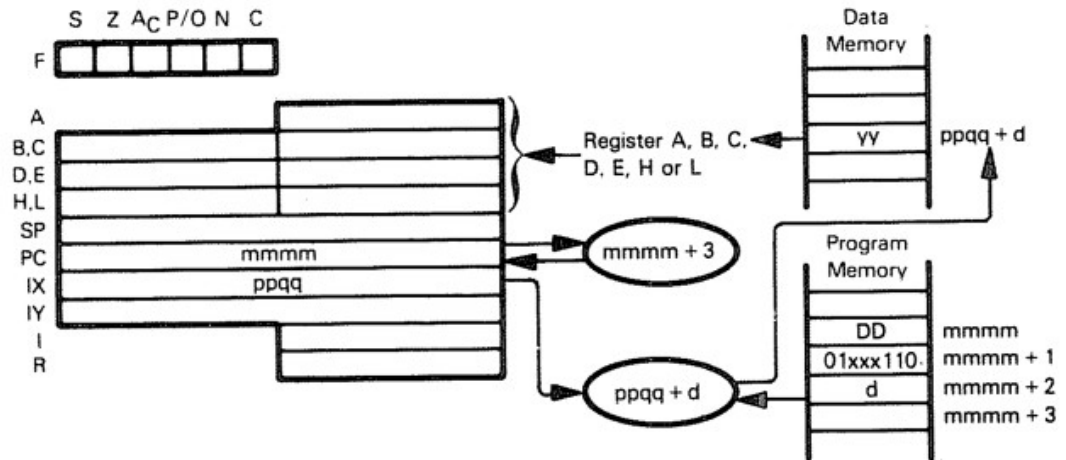
The illustration shows execution of LD reg,(IX+disp):
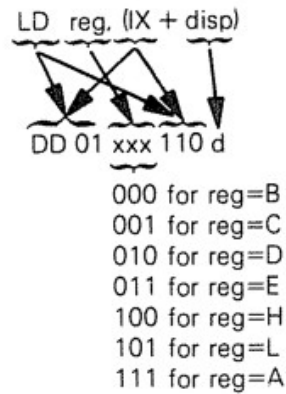
LD   reg, (IX + disp)

DD 01 xxx 110 d

```
000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A
```

Load specified register from memory location (specified by the sum of the contents of the IX register and the displacement digit d).

Suppose $ppqq=4004_{16}$ and memory location 4010_{16} contains FF_{16}. After the instruction

LD B(IX+0CH)

has executed, Register B will contain FF_{16}.

LD   reg, (IY + disp)

FD 01 xxx 110 d

same as for LD reg,(IX+disp)
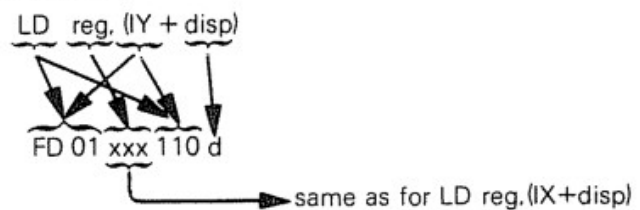
This instruction is identical to LD reg,(IX+disp), except that it uses the IY register instead of the IX register.

LD reg,(HL)
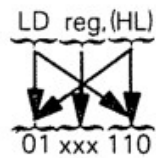
01 xxx 110 ────► same as for LD reg,(IX+disp)

Load specified register from memory location (specified by the contents of the HL register pair).

LD SP,HL — MOVE CONTENTS OF HL OR INDEX REGISTER
LD SP,IX   TO STACK POINTER
LD SP,IY

```
        S   Z  A C P/O N  C
    F [ ][ ][ ][ ][ ][ ]

    A
    B,C
    D,E
    H,L      pp           qq
    SP
    PC          mmmm              mmmm + 1
    IX
    IY
    I
    R
```

Data Memory

Program Memory

F9	mmmm
	mmmm + 1
	mmmm + 2
	mmmm + 3
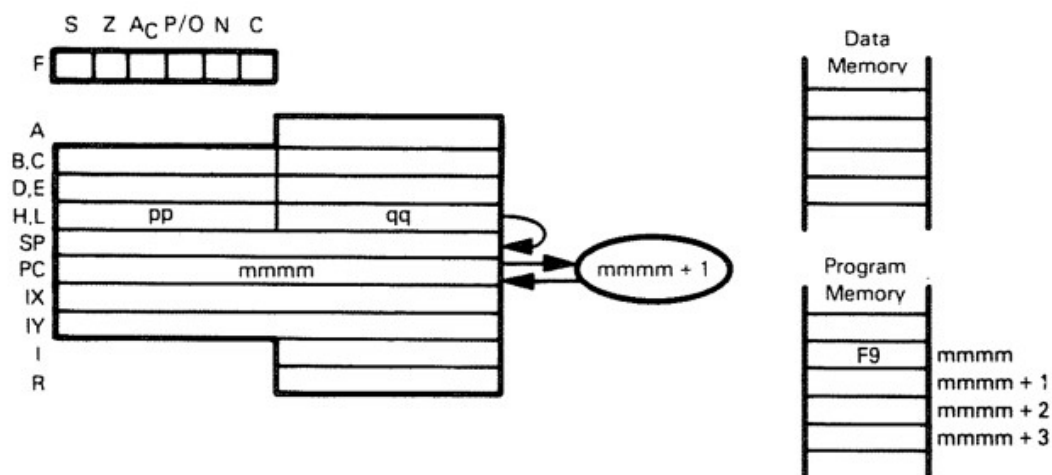
The illustration shows execution of LD SP,HL.

LD SP,HL
F9

Load contents of HL into Stack Pointer.

Suppose $pp=08_{16}$ and $qq=3F_{16}$. After the instruction

LD SP,HL

has executed. the Stack Pointer will contain $083F_{16}$.

LD SP,IX
DD F9

Load contents of Index Register IX into Stack Pointer.

LD SP,IY
FD F9

Load contents of Index Register IY into Stack Pointer.

3-103

LD (addr),A — STORE ACCUMULATOR IN MEMORY USING DIRECT ADDRESSING
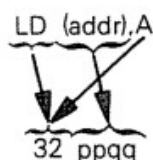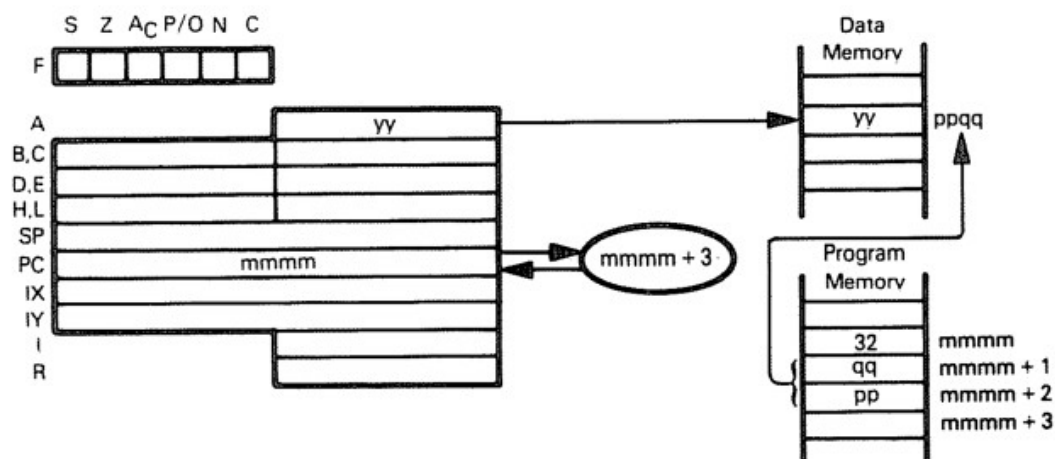
Store the Accumulator contents in the memory byte addressed directly by the second and third bytes of the LD (addr),A instruction object code.

Suppose the Accumulator contains $3A_{16}$. After the instruction

```
label       EQU       084AH
            -
            -
LD          (label),A
```

has executed, memory byte $084A_{16}$ will contain $3A_{16}$.

Remember that EQU is an assembler directive rather than an instruction: it tells the Assembler to use the 16-bit value 084AH whenever the word "label" appears.

The instruction

```
                        LD  (addr),A
```

is equivalent to the two instructions

```
                        LD  H,label
                        LD  (HL),A
```

When you are storing a single data value in memory, the LD (label),A instruction is preferred because it uses one instruction and three object program bytes to do what the LD H(label), LD (HL),A combination does in two instructions and four object program bytes. Also, the LD H(label), LD (HL),A combination uses the H and L registers, while the LD (label),A instruction does not.

LD (addr),HL — STORE REGISTER PAIR OR INDEX
LD (addr),rp   REGISTER IN MEMORY USING DIRECT
LD (addr),xy   ADDRESSING
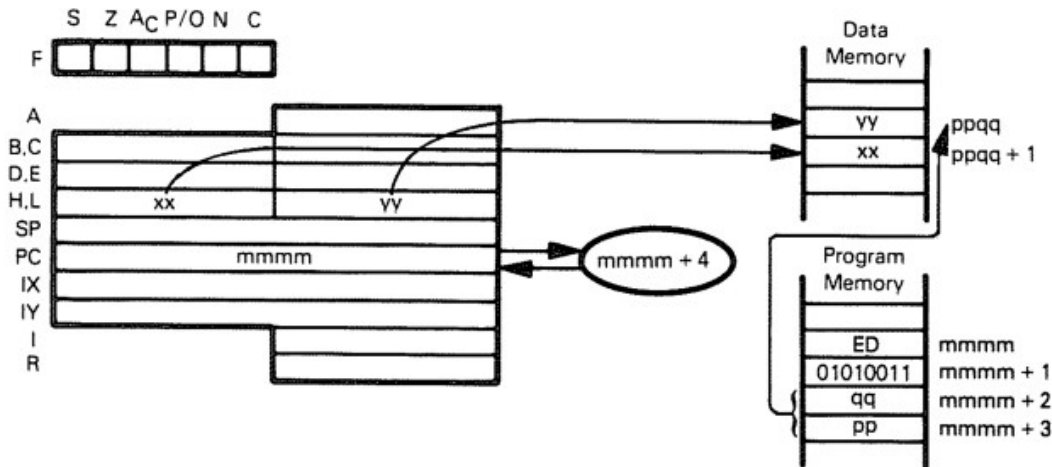
The illustration shows execution of LD (ppqq),DE:

LD (addr), rp

ED 01 xx 0011 ppqq

00 for rp is register pair BC
01 for rp is register pair DE
10 for rp is register pair HL
11 for rp is Stack Pointer
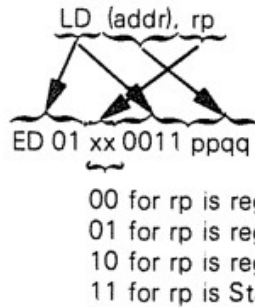
Store the contents of the specified register pair in memory. The third and fourth object code bytes give the address of the memory location where the low-order byte is to be written. The high-order byte is written into the next sequential memory location.

Suppose the BC register pair contains $3C2A_{16}$. After the instruction

      label   EQU   084AH
      -
      -
      -
      LD   (label),BC

has executed, memory byte $084A_{16}$ will contain $2A_{16}$. Memory byte $084B_{16}$ will contain $3C_{16}$.

Remember that EQU is an assembler directive rather than an instruction; it tells the Assembler to use the 16-bit value $084A_{16}$ whenever the word "label" appears.

LD (addr),HL

22 ppqq
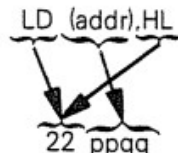
This is a three-byte version of LD (addr),rp which directly specifies HL as the source register pair.

LD (addr),IX

DD 22 ppqq
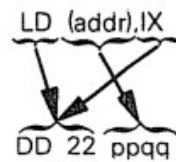
Store the contents of Index register IX in memory. The third and fourth object code bytes give the address of the memory location where the low-order byte is to be written. The high-order byte is written into the next sequential memory location.

LD (addr),IY

FD 22 ppqq
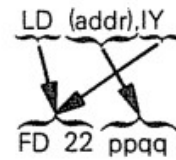
This instruction is identical to the LD (addr),IX instruction, except that it uses the IY register instead of the IX register.

LD (HL),data — LOAD IMMEDIATE INTO MEMORY
LD (IX+disp),data
LD (IY+disp),data

S   Z   A$_C$ P/O N   C

F [ ][ ][ ][ ][ ][ ]

A
B,C
D,E
H,L
SP
PC   mmmm
IX   ppqq
IY
I
R

mmmm + 4

ppqq + d

Data Memory

xx   ppqq + d

Program Memory

DD   mmmm
36   mmmm + 1
d   mmmm + 2
xx   mmmm + 3
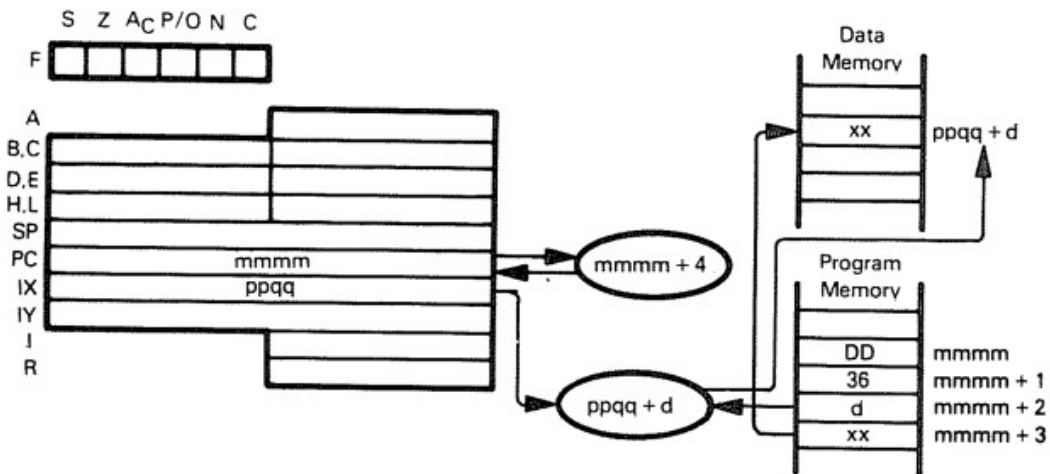
The illustration shows execution of LD (IX+d),xx:

LD (IX+disp),data

DD 36 d   xx

Load Immediate into the Memory location designated by base relative addressing.

Suppose ppqq=5400_{16}. After the instruction

LD (IX+9),FAH

has executed, memory location 5409_{16} will contain FA_{16}.

LD (IY+disp),data

FD 36 d   xx

This instruction is identical to LD (IX+disp),data, but uses the IY register instead of the IX register.

LD (HL),data

36   xx

Load Immediate into the Memory location (specified by the contents of the HL register pair).

The Load Immediate into Memory instructions are used much less than the Load Immediate into Register instructions.

LD (HL),reg — LOAD MEMORY FROM REGISTER
LD (IX+disp),reg
LD (IY+disp),reg
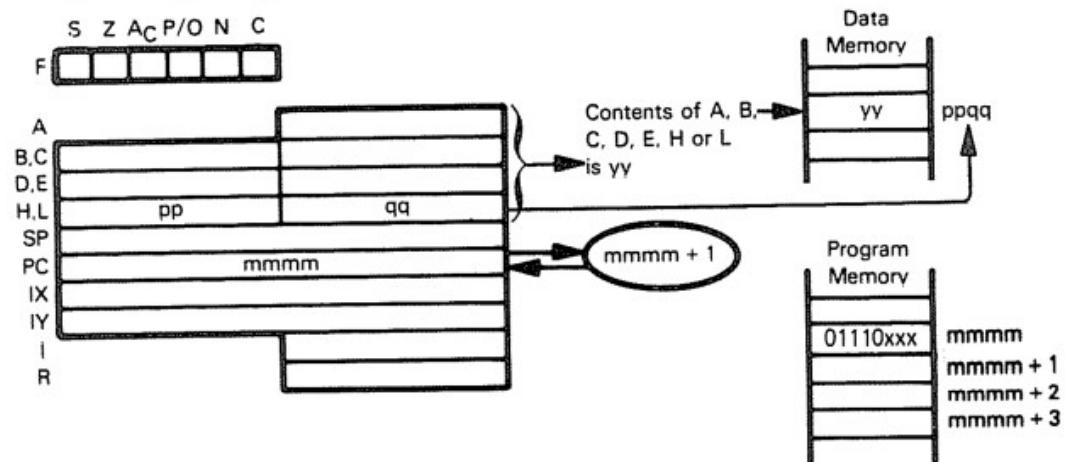
The illustration shows execution of LD (HL),reg:

LD (HL),reg

01110 xxx

000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A
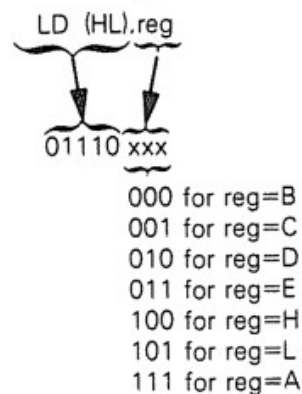
Load memory location (specified by the contents of the HL register pair) from specified register.

Suppose ppqq=4500_{16} and Register C contains $F9_{16}$. After the instruction

LD (HL),C

has executed, memory location 4500_{16} will contain $F9_{16}$.
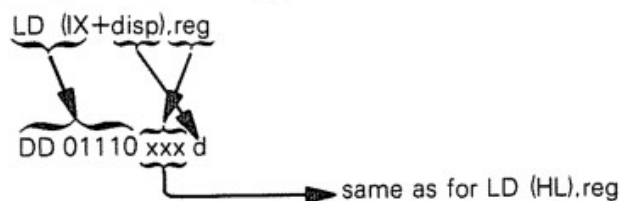
LD (IX+disp),reg

DD 01110 xxx d

same as for LD (HL),reg

Load memory location (specified by the sum of the contents of the IX register and the

displacement value d) from specified register.

LD (IY+disp),reg

FD 01110 xxx d

same as for LD (HL),reg
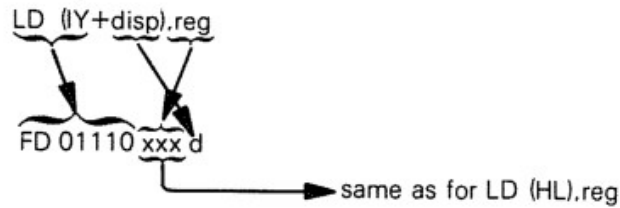
This instruction is identical to LD (IX+disp),reg, except that it uses the IY register instead of the IX register.

LD (rp),A — LOAD ACCUMULATOR INTO THE MEMORY LOCATION ADDRESSED BY REGISTER PAIR

LD (rp),A

000 x 0010

0 if register pair=BC
1 if register pair=DE
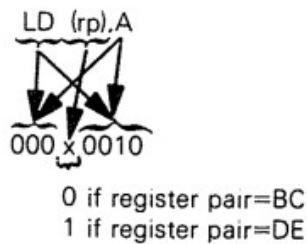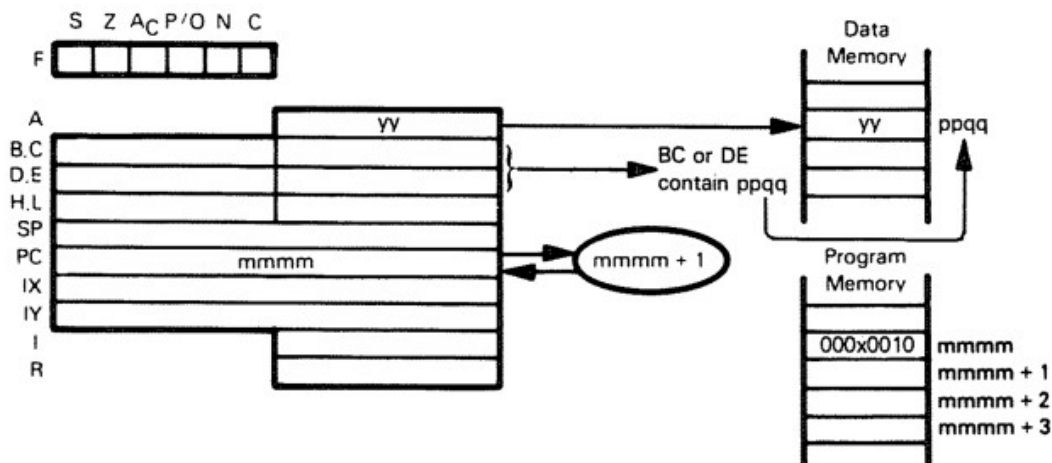
Store the Accumulator in the memory byte addressed by the BC or DE register pair.

Suppose the BC register pair contains $084A_{16}$ and the Accumulator contains $3A_{16}$. After the instruction

LD (BC),A

has executed, memory byte $084A_{16}$ will contain $3A_{16}$.

The LD (rp),A and LD rp,data will normally be used together, since the LD rp,data instruction loads a 16-bit address into the BC or DE registers as follows:

LD BC,084AH
LD (BC),A

3-109

LDD — TRANSFER DATA BETWEEN MEMORY LOCATIONS, DECREMENT DESTINATION AND SOURCE ADDRESSES

Set if BC-1 $\neq$ 0, reset otherwise
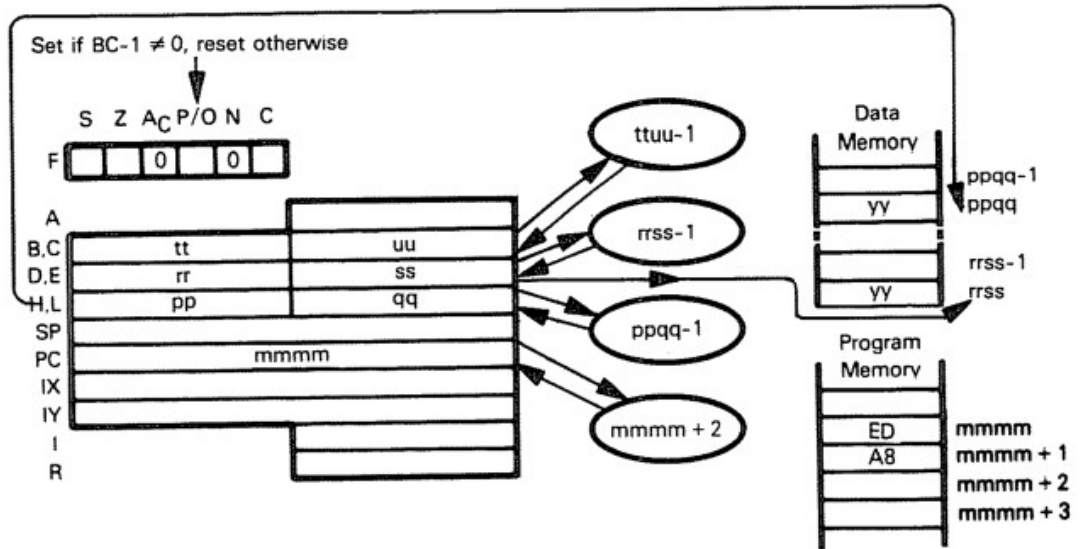
$$\underbrace{LDD}_{ED\ A8}$$

Transfer a byte of data from memory location addressed by the HL register pair to memory location addressed by the DE register pair. Decrement contents of register pairs BC, DE, and HL.

Suppose register pair BC contains $004F_{16}$, DE contains 4545_{16}, HL contains 2012_{16}, and memory location 2012_{16} contains 18_{16}. After the instruction

$$LDD$$

has executed, memory location 4545_{16} will contain 18_{16}, register pair BC will contain $004E_{16}$, DE will contain 4544_{16}, and HL will contain 2011_{16}.

LDDR — TRANSFER DATA BETWEEN MEMORY LOCATIONS UNTIL BYTE COUNTER IS ZERO. DECREMENT DESTINATION AND SOURCE ADDRESSES

$$\underbrace{\text{LDDR}}_{\text{ED B8}}$$

This instruction is identical to LDD, except that it is repeated until the BC register pair contains zero. After each data transfer, interrupts will be recognized and two refresh cycles will be executed.

Suppose we have the following contents in memory and register pairs:

Register/Contents	Location/Contents
HL 2012_{16}	2012_{16} 18_{16}
DE 4545_{16}	2011_{16} AA_{16}
BC 0003_{16}	2010_{16} 25_{16}

After execution of

LDDR

register pairs and memory locations will have the following contents:

Register/Contents	Location/Contents	Location/Contents
HL 2009_{16}	2012_{16} 18_{16}	4545_{16} 18_{16}
DE 4542_{16}	2011_{16} AA_{16}	4544_{16} AA_{16}
BC 0000_{16}	2010_{16} 25_{16}	4543_{16} 25_{16}

This instruction is extremely useful for transferring blocks of data from one area of memory to another.

LDI — TRANSFER DATA BETWEEN MEMORY LOCATIONS, INCREMENT DESTINATION AND SOURCE ADDRESSES

Set if BC-1 ≠ 0, reset otherwise

S Z A$_C$ P/O N C

F [| | 0 | | 0 |]

A
B,C   tt   uu
D,E   rr   ss
H,L   pp   qq
SP
PC   mmmm
IX
IY
I
R

ttuu-1

rrss + 1

ppqq + 1

mmmm + 2

Data Memory

yy	ppqq
	ppqq + 1
yy	rrss
	rrss + 1

Program Memory

ED	mmmm
A0	mmmm + 1
	mmmm + 2
	mmmm + 3
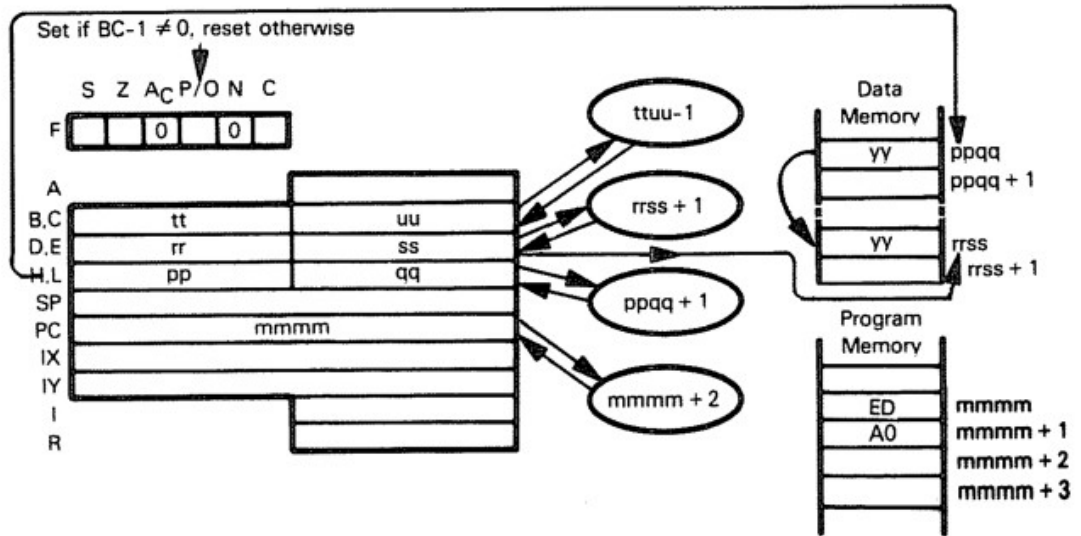
LDI
$\underbrace{\qquad}$
ED A0

Transfer a byte of data from memory location addressed by the HL register pair to memory location addressed by the DE register pair. Increment contents of register pairs HL and DE. Decrement contents of the BC register pair.

Suppose register pair BC contains $004F_{16}$, DE contains 4545_{16}, HL contains 2012_{16}, and memory location 2012_{16} contains 18_{16}. After the instruction

LDI

has executed, memory location 4545_{16} will contain 18_{16}, register pair BC will contain $004E_{16}$, DE will contain 4546_{16}, and HL will contain 2013_{16}.

LDIR — TRANSFER DATA BETWEEN MEMORY LOCATIONS UNTIL BYTE COUNTER IS ZERO. INCREMENT DESTINATION AND SOURCE ADDRESSES

$$\underbrace{\text{LDIR}}_{\text{ED B0}}$$

This instruction is identical to LDI, except that it is repeated until the BC register pair contains zero. After each data transfer, interrupts will be recognized and two refresh cycles will be executed.

Suppose we have the following contents in memory and register pairs:

Register/Contents	Location/Contents
HL 2012_{16}	2012_{16} 18_{16}
DE 4545_{16}	2013_{16} CD_{16}
BC 0003_{16}	2014_{16} $F0_{16}$

After execution of

LDIR

register pairs and memory will have the following contents:

Register/Contents	Location/Contents	Location/Contents
HL 2015_{16}	2012_{16} 18_{16}	4545_{16} 18_{16}
DE 4548_{16}	2013_{16} CD_{16}	4546_{16} CD_{16}
BC 0000_{16}	2014_{16} $F0_{16}$	4547_{16} $F0_{16}$

This instruction is extremely useful for transferring blocks of data from one area of memory to another.

NEG — NEGATE CONTENTS OF ACCUMULATOR
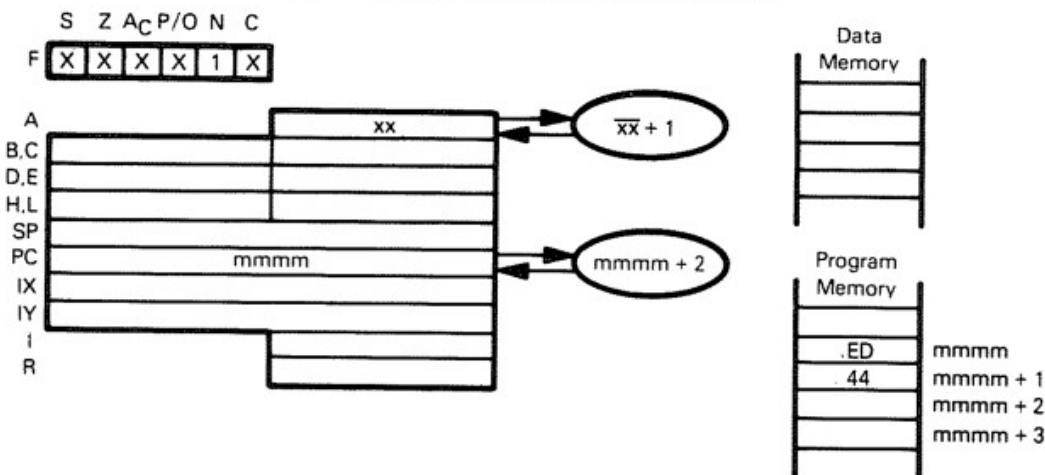
Negate contents of Accumulator. This is the same as subtracting contents of the Accumulator from zero. The result is the two's complement. 80H will be left unchanged.

Suppose $xx=5A_{16}$. After the instruction

NEG

has executed, the Accumulator will contain $A6_{16}$.

$$5A = 0101 \quad 1010$$
$$\text{Two's complement} = 1010 \quad 0110$$

NOP — NO OPERATION

S Z A_C P/O N C

NOP
⌣
00
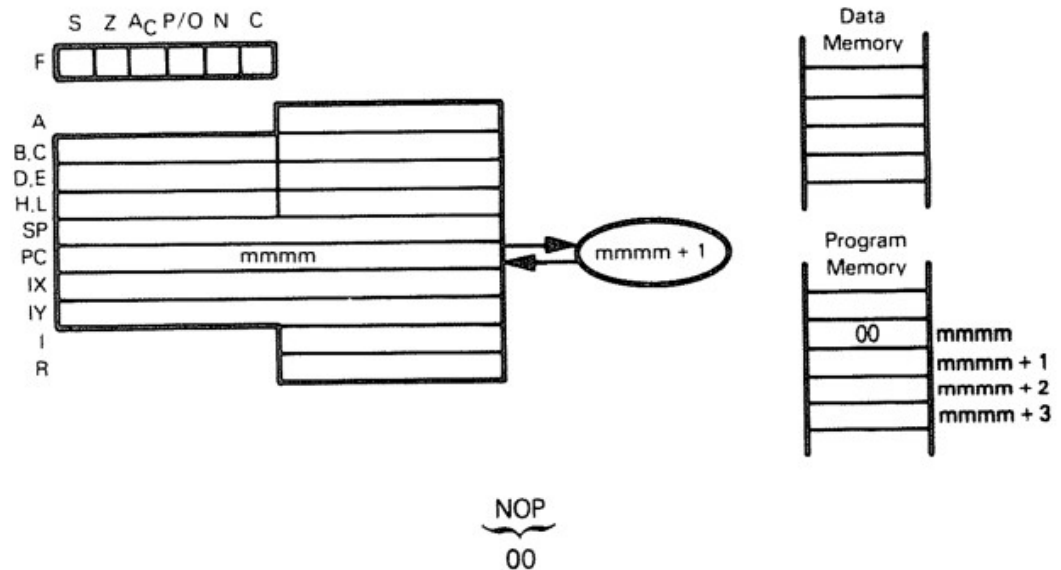
This is a one-byte instruction which performs no operation, except that the Program Counter is incremented and memory refresh continues. This instruction is present for several reasons:

1) A program error that fetches an object code from non-existent memory will fetch 00. It is a good idea to ensure that the most common program error will do nothing.

2) The NOP instruction allows you to give a label to an object program byte:
   HERE   NOP

3) To fine-tune delay times. Each NOP instruction adds four clock cycles to a delay.

NOP is not a very useful or frequently used instruction.

OR data — OR IMMEDIATE WITH ACCUMULATOR
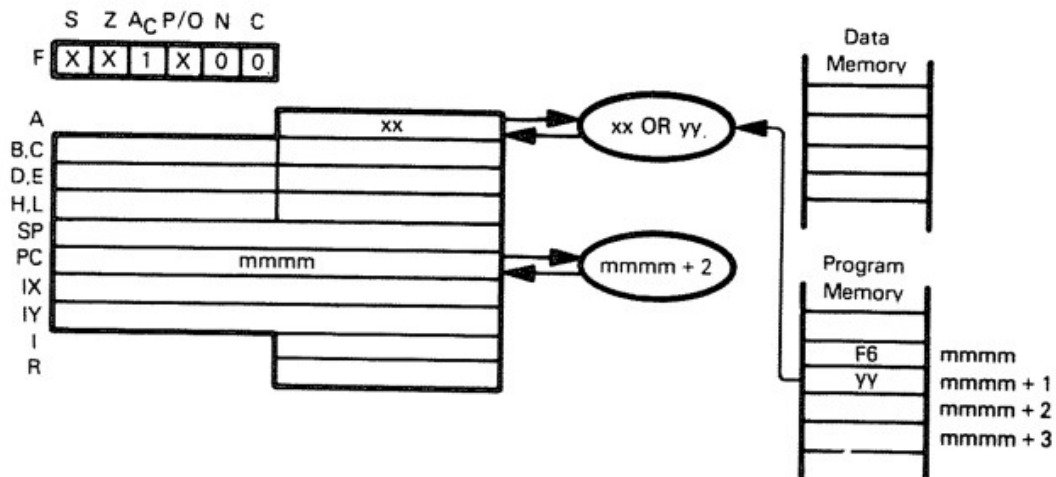
$$
\underbrace{\text{OR}}_{\text{F6}} \quad \underbrace{\text{data}}_{\text{yy}}
$$

OR the Accumulator with the contents of the second instruction object code byte.

Suppose $xx=3A_{16}$. After the instruction

OR 7CH

has executed, the Accumulator will contain $7E_{16}$.

```
3A  =  0011   1010
7C  =  0111   1100
       0111   1110
```

0 sets S to 0 ◄—

— Six 1 bits, set P/O to 1

— Non-zero result, set Z to 0

This is a routine logical instruction; it is often used to turn bits "on". For example, the instruction

OR 80H

will unconditionally set the high-order Accumulator bit to 1.

OR reg — OR REGISTER WITH ACCUMULATOR
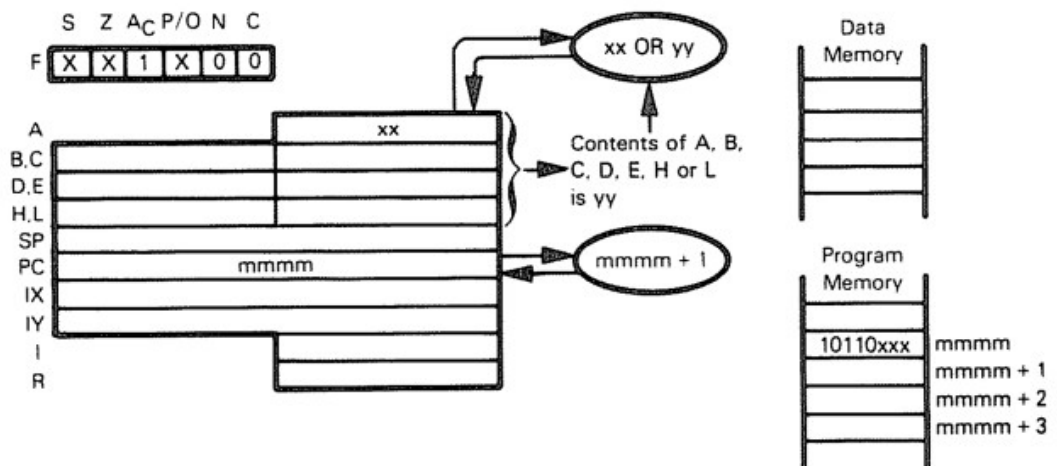
$$\underbrace{\text{OR}}_{10110} \quad \underbrace{\text{reg}}_{\text{xxx}}$$

000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A

Logically OR the contents of the Accumulator with the contents of Register A, B, C, D, E, H or L. Store the result in the Accumulator.

Suppose $xx=E3_{16}$ and Register E contains $A8_{16}$. After the instruction

OR E

has executed, the Accumulator will contain EB_{16}.

```
E3 = 1110  0011
A8 = 1010  1000
     1110  1011
```

1 sets S to 1 ◄——

Six 1 bits, set P/O to 1

Non-zero result, set Z to 0

OR (HL) — OR MEMORY WITH ACCUMULATOR
OR (IX+disp)
OR (IY+disp)
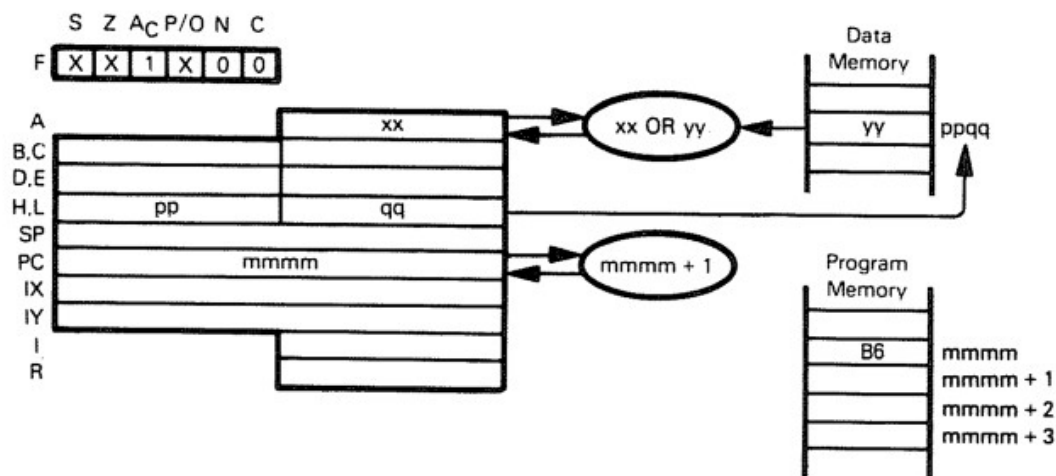
The illustration shows execution of OR (HL):

$$\underbrace{\text{OR (HL)}}_{\text{B6}}$$

OR contents of memory location (specified by the contents of the HL register pair) with the Accumulator.

Suppose $xx=E3_{16}$, $ppqq=4000_{16}$, and memory location 4000_{16} contains $A8_{16}$. After the instruction

OR (HL)

has executed. the Accumulator will contain EB_{16}.

```
E3 = 1110   0011
A8 = 1010   1000
     1110   1011
```

1 sets S to 1 ←

Six 1 bits, set P/O to 1

Non-zero result. set Z to 0

$$\underbrace{\text{OR (IX+disp)}}_{\text{DD B6 d}}$$

OR contents of memory location (specified by the sum of the contents of the IX register and the displacement value d) with the Accumulator.

$$\underbrace{\text{OR (IY+disp)}}_{\text{FD B6 d}}$$

This instruction is identical to OR (IX+disp), except that it uses the IY register instead of the IX register.

OUT (C),reg — OUTPUT FROM REGISTER

OUT (C),reg

ED 01 xxx 001
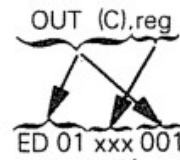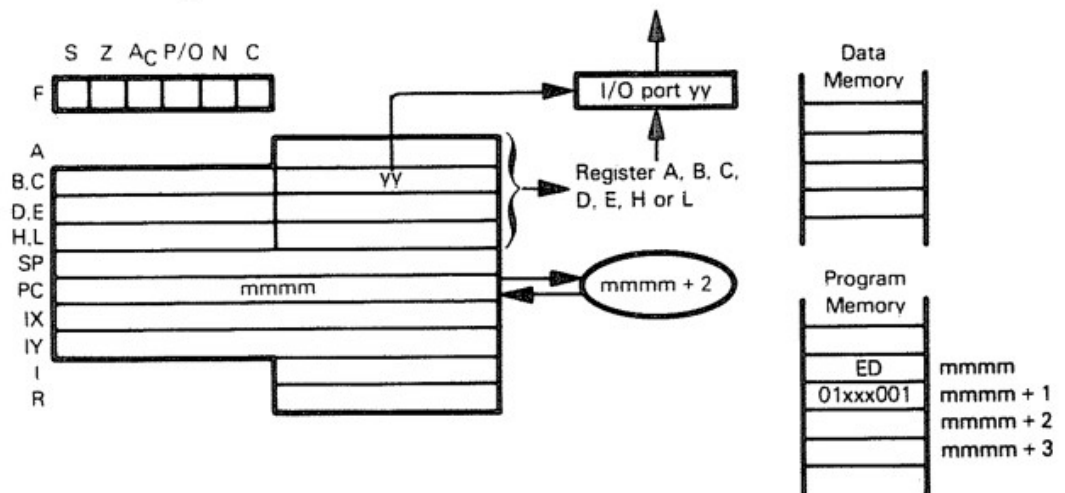
000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A

Suppose yy=$1F_{16}$ and the contents of H are AA_{16}. After the execution of

OUT (C),H

AA_{16} will be in the buffer of I/O port $1F_{16}$.

OUTD — OUTPUT FROM MEMORY. DECREMENT ADDRESS
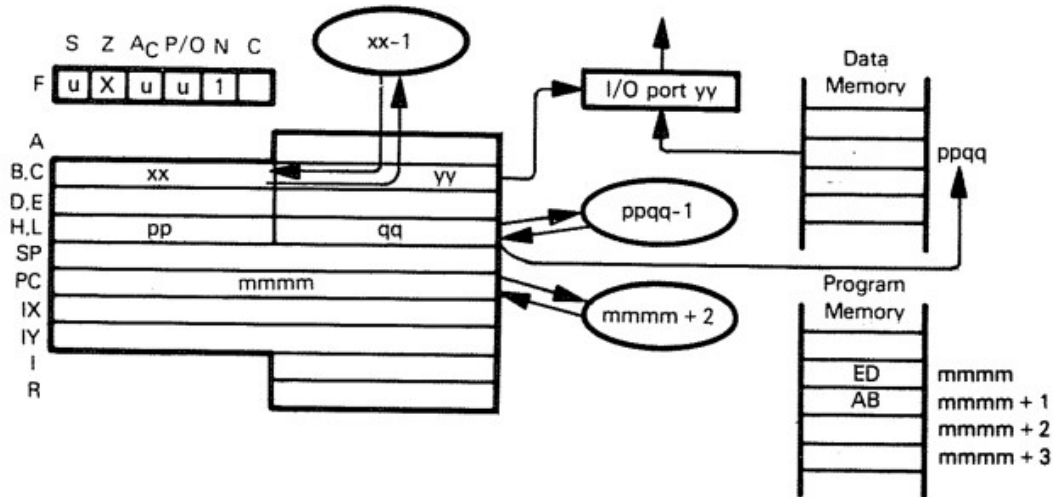
$$OUTD \atop ED \ AB$$

Output from memory location specified by HL to I/O port addressed by Register C. Registers B and HL are decremented.

Suppose $xx=0A_{16}$, $yy=FF_{16}$, $ppqq=5000_{16}$, and memory location 5000_{16} contains 77_{16}. After the instruction

OUTD

has executed, 77_{16} will be held in the buffer of I/O port FF_{16}. The B register will contain 09_{16}, and the HL register pair $4FFF_{16}$.

OTDR — OUTPUT FROM MEMORY. DECREMENT ADDRESS, CONTINUE UNTIL REGISTER B=0

$$OTDR \atop ED \ BB$$

OTDR is identical to OUTD, but is repeated until Register B contains 0.

Suppose Register B contains 03_{16}, Register C contains FF_{16}, and HL contains 5000_{16}. Memory locations $4FFE_{16}$ through 5000_{16} contain:

Location/Contents	
$4FFE_{16}$	CA_{16}
$4FFF_{16}$	$1B_{16}$
5000_{16}	$F1_{16}$

After execution of

OTDR

register pair HL will contain $4FFD_{16}$. Register B will contain zero, and the sequence $F1_{16}$, $1B_{16}$, CA_{16} will have been written to I/O port FF_{16}.

This instruction is very useful for transferring blocks of data from memory to output devices.

OUTI — OUTPUT FROM MEMORY. INCREMENT ADDRESS
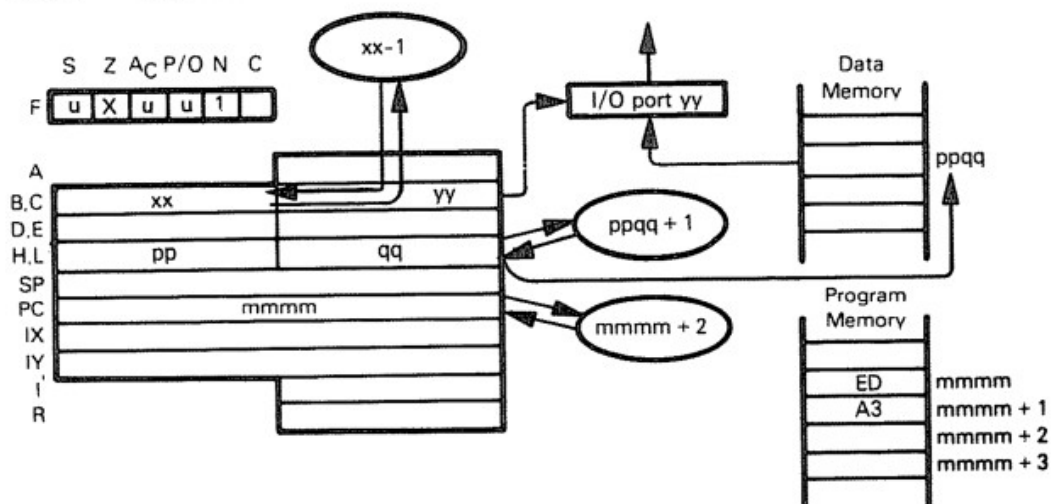
$$\text{OUTI}$$
$$\overbrace{}$$
$$\text{ED} \quad \text{A3}$$

Output from memory location specified by HL to I/O port addressed by Register C. Register B is decremented and the HL register pair is incremented.

Suppose $xx=0A_{16}$, $yy=FF_{16}$, $ppqq=5000_{16}$, and memory location 5000_{16} contains 77_{16}. After the instruction

$$\text{OUTI}$$

has executed, 77_{16} will be held in the buffer of I/O port FF_{16}. The B register will contain 09_{16} and the HL register pair will contain 5001_{16}.

OTIR — OUTPUT FROM MEMORY. INCREMENT ADDRESS, CONTINUE UNTIL REGISTER B=0

$$\text{OTIR}$$
$$\overbrace{}$$
$$\text{ED} \quad \text{B3}$$

OTIR is identical to OUTI. except that it is repeated until Register B contains 0.

Suppose Register B contains 04_{16}. Register C contains FF_{16}, and HL contains 5000_{16}. Memory locations 5000_{16} through 5003_{16} contain:

Location/Contents	
5000_{16}	CA_{16}
5001_{16}	$1B_{16}$
5002_{16}	$B1_{16}$
5003_{16}	AD_{16}

After execution of

$$\text{OTIR}$$

register pair HL will contain 5004_{16}. Register B will contain zero and the sequence CA_{16}, $1B_{16}$, $B1_{16}$ and AD_{16} will have been written to I/O port FF_{16}.

This instruction is very useful for transferring blocks of data from memory to an output device.

OUT (port),A — OUTPUT FROM ACCUMULATOR
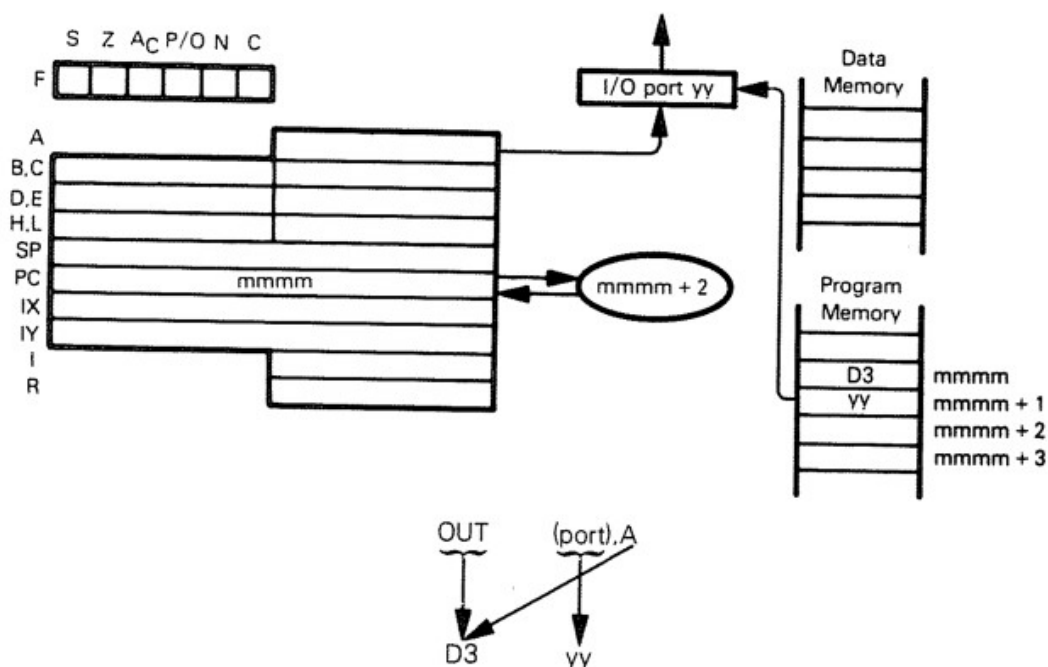
Output the contents of the Accumulator to the I/O port identified by the second OUT instruction object code byte.

Suppose 36_{16} is held in the Accumulator. After the instruction

OUT (1AH),A

has executed, 36_{16} will be in the buffer of I/O port $1A_{16}$.

The OUT instruction does not affect any statuses. Use of the OUT instruction is very hardware-dependent. Valid I/O port addresses are determined by the way in which I/O logic has been implemented. It is also possible to design a microcomputer system that accesses external logic using memory reference instructions with specific memory addresses. OUT instructions are frequently used in special ways to control microcomputer logic external to the CPU.

POP rp — READ FROM THE TOP OF THE STACK
POP IX
POP IY

The illustration shows execution of POP BC.

POP the two top stack bytes into the designated register pair.

Suppose $qq=01_{16}$ and $pp=2A_{16}$. Execution of

POP HL

loads 01_{16} into the L register and $2A_{16}$ into the H register. Execution of the instruction

POP AF

loads 01 into the status flags and $2A_{16}$ into the Accumulator. Thus, the Carry status will be set to 1 and other statuses will be cleared.

POP IX

DD E1

POP the two top stack bytes into the IX register.

POP IY

FD E1

POP the two top stack bytes into the IY register.

The POP instruction is most frequently used to restore register and status contents which have been saved on the stack; for example, while servicing an interrupt.

PUSH rp — WRITE TO THE TOP OF THE STACK
PUSH IX
PUSH IY
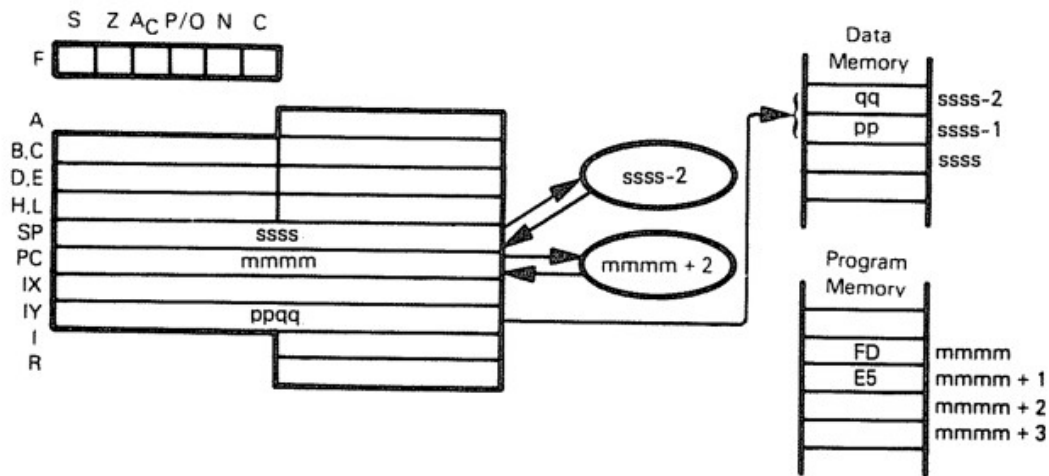
The illustration shows execution of PUSH IY:

$$\underbrace{\text{PUSH IY}}_{\text{FD E5}}$$

PUSH the contents of the IY register onto the top of the stack.

Suppose the IY register contains $45FF_{16}$. Execution of the instruction

PUSH IY

loads 45_{16}, then FF_{16} onto the top of the stack.

$$\underbrace{\text{PUSH IX}}_{\text{DD E5}}$$

PUSH the contents of the IX register onto the top of the stack.

PUSH rp

11 xx 0101

00 for rp is register pair BC
01 for rp is register pair DE
10 for rp is register pair HL
11 for rp is register pair A and F

PUSH contents of designated register pair onto the top of the stack.

Execution of the instruction

PUSH AF

loads the Accumulator and then the status flags onto the top of the stack.

The PUSH instruction is most frequently used to save register and status contents; for example, before servicing an interrupt.

RES b,reg — RESET INDICATED REGISTER BIT

S Z A$_C$ P/O N   C

F [ ][ ][ ][ ][ ][ ]

A
B.C
D.E
H.L
SP
PC   mmmm
IX
IY
I
R

yyyyyyyy

0

mmmm + 2

Data
Memory

Program
Memory

CB	mmmm
10bbbxxx	mmmm + 1
	mmmm + 2
	mmmm + 3
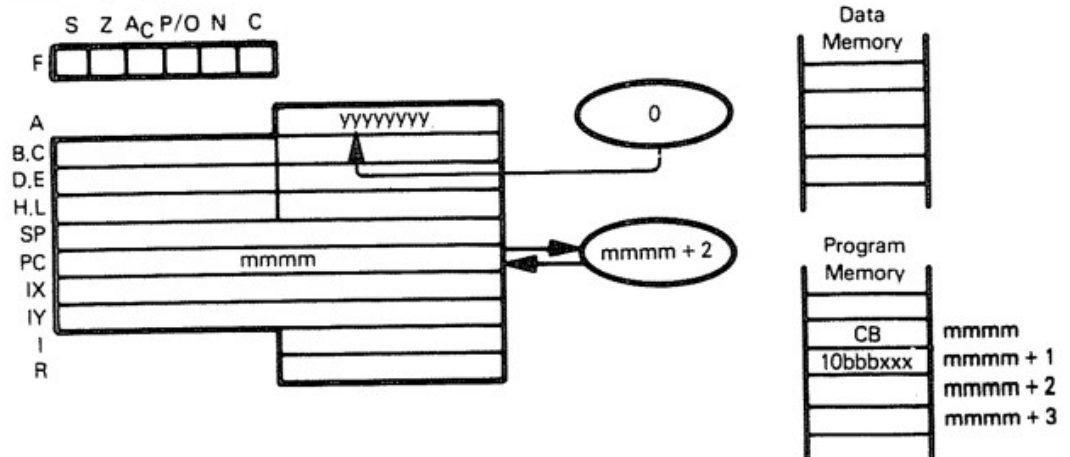
```
          RES      b.reg

     CB   10    bbb  xxx
```

Bit	bbb	xxx	Register
0	000	000	B
1	001	001	C
2	010	010	D
3	011	011	E
4	100	100	H
5	101	101	L
6	110	111	A
7	111		

Reset indicated bit within specified register.

After the instruction

RES 6.H

has executed. bit 6 in Register H will be reset. (Bit 0 is the least significant bit.)

RES b,(HL) — RESET BIT b OF INDICATED MEMORY POSITION
RES b,(IX+disp)
RES b,(IY+disp)
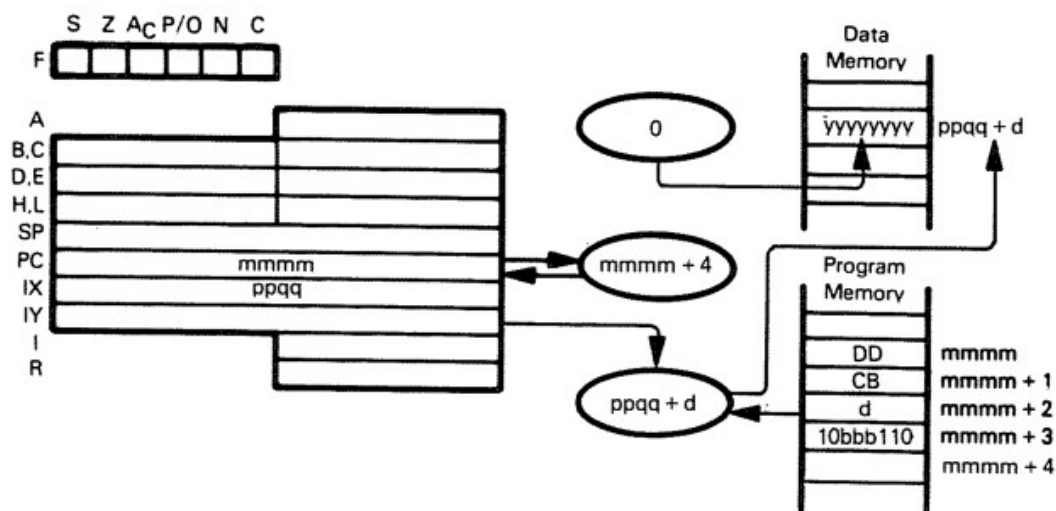
The illustration shows execution of SET b,(IX+disp). Bit 0 is execution of SET b,(IX+disp). Bit 0 is the least significant bit.

bbb	Bit Reset
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7
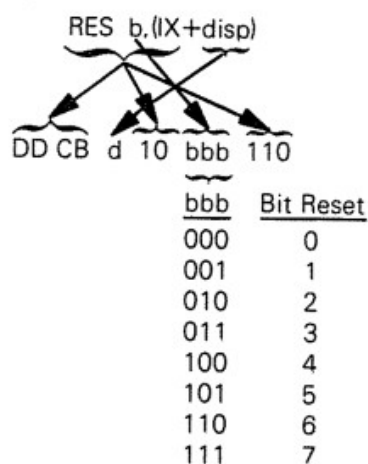
Reset indicated bit within memory location indicated by the sum of Index Register IX and d.

Suppose IX contains 4110_{16}. After the instruction

RES 0,(IX+7)

has executed, bit 0 in memory location 4117_{16} will be 0.

bbb is the same as in RES b,(IX+disp)
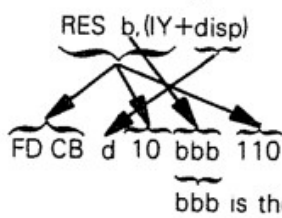
This instruction is identical to RES b,(IX+disp), except that it uses the IY register instead

of the IX register.

RES b.(HL)

CB 10 bbb 110
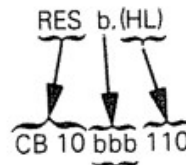
bbb is the same as in RES b.(IX+disp)

Reset indicated bit within memory location indicated by HL.

Suppose HL contains 4444_{16}. After execution of

RES 7.(HL)

bit 7 in memory location 4444_{16} will be 0.

RET — RETURN FROM SUBROUTINE
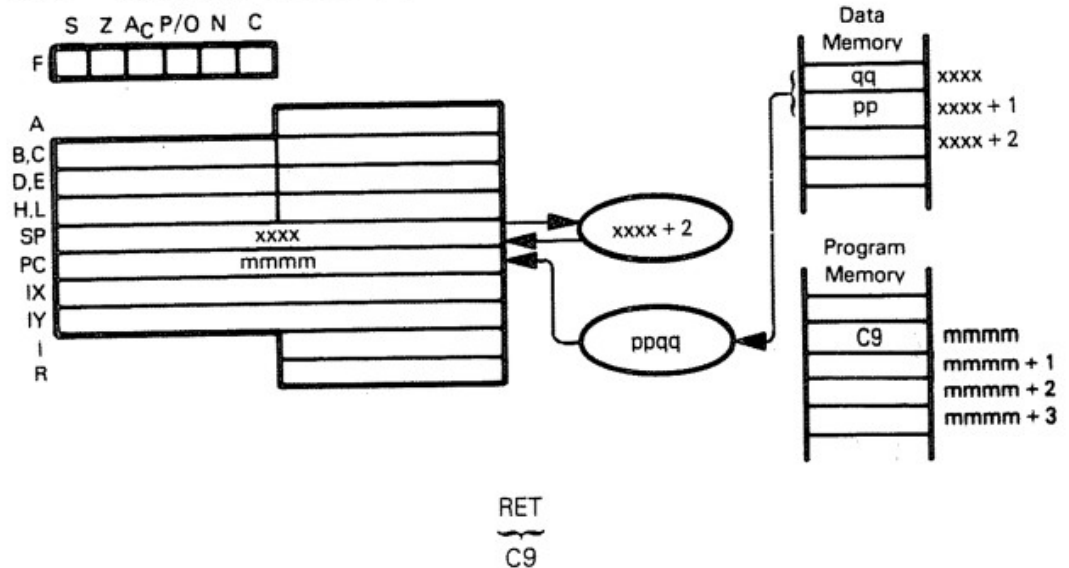
RET

C9

Move the contents of the top two stack bytes to the Program Counter; these two bytes provide the address of the next instruction to be executed. Previous Program Counter contents are lost. Increment the Stack Pointer by 2, to address the new top of stack.

Every subroutine must contain at least one Return (or conditional Return) instruction; this is the last instruction executed within the subroutine, and causes execution to return to the calling program.

RET cond — RETURN FROM SUBROUTINE IF CONDITION IS SATISFIED
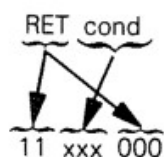
RET cond

11 xxx 000

		Condition	Relevant Flag
000	NZ	Non-Zero	Z
001	Z	Zero	Z
010	NC	Non-Carry	C
011	C	Carry	C
100	PO	Parity Odd	P/O
101	PE	Parity Even	P/O
110	P	Sign Positive	S
111	M	Sign Negative	S

This instruction is identical to the RET instruction, except that the return is not executed unless the condition is satisfied; otherwise, the instruction sequentially following the RET cond instruction will be executed.

Consider the instruction sequence:

```
        CALL    SUBR
        AND     7CH
        -
        -
SUBR                      :First subroutine instruction
        -
        -                 condition satisfied
        RET     cond

        condition not
        satisfied

        OR      80H
```
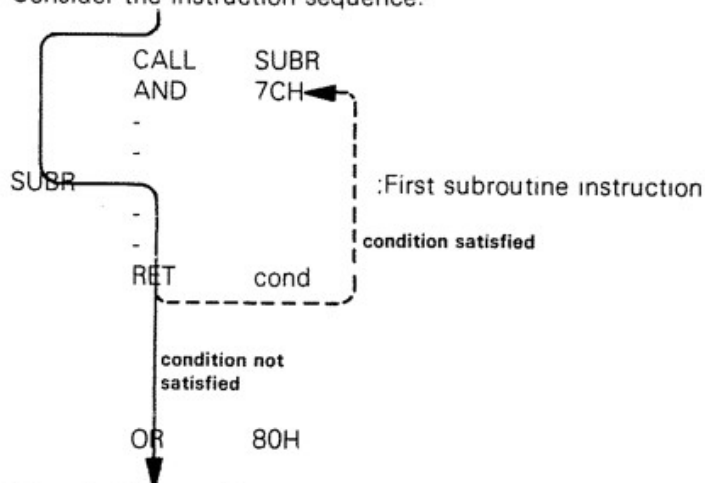
After the RET cond is executed, if the condition is satisfied then execution returns to the AND instruction which follows the CALL. If the condition is not satisfied, the OR instruction, being the next sequential instruction, is executed.

RETI — RETURN FROM INTERRUPT
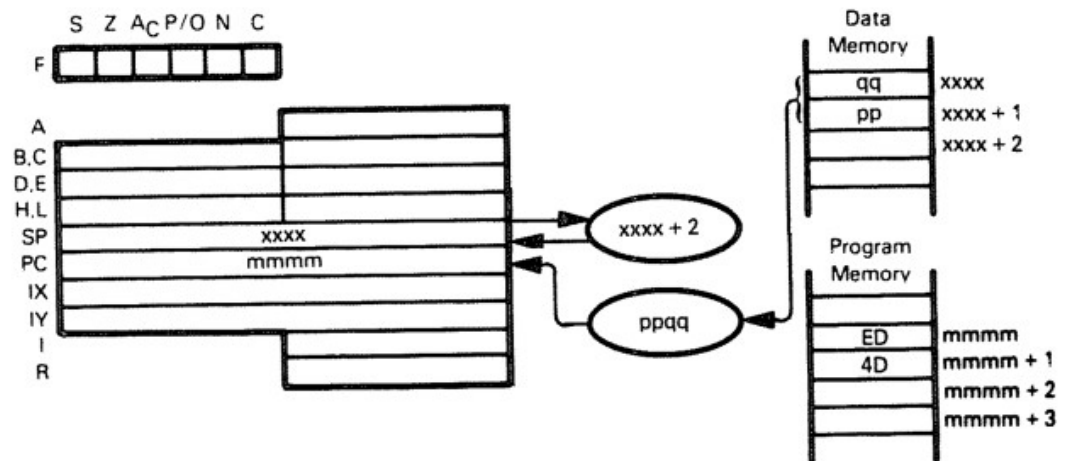
$$\underbrace{\text{RETI}}_{\text{ED 4D}}$$

Move the contents of the top two stack bytes to the Program Counter; these two bytes provide the address of the next instruction to be executed. Previous Program Counter contents are lost. Increment the Stack Pointer by 2, and address the new top of stack.

This instruction is used at the end of an interrupt service routine, and, in addition to returning control to the interrupted program, it is used to signal an I/O device that the interrupt routine has been completed. The I/O device must provide the logic necessary to sense the instruction operation code: refer to An Introduction to Microcomputers: Volume 2 for a description of how the RETI instruction operates with the Z80 family of devices.

RETN — RETURN FROM NON-MASKABLE INTERRUPT
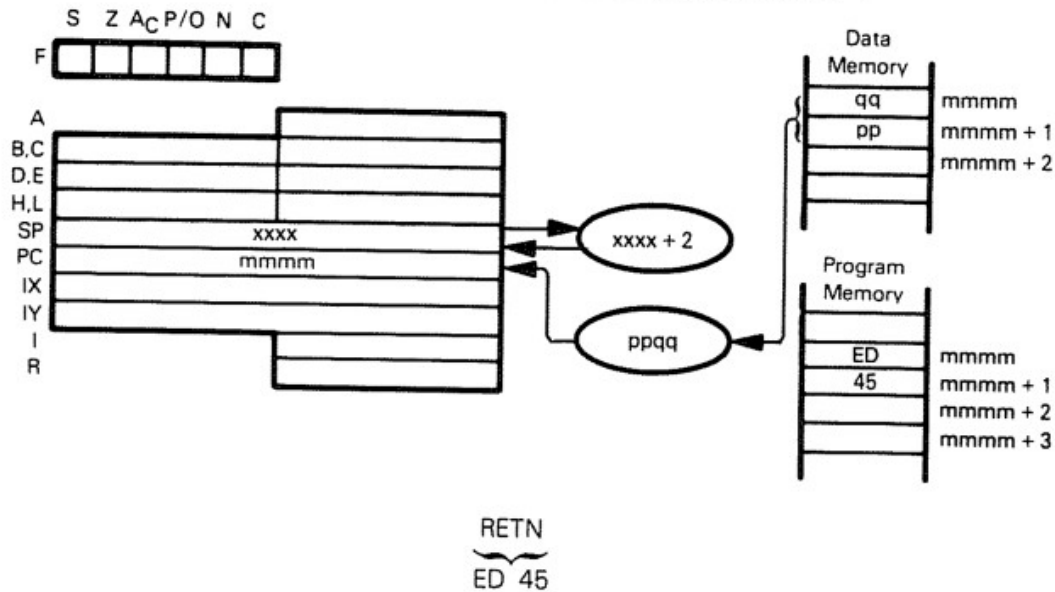
RETN
‿‿
ED 45

Move the contents of the top two stack bytes to the Program Counter: these two bytes provide the address of the next instruction to be executed. Previous Program Counter contents are lost. Increment the Stack Pointer by 2 to address the new top of stack. Restore the interrupt enable logic to the state it had prior to the occurrence of the non-maskable interrupt.

This instruction is used at the end of a service routine for a non-maskable interrupt, and causes execution to return to the program that was interrupted.

RL reg — ROTATE CONTENTS OF REGISTER LEFT THROUGH CARRY
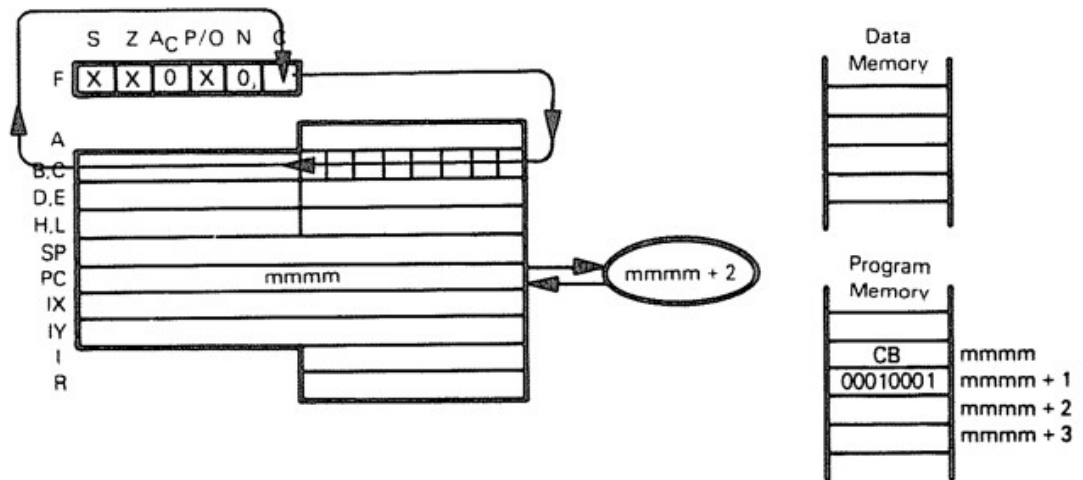
The illustration shows execution of RL C:

```
        RL  reg
       ⌣⌣⌣  ⌣⌣⌣
        │     │
        ▼     ▼
      ⌣⌣⌣⌣⌣  ⌣⌣⌣
      CB 00010 xxx
              ⌣⌣⌣
              000 for reg=B
              001 for reg=C
              010 for reg=D
              011 for reg=E
              100 for reg=H
              101 for reg=L
              111 for reg=A
```
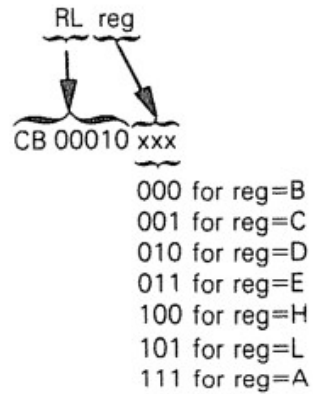
Rotate contents of specified register left one bit through Carry.

Suppose D contains $A9_{16}$ and Carry=0. After the instruction

RL D

has executed, D will contain 52_{16} and Carry will be 1:
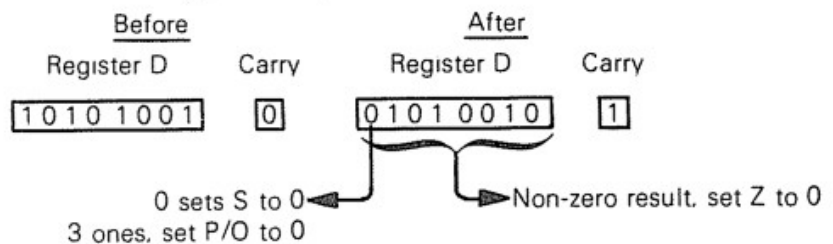
RL (HL) — ROTATE CONTENTS OF MEMORY LOCATION
RL (IX+disp) LEFT THROUGH CARRY
RL (IY+disp)
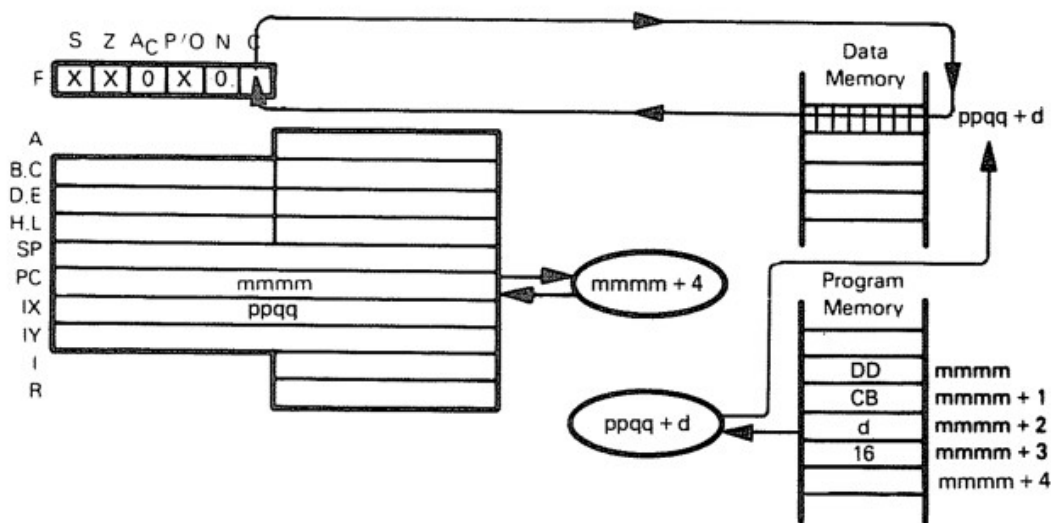
The illustration shows execution of RL (IX+disp):
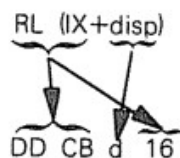
RL (IX+disp)

DD CB d 16

Rotate contents of memory location (specified by the sum of the contents of Index Register IX and displacement integer d) left one bit through Carry.

Suppose the IX register contains 4000_{16}, memory location 4007_{16} contains $2F_{16}$, and Carry is set to 1. After execution of the instruction

RL (IX+7)

memory location 4007_{16} will contain $5F_{16}$, and Carry is 0:

RL (IY+disp)

FD CB d 16

This instruction is identical to RL (IX+disp), but uses the IY register instead of the IX register.

$$\underbrace{\text{RL (HL)}}_{\text{CB 16}}$$

Rotate contents of memory location (specified by the contents of the HL register pair) left one bit through Carry.

RLA — ROTATE ACCUMULATOR LEFT THROUGH CARRY
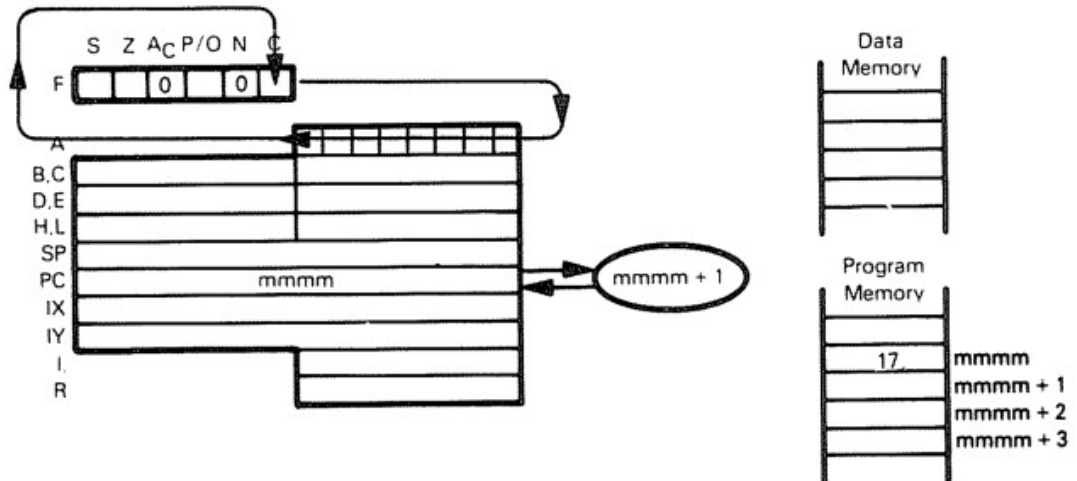
$$\underbrace{\text{RLA}}_{17}$$

Rotate Accumulator contents left one bit through Carry status.

Suppose the Accumulator contains $2A_{16}$ and the Carry status is set to 1. After the instruction

RLA

has executed, the Accumulator will contain $F5_{16}$ and the Carry status will be reset to 0:

Before		After	
Accumulator	Carry	Accumulator	Carry
0 1 1 1 1 0 1 0	1	1 1 1 1 0 1 0 1	0

RLC reg — ROTATE CONTENTS OF REGISTER LEFT CIRCULAR
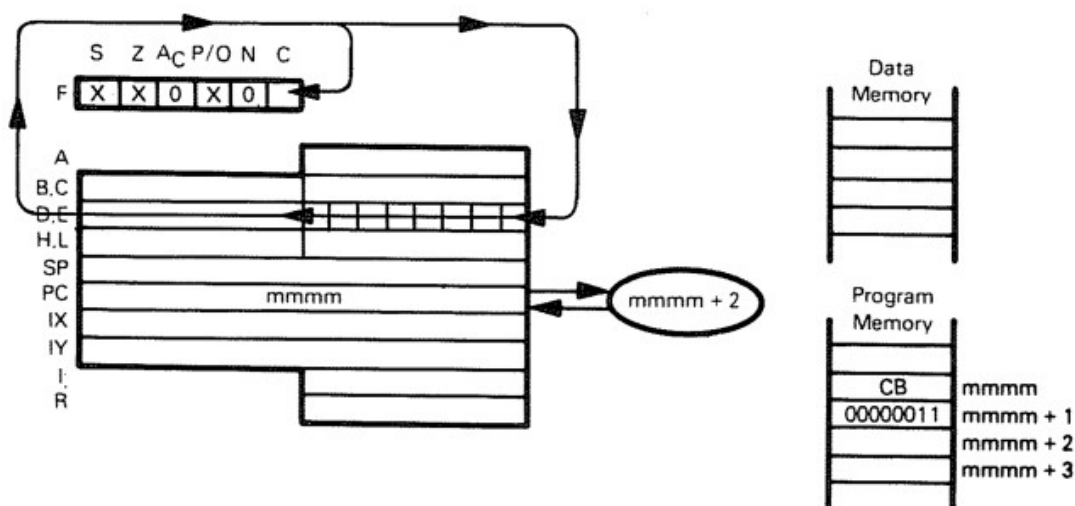
The illustration shows execution of RLC E:


```
CB 000 00 xxx
```

000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A
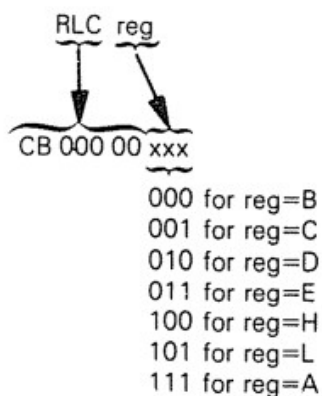
Rotate contents of specified register left one bit. copying bit 7 into Carry.

Suppose Register D contains $A9_{16}$ and Carry is 1. After execution of

RLC D

Register D will contain 53_{16} and Carry will be 1:

0 sets S to 0
4 ones, set P/O to 1

Non-zero result, set Z to 0
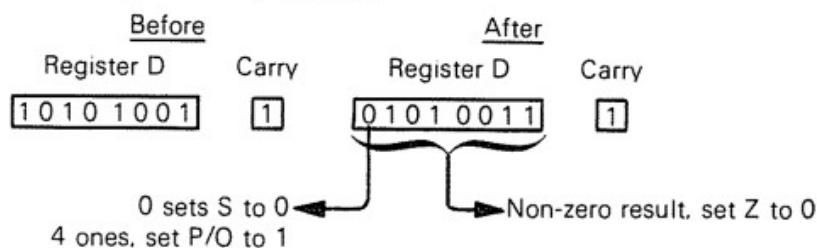
RLC (HL) — ROTATE CONTENTS OF MEMORY LOCATION
RLC (IX+disp) LEFT CIRCULAR
RLC (IY+disp)
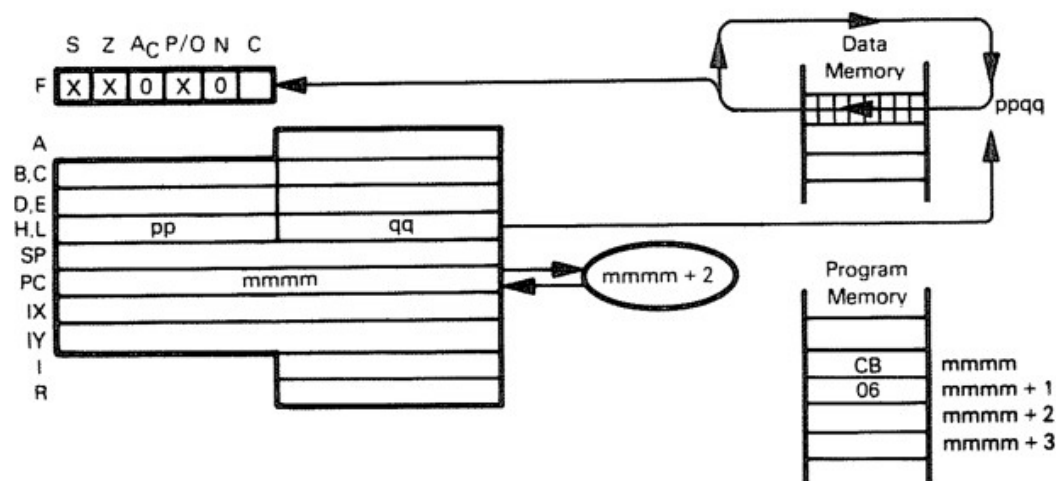
The illustration shows execution of RLC (HL):

$$\underbrace{\text{RLC (HL)}}_{\text{CB 06}}$$

Rotate contents of memory location (specified by the contents of the HL register pair) left one bit, copying bit 7 into Carry.

Suppose register pair HL contains $54FF_{16}$. Memory location $54FF_{16}$ contains $A5_{16}$, and Carry is 0. After execution of

RLC (HL)

memory location $54FF_{16}$ will contain $4B_{16}$, and Carry will be 1:

$$\underbrace{\text{RLC (IX+disp)}}_{\text{DD CB d 06}}$$
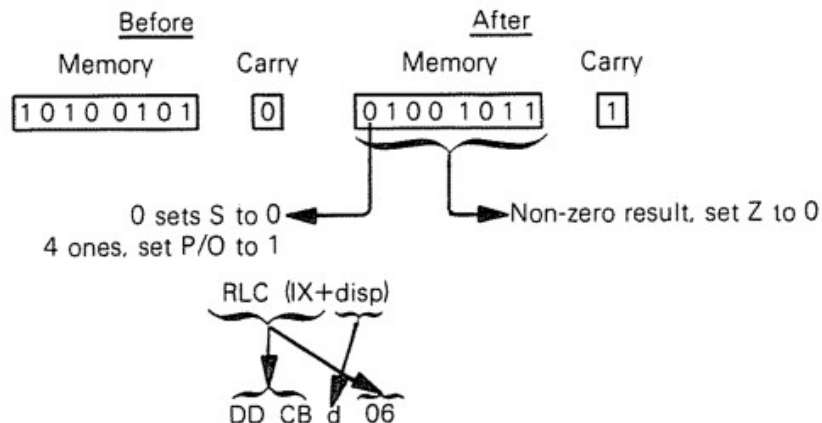
Rotate memory location (specified by the sum of the contents of Index register IX and displacement integer d) left one bit, copying bit 7 into Carry.

Suppose the IX register contains 4000_{16}. Carry is 1, and memory location 4007_{16} contains $2F_{16}$. After the instruction

RLC (IX+7)

has executed, memory location 4007_{16} will contain $5E_{16}$, and Carry will be 0:

Before		After	
Memory	Carry	Memory	Carry
`0 0 1 0 1 1 1 1`	`1`	`0 1 0 1 1 1 1 0`	`0`

0 sets S to 0
5 ones, set P/O to 0

Non-zero result, set Z to 0

RLC (IY+disp)

FD CB d 06

This instruction is identical to RLC (IX+disp), but uses the IY register instead of the IX register.

RLCA — ROTATE ACCUMULATOR LEFT CIRCULAR

S Z A$_C$ P/O N C

F

A

B,C
D,E
H,L
SP
PC   mmmm   mmmm + 1
IX
IY
I
R

Data
Memory

Program
Memory

07   mmmm
       mmmm + 1
       mmmm + 2
       mmmm + 3

RLCA

07
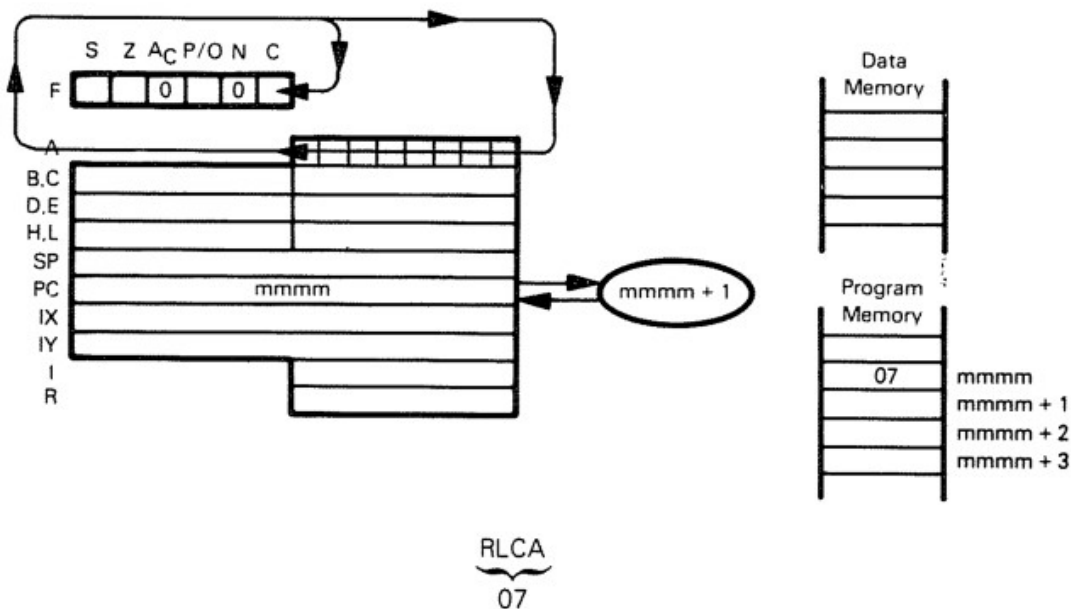
Rotate Accumulator contents left one bit, copying bit 7 into Carry.

Suppose the Accumulator contains $7A_{16}$ and the Carry status is set to 1. After the instruction

RLCA

has executed, the Accumulator will contain $F4_{16}$ and the Carry status will be reset to 0:

Before		After	
Accumulator	Carry	Accumulator	Carry
`0 1 1 1 1 0 1 0`	`1`	`1 1 1 1 0 1 0 0`	`0`
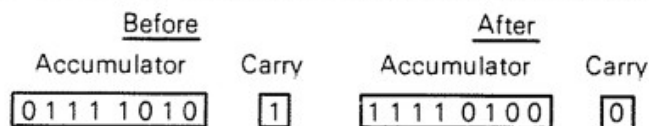
RLCA should be used as a logical instruction.

RLD — ROTATE ONE BCD DIGIT LEFT BETWEEN THE ACCUMULATOR AND MEMORY LOCATION
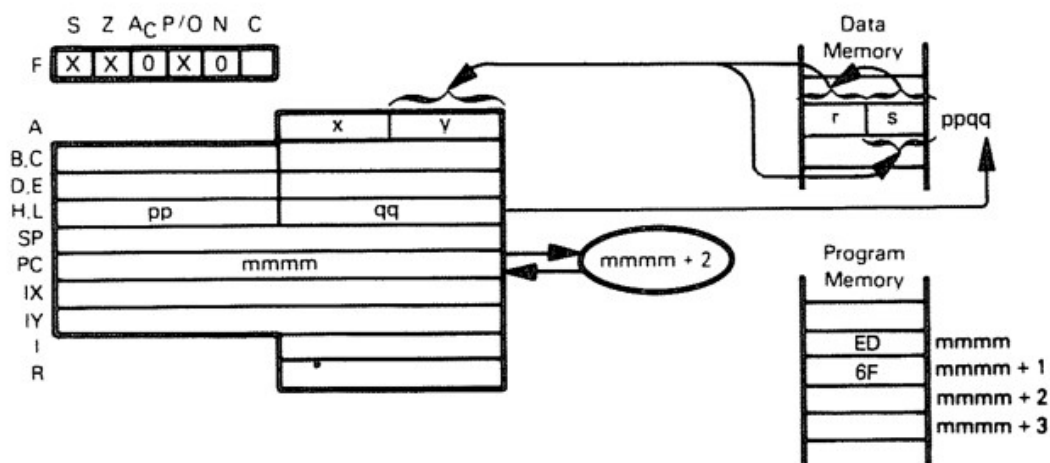
RLD
ED 6F

The four low-order bits of a memory location (specified by the contents of register pair HL) are copied into the four high-order bits of the same memory location. The previous contents of the four high-order bits of that memory location are copied into the four low-order bits of the Accumulator. The previous four low-order bits of the Accumulator are copied into the four low-order bits of the specified memory location.

Suppose the Accumulator contains $7F_{16}$, HL register pair contains 4000_{16}, and memory location 4000_{16} contains 12_{16}. After execution of the instruction

RLD

the Accumulator will contain 71_{16} and memory location 4000_{16} will contain $2F_{16}$:
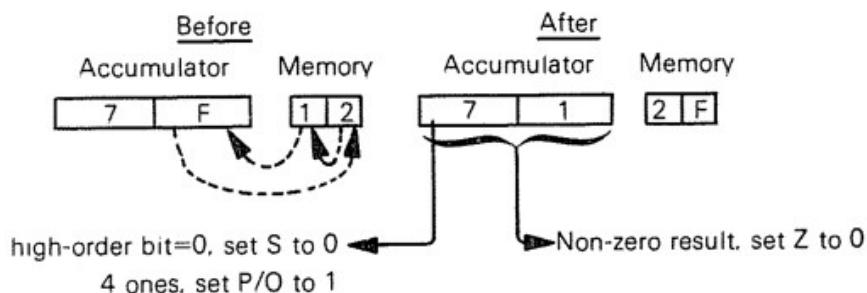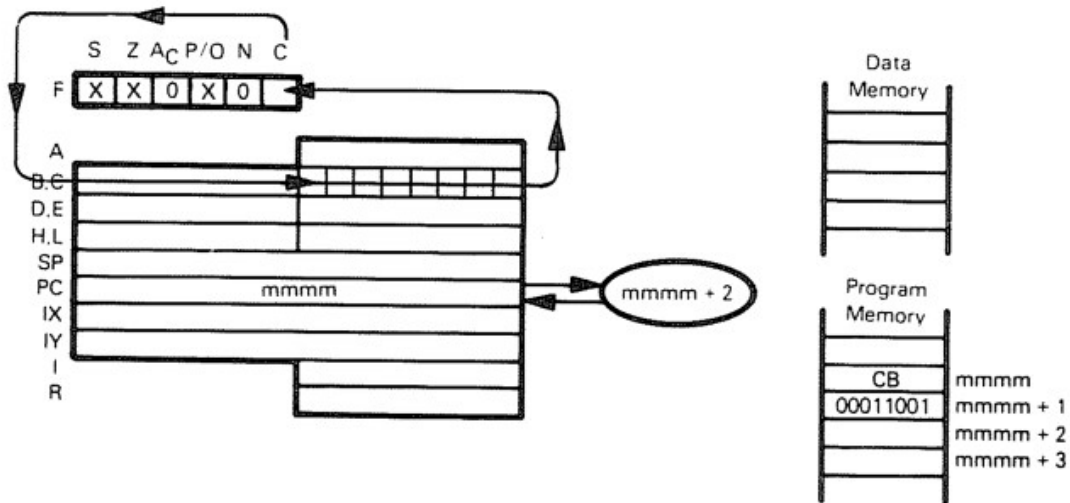
high-order bit=0, set S to 0
4 ones, set P/O to 1

Non-zero result, set Z to 0

RR reg — ROTATE CONTENTS OF REGISTER RIGHT THROUGH CARRY

The illustration shows execution of RR C:


```
CB 00011 xxx
            000 for reg=B
            001 for reg=C
            010 for reg=D
            011 for reg=E
            100 for reg=H
            101 for reg=L
            111 for reg=A
```
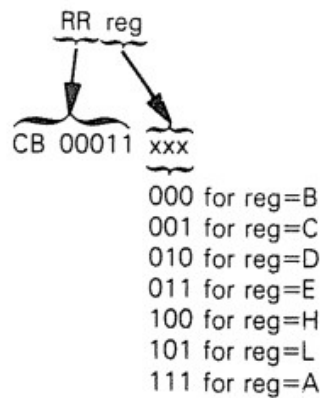
Rotate contents of specified register right one bit through Carry.

Suppose Register H contains $0F_{16}$ and Carry is set to 1. After the instruction

$$RR\ H$$

has executed, Register H will contain 87_{16}, and Carry will be 1:
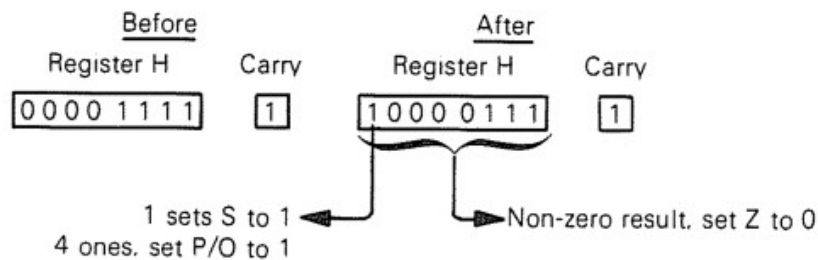
RR (HL) — ROTATE CONTENTS OF MEMORY LOCATION
RIGHT THROUGH CARRY
RR (IX+disp)
RR (IY+disp)
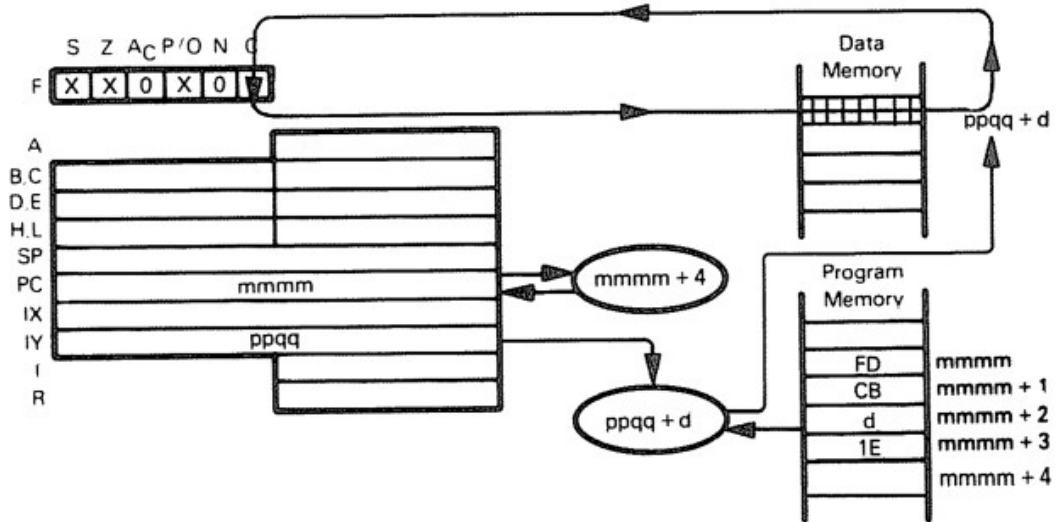
The illustration shows execution of RR (IY+disp):
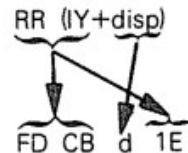
RR (IY+disp)

FD CB d 1E

Rotate contents of memory location (specified by the sum of the contents of the IY register and the displacement value d) right one bit through Carry.

Suppose the IY register contains 4500_{16}, memory location $450F_{16}$ contains $1D_{16}$, and Carry is set to 0. After execution of the instruction

RR (IY+0FH)

memory location $450F_{16}$ will contain $0E_{16}$, and Carry will be 1.

RR (IX+disp)

DD CB d 1E
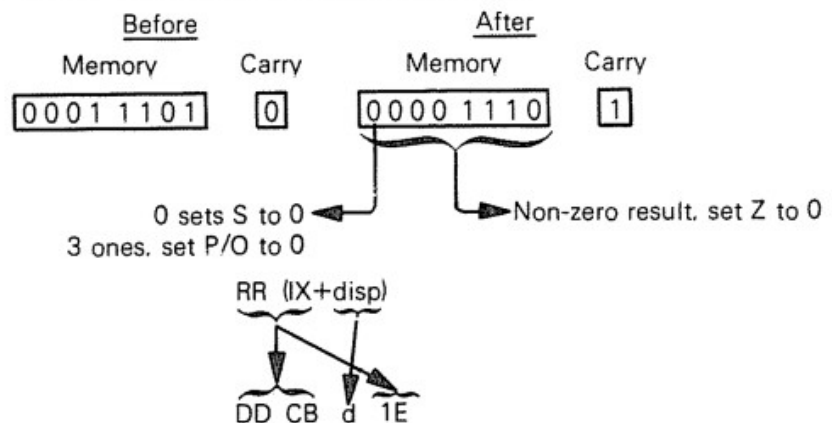
This instruction is identical to RR (IY+disp), but uses the IX register instead of the IY register.

$$\underbrace{\text{RR (HL)}}_{\text{CB 1E}}$$

Rotate contents of memory location (specified by the contents of the HL register pair) right one bit through Carry.

RRA — ROTATE ACCUMULATOR RIGHT THROUGH CARRY

$$\underbrace{\text{RRA}}_{\text{1F}}$$
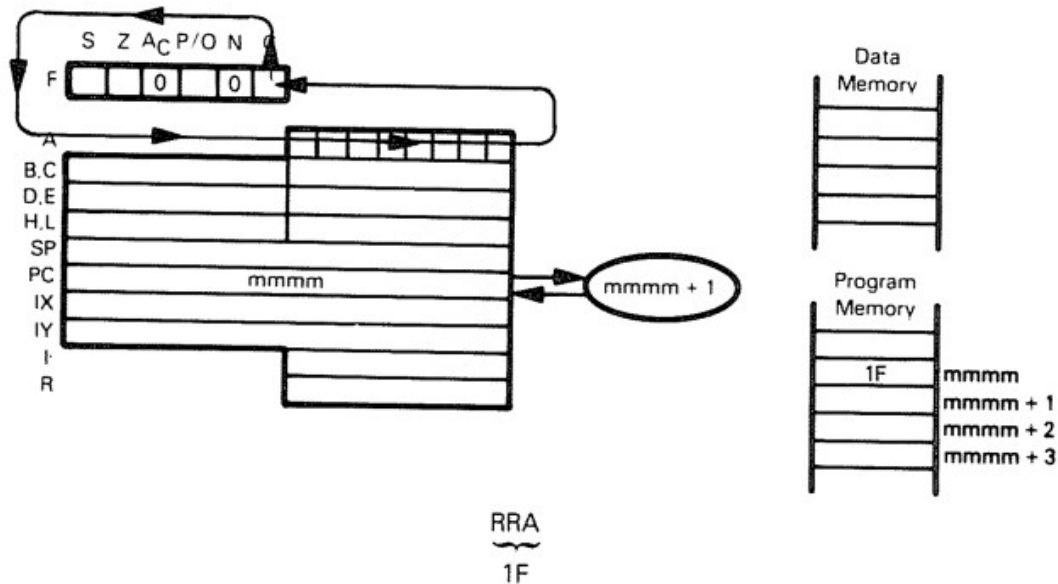
Rotate Accumulator contents right one bit through Carry status.

Suppose the Accumulator contains $7A_{16}$ and the Carry status is set to 1. After the instruction

<div align="center">RRA</div>

has executed, the Accumulator will contain BD_{16} and the Carry status will be reset to 0:

Before		After	
Accumulator	Carry	Accumulator	Carry
0 1 1 1 1 0 1 0	1	1 0 1 1 1 1 0 1	0

RRC reg — ROTATE CONTENTS OF REGISTER RIGHT CIRCULAR
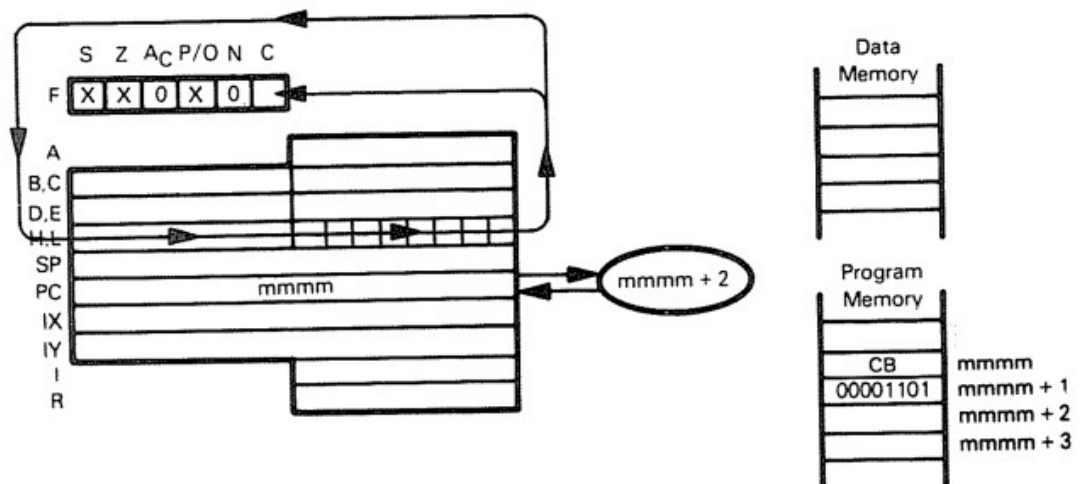
The illustration shows execution of RRC L:

RRC reg

CB 00001 xxx

000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A
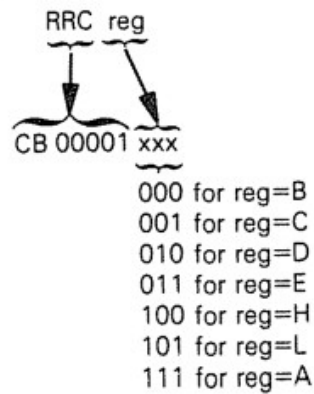
Rotate contents of specified register right one bit circularly, copying bit 0 into the Carry status.

Suppose Register D contains $A9_{16}$ and Carry is 0. After execution of

RRC D

Register D will contain $D4_{16}$, and Carry will be 1:

Before		After	
Register D	Carry	Register D	Carry
1 0 1 0 1 0 0 1	0	1 1 0 1 0 1 0 0	1

1 sets S to 1
4 ones, set P/O to 1

Non-zero result, set Z to 0
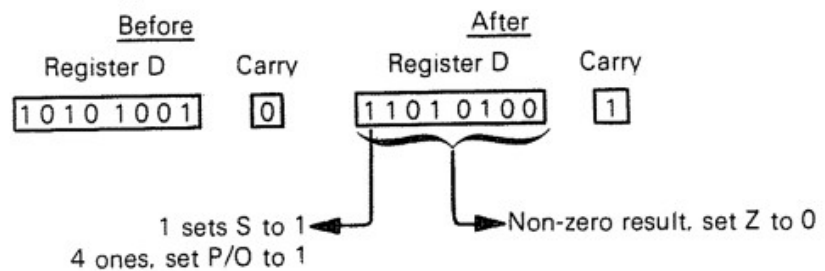
RRC (HL) — ROTATE CONTENTS OF MEMORY LOCATION
RRC (IX+disp) RIGHT CIRCULAR
RRC (IY+disp)
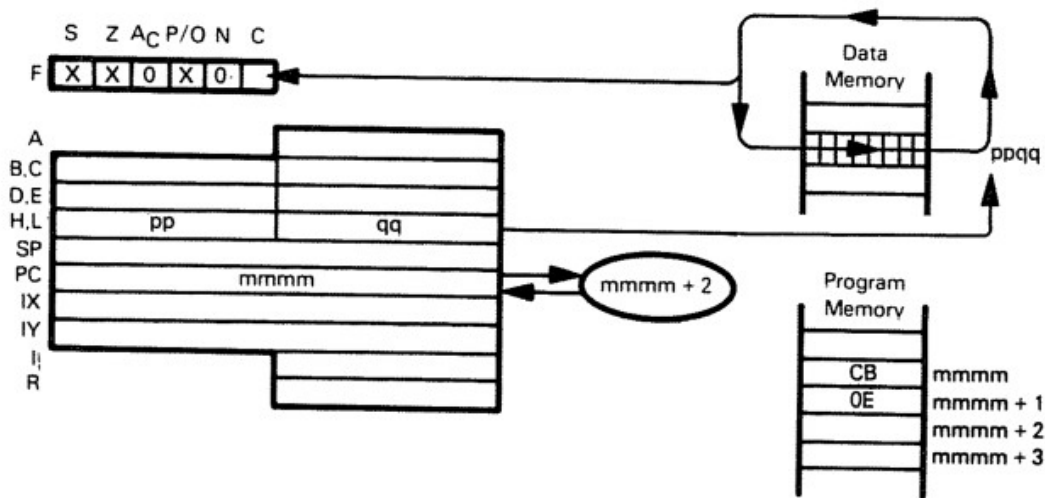
The illustration shows execution of RRC (HL):

$$\underbrace{\text{RRC (HL)}}_{\text{CB 0E}}$$

Rotate contents of memory location (specified by the contents of the HL register pair) right one bit circularly, copying bit 0 into the Carry status.

Suppose the HL register pair contains 4500_{16}, memory location 4500_{16} contains 34_{16}, and Carry is set to 1. After execution of

RRC (HL)

memory location 4500_{16} will contain $1A_{16}$, and Carry will be 0:
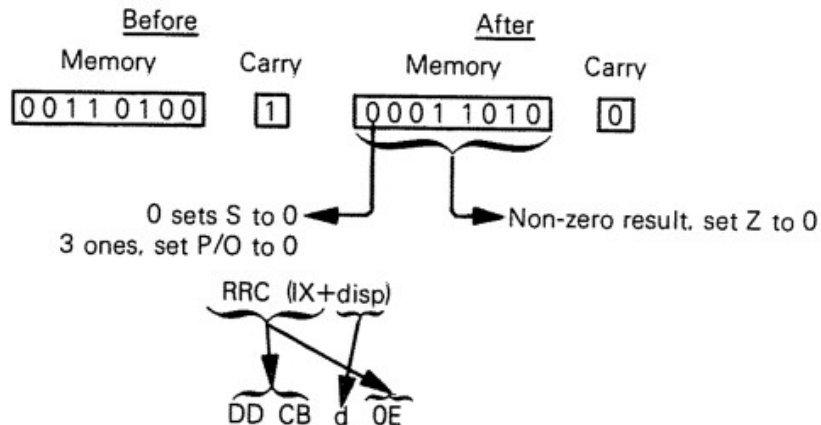
$$\underbrace{\text{RRC (IX+disp)}}$$

DD CB d 0E

Rotate contents of memory location (specified by the sum of the contents of the IX

register and the displacement value d) right one bit circularly, copying bit 0 into the Carry status.

$$RRC \quad (IY+disp)$$

FD CB d 0E
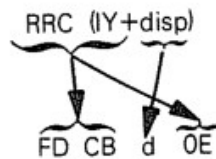
This instruction is identical to the RRC (IX+disp) instruction, but uses the IY register instead of the IX register.

RRCA — ROTATE ACCUMULATOR RIGHT CIRCULAR
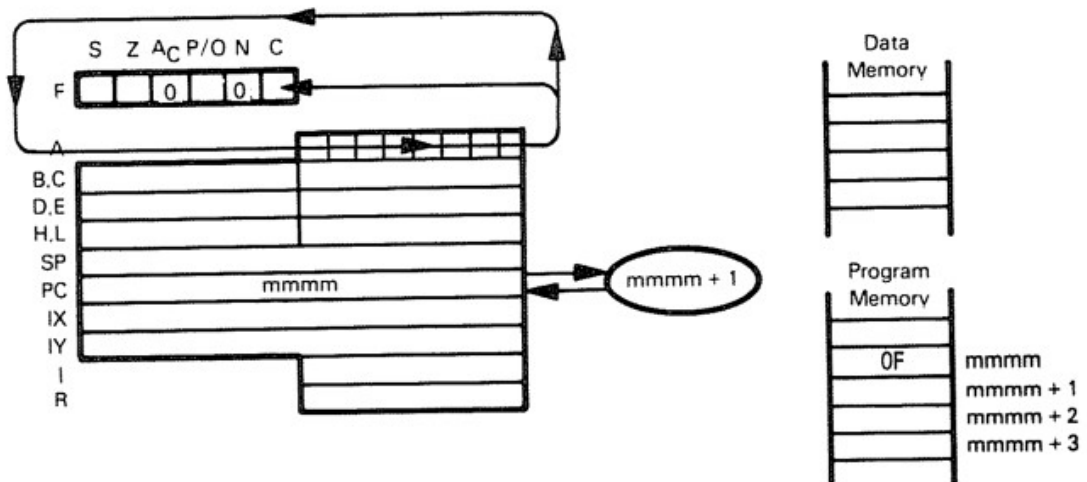
RRCA
⏜
0F

Rotate Accumulator contents right one bit circularly, copying bit 0 into the Carry status.

Suppose the Accumulator contains $7A_{16}$ and the Carry status is set to 1. After the instruction

RRCA

has executed, the Accumulator will contain $3D_{16}$ and the Carry status will be reset to 0:

Before		After	
Accumulator	Carry	Accumulator	Carry
0 1 1 1 1 0 1 0	1	0 0 1 1 1 1 0 1	0

RRCA should be used as a logical instruction.

3-142

RRD — ROTATE ONE BCD DIGIT RIGHT BETWEEN THE ACCUMULATOR AND MEMORY LOCATION
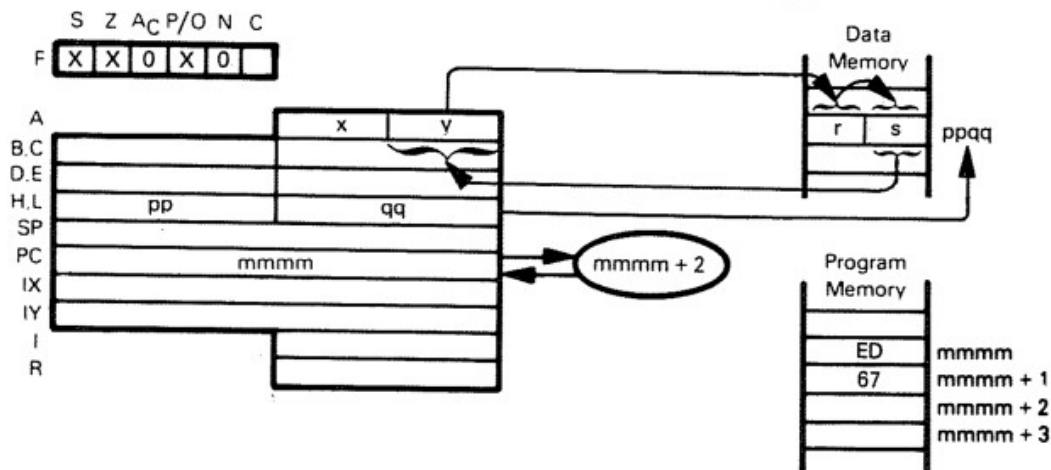
RRD
ED 67

The four high-order bits of a memory location (specified by the contents of register pair HL) are copied into the four low-order bits of the same memory location. The previous contents of the four low-order bits are copied into the four low-order bits of the Accumulator. The previous four low-order bits of the Accumulator are copied into the four high-order bits of the specified memory location.

Suppose the Accumulator contains $7F_{16}$, HL register pair contains 4000_{16}, and memory location 4000_{16} contains 12_{16}. After execution of the instruction

RRD

the Accumulator will contain 72_{16} and memory location 4000_{16} will contain $F1_{16}$:
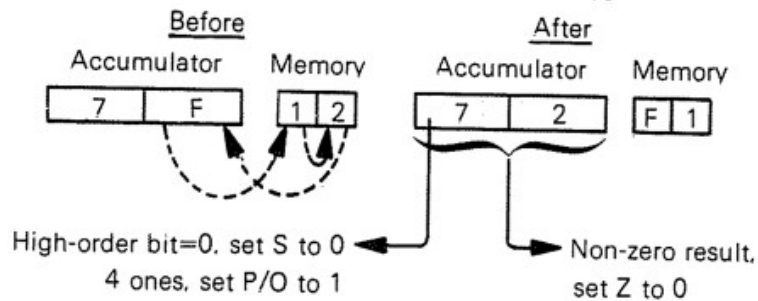
RST n — RESTART
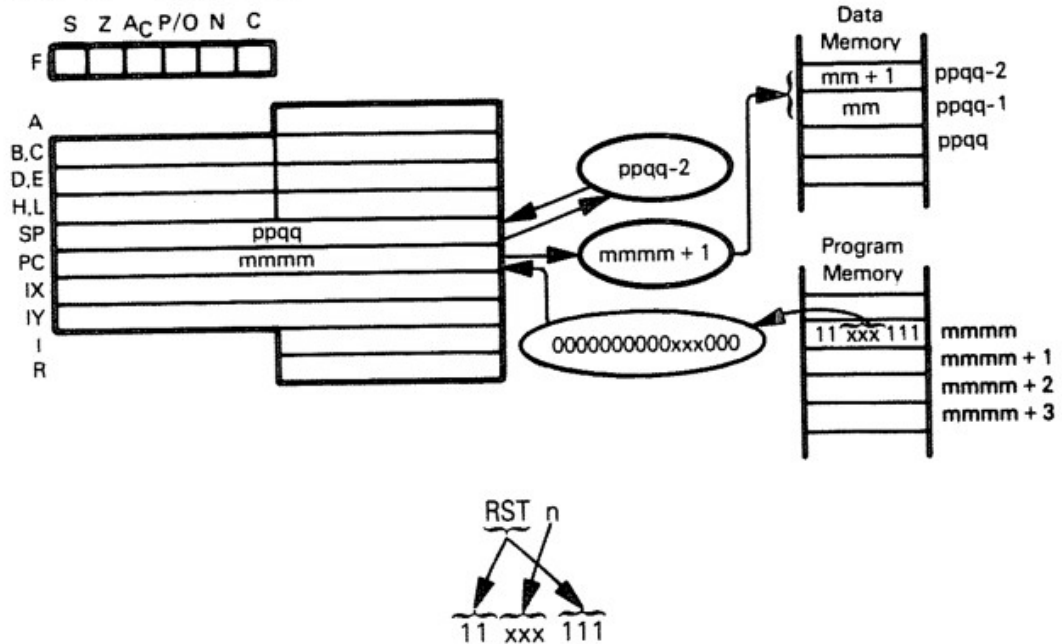
Call the subroutine origined at the low memory address specified by n.

When the instruction

RST 18H

has executed, the subroutine origined at memory location 0018_{16} is called. The previous Program Counter contents are pushed to the top of the stack.

Usually, the RST instruction is used in conjunction with interrupt processing, as described in Chapter 12.

If your application does not use all RST instruction codes to service interrupts, do not overlook the possibility of calling subroutines using RST instructions. Origin frequently used subroutines at appropriate RST addresses, and these subroutines can be called with a single-byte RST instruction instead of a three-byte CALL instruction.

| SUBROUTINE CALL USING RST |

SBC A,data — SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR WITH BORROW

$$\underbrace{\text{SBC A,}}_{\text{DE}} \quad \underbrace{\text{data}}_{\text{yy}}$$
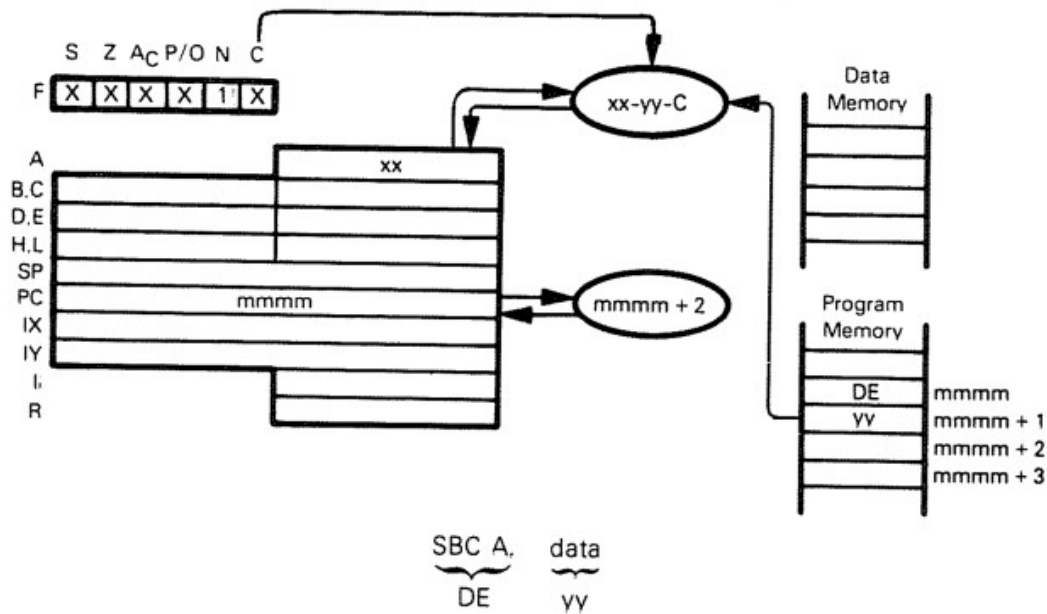
Subtract the contents of the second object code byte and the Carry status from the Accumulator.

Suppose $xx=3A_{16}$ and Carry=1. After the instruction

SBC A,7CH

has executed, the Accumulator will contain BD_{16}.

```
         3A  =  0011   1010
Twos comp of 7C  =  1000   0100
Twos comp of Carry  =  1111   1111
                    1,011   1101
```

1 sets S to 1 ◄──────────────── Non-zero result, set Z to 0

Borrow, set C to 1 ◄──────────── Borrow, set A_C to 1

1 ∀ 1=0, set P/O to 0 ──────── Subtract instruction, set N to 1
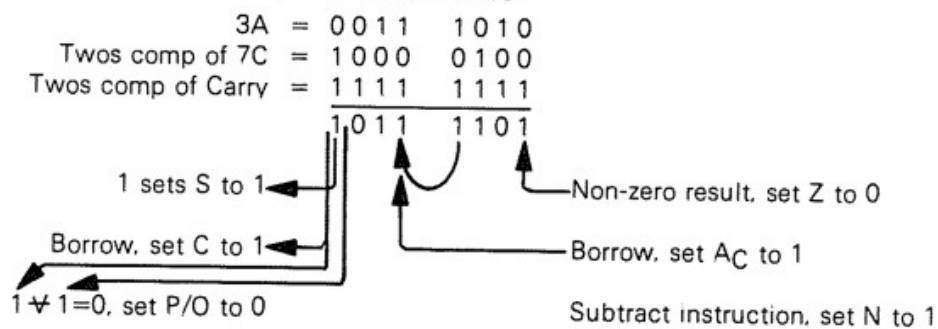
The Carry flag is set to 1 for a borrow and reset to 0 if there is no borrow.

SBC A,reg — SUBTRACT REGISTER WITH BORROW FROM ACCUMULATOR
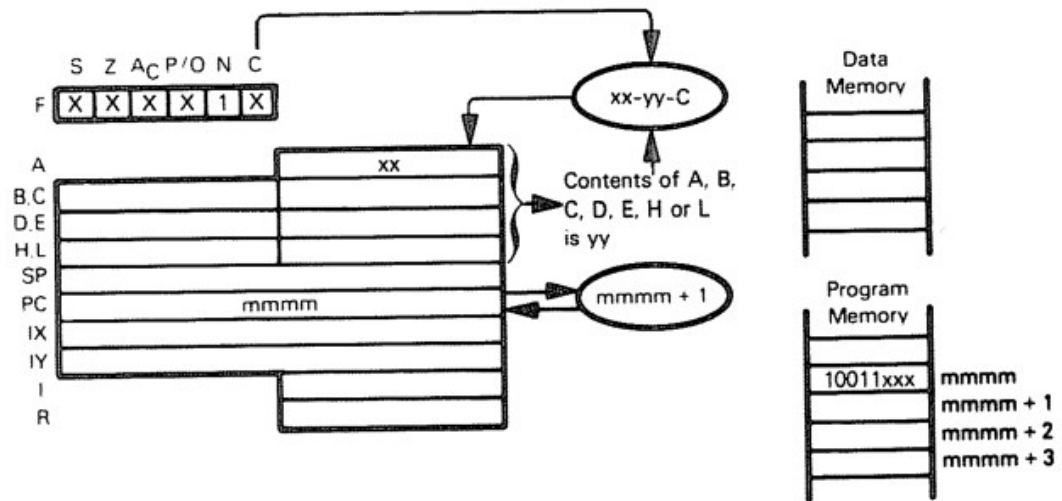

```
SBC A,   reg
 ~~~~~   ~~~
10011    xxx
         000  for reg=B
         001  for reg=C
         010  for reg=D
         011  for reg=E
         100  for reg=H
         101  for reg=L
         111  for reg=A
```

Subtract the contents of the specified register and the Carry status from the Accumulator.

Suppose $xx=E3_{16}$, Register E contains $A0_{16}$, and Carry=1. After the instruction

SBC A,E

has executed, the Accumulator will contain 42_{16}.

```
              E3  =  1 1 1 0   0 0 1 1
Two's comp of A0  =  0 1 1 0   0 0 0 0
 Two's comp of 1  =  1 1 1 1   1 1 1 1
                    0 1 0 0   0 0 1 0
```

0 sets S to 0 ◄─────────────────────────────── Non-zero result, set Z to 0

No borrow, set C to 0 ◄─────────────────────── No borrow, set A_C to 0

$1 \forall 1=0$, set P/O to 0 ─────────── Subtract instruction, set N to 1

The Carry flag is set to 1 for a borrow and reset to 0 if there is no borrow.

SBC A,(HL) —
SBC A,(IX+disp)
SBC A,(IY+disp)

SUBTRACT MEMORY AND CARRY FROM ACCUMULATOR
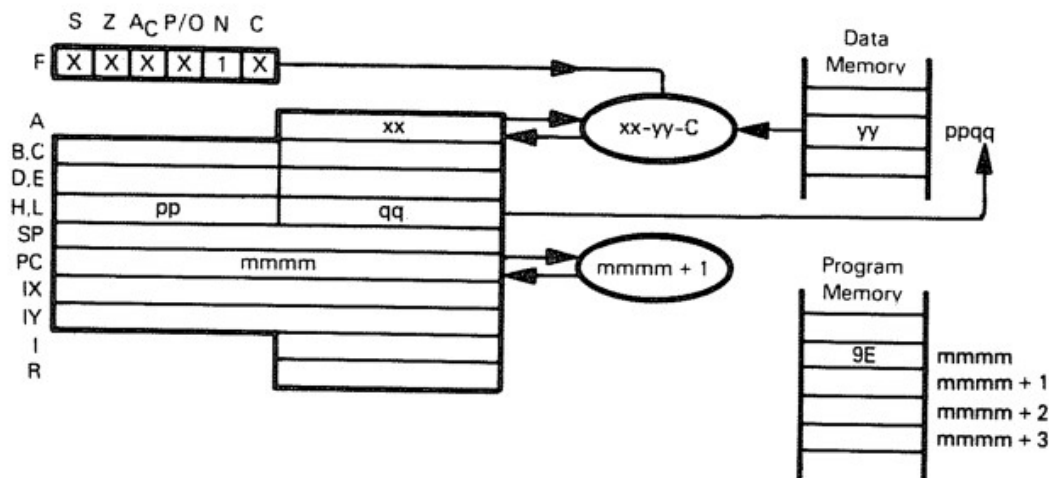
The illustration shows execution of SBC A,(HL):

$$\underbrace{\text{SBC A,(HL)}}_{\text{9E}}$$

Subtract the contents of memory location (specified by the contents of the HL register pair) and the Carry from the Accumulator.

Suppose Carry=0, ppqq=4000_{16}, xx=$3A_{16}$, and memory location 4000_{16} contains $7C_{16}$. After execution of the instruction

SBC A,(HL)

the Accumulator will contain BE_{16}.
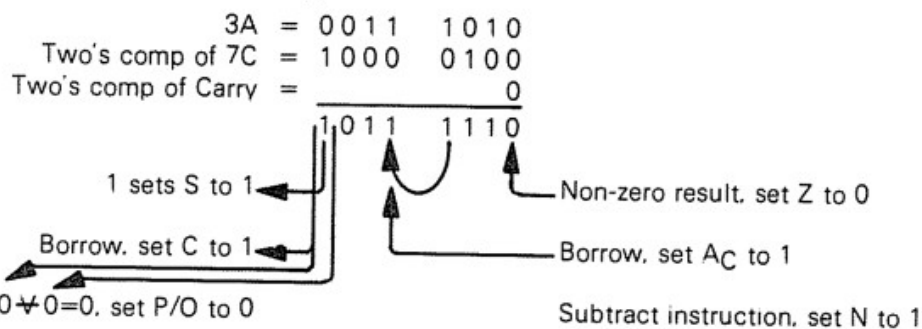
The Carry flag is set to 1 for a borrow and reset to 0 if there is no borrow.

$$\underbrace{\text{SBC A,(IX+disp)}}_{\text{DD 9E} \qquad \text{d}}$$

Subtract the contents of memory location (specified by the sum of the contents of the IX register and the displacement value d) and the Carry from the Accumulator.

$$\underbrace{\text{SBC A,(IY+disp)}}_{\text{FD 9E} \qquad \text{d}}$$

This instruction is identical to the SBC A,(IX+disp) instruction, except that it uses the IY register instead of the IX register.

SBC HL,rp — SUBTRACT REGISTER PAIR WITH CARRY FROM H AND L
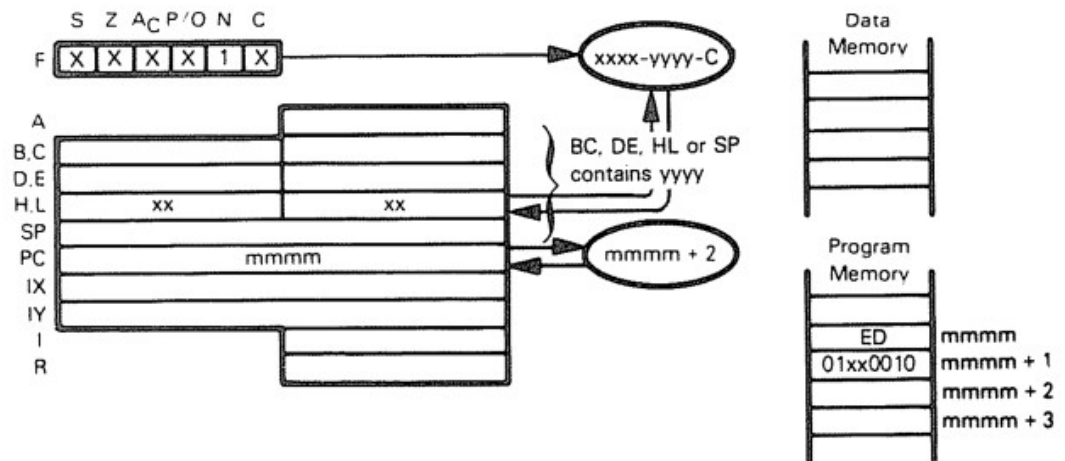
S Z A$_C$ P/O N C

F [X][X][X][X][1][X] → xxxx-yyyy-C

Data Memory

A
B,C
D,E
H,L   xx   xx
SP
PC   mmmm
IX
IY
I
R

BC, DE, HL or SP contains yyyy

mmmm + 2

Program Memory

ED	mmmm
01xx0010	mmmm + 1
	mmmm + 2
	mmmm + 3

SBC HL, rp

01 xx 0010

00 for rp is register pair BC
01 for rp is register pair DE
10 for rp is register pair HL
11 for rp is Stack Pointer

Subtract the contents of the designated register pair and the Carry status from the HL register pair.

Suppose HL contains F4A2$_{16}$. BC contains A034$_{16}$, and Carry=0. After the instruction

SBC HL,BC

has executed, the HL register pair will contain 546E$_{16}$:

```
Two's comp of F4A2  =  1111  0100  1010  0010
Two's comp of A034  =  0101  1111  1100  1100
Two's comp of Carry =                       0
                      ─────────────────────────
                      0101  0100  0110  1110
```

0 sets S to 0 ◄

Non-zero result, set Z to 0

No borrow, set C to 0 ◄

No borrow.

1 ∀ 1=0, set P/O to 0

Subtract instruction, set N to 1
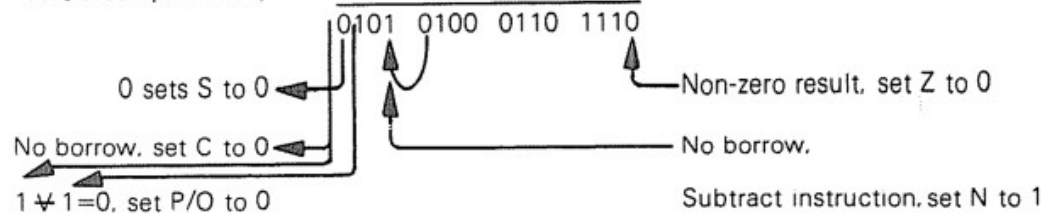
The Carry flag is set to 1 for a borrow and reset to 0 if there is no borrow.

SCF — SET CARRY FLAG

SCF
~~
37
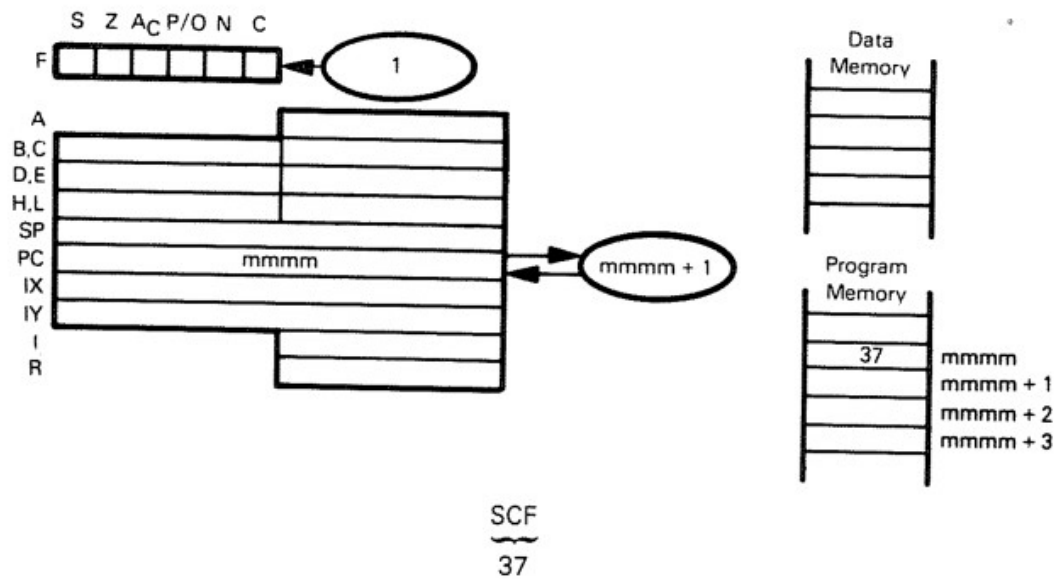
When the SCF instruction is executed, the Carry status is set to 1 regardless of its previous value. No other statuses or register contents are affected.

SET b,reg — SET INDICATED REGISTER BIT

S Z A_C P/O N C

(diagram of registers F, A, B,C, D,E, H,L, SP, PC, IX, IY, I, R)

yyyy yyyy

mmmm

1

mmmm + 2

Data
Memory

Program
Memory

CB	mmmm
11bbbxxx	mmmm + 1
	mmmm + 2
	mmmm + 3
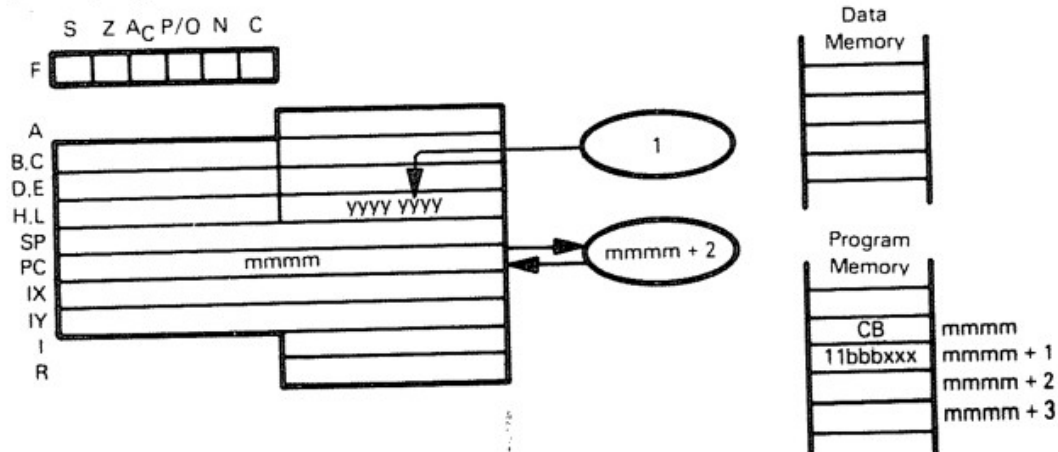
SET b,reg

CB 11bbb xxx

Bit	bbb	xxx	Register
0	000	000	B
1	001	001	C
2	010	010	D
3	011	011	E
4	100	100	H
5	101	101	L
6	110	111	A
7	111		

SET indicated bit within specified register. After the instruction

SET 2,L

has executed, bit 2 in Register L will be set. (Bit 0 is the least significant bit.)

SET b,(HL) — SET BIT b OF INDICATED MEMORY POSITION
SET b,(IX+disp)
SET b,(IY+disp)
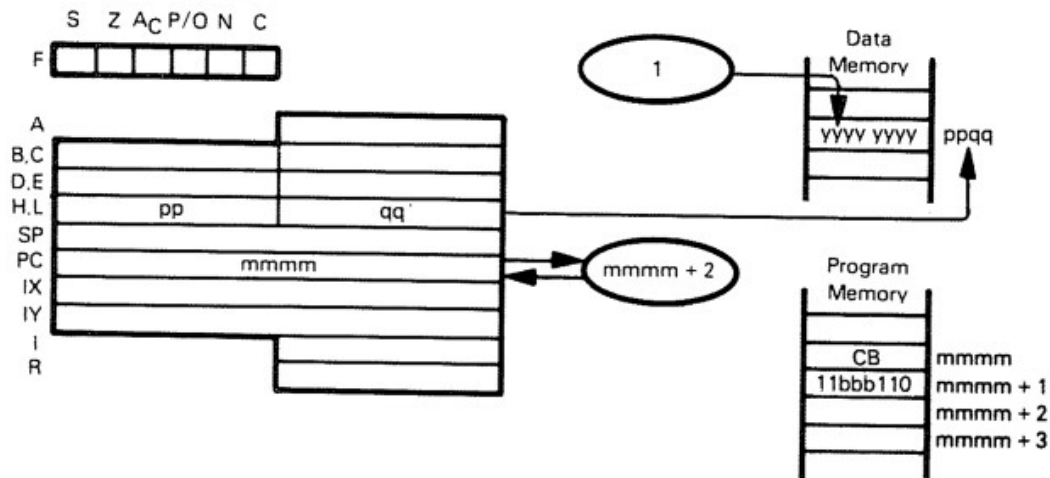
The illustration shows execution of SET b,(HL). Bit 0 is the least significant bit.

SET b,(HL)

CB 11 bbb 110

Bit Set	bbb
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Set indicated bit within memory location indicated by HL.

Suppose HL contains 4000_{16}. After the instruction

SET 5,(HL)

has executed, bit 5 in memory position 4000_{16} will be 1.

SET b,(IX+disp)

DD CB d 11 bbb 110
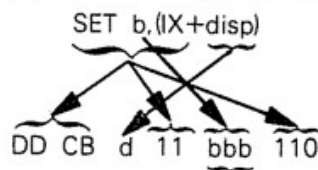
bbb is the same as in SET b,(HL)

Set indicated bit within memory location indicated by the sum of Index Register IX and displacement.

Suppose Index Register IX contains 4000_{16}. After execution of

SET 6,(IX+5H)

bit 6 in memory location 4005_{16} will be 1.

SET b,(IY+disp)

FD CB d 11 bbb 110
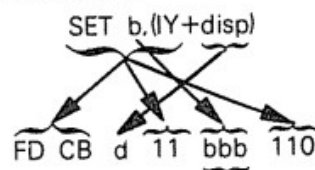
bbb is the same as in SET b,(HL)

This instruction is identical to SET b,(IX+disp), except that it uses the IY register instead of the IX register.

SLA reg — SHIFT CONTENTS OF REGISTER LEFT ARITHMETIC
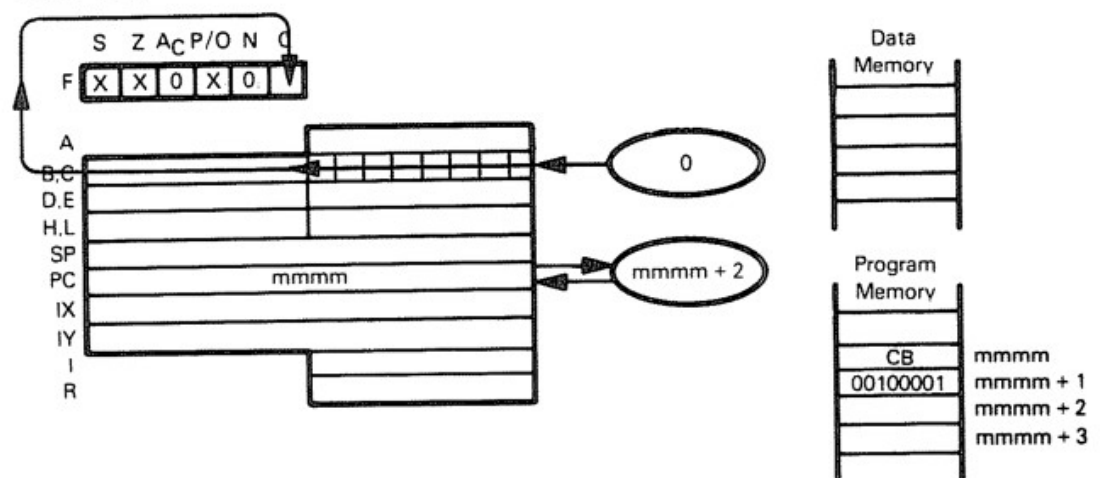
The illustration shows execution of SLA C:

SLA reg

CB 00100 xxx

000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A
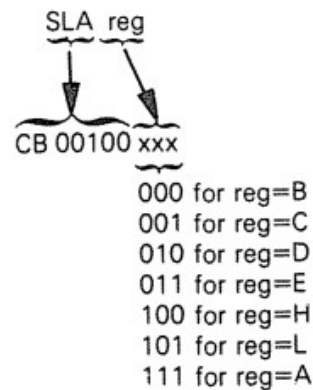
Shift contents of specified register left one bit, resetting the least significant bit to 0.

Suppose Register B contains $1F_{16}$, and Carry=1. After execution of

SLA B

Register B will contain $3E_{16}$ and Carry will be zero.

Before

Register B   Carry   After

Register B   Carry   Register B   Carry

`0 0 0 1 1 1 1 1`   `1`   `0 0 1 1 1 1 1 0`   `0`

0 sets S to 0 ◄───────

5 ones, set P/O to 0   ──► Non-zero result, set Z to 0
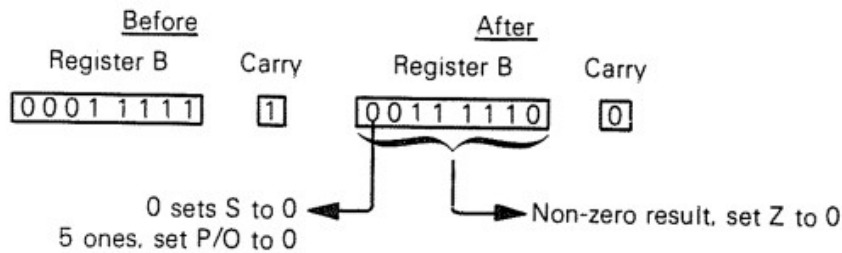
SLA (HL) —   SHIFT CONTENTS OF MEMORY LOCATION
SLA (IX+disp)   LEFT ARITHMETIC
SLA (IY+disp)
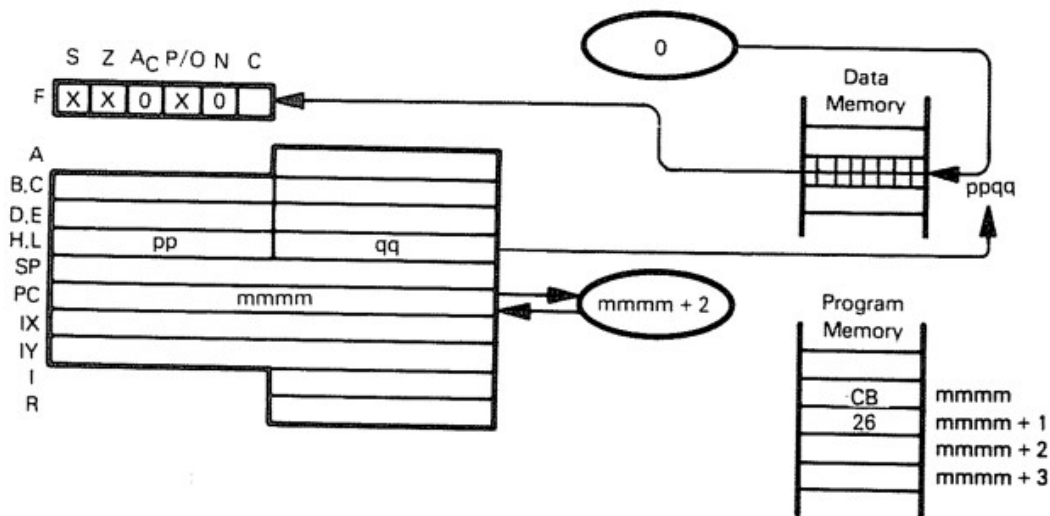
The illustration shows execution of SLA (HL):

SLA (HL)
$\underbrace{\qquad\qquad}$
CB 26

Shift contents of memory location (specified by the contents of the HL register pair) left one bit, resetting the least significant bit to 0.

Suppose the HL register pair contains 4500_{16}, memory location 4500_{16} contains 84_{16}, and Carry=0. After execution of

SLA (HL)

memory location 4500_{16} will contain 08_{16}, and Carry will be 1.
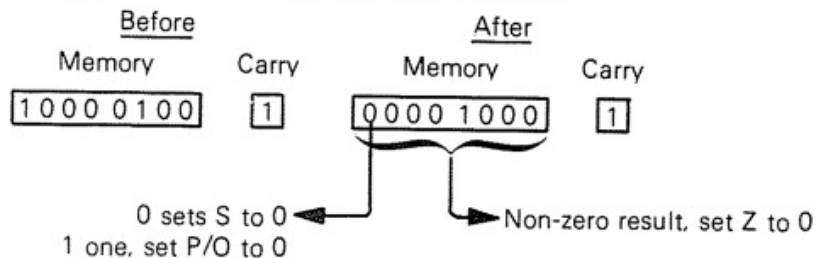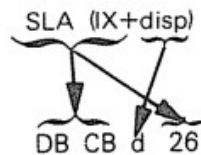
Before   After

Memory   Carry   Memory   Carry

`1 0 0 0 0 1 0 0`   `1`   `0 0 0 0 1 0 0 0`   `1`

0 sets S to 0 ◄───────

1 one, set P/O to 0   ──► Non-zero result, set Z to 0

SLA (IX+disp)

DB CB d 26

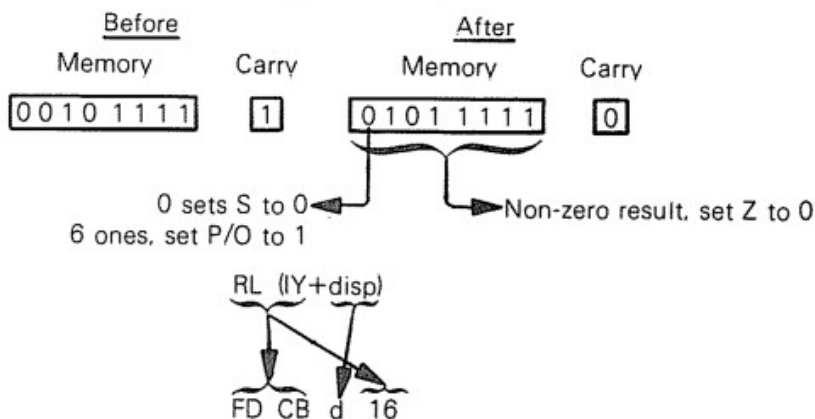Shift contents of memory location (specified by the sum of the contents of the IX register and the displacement value d) left one bit arithmetically, resetting least significant bit to 0.

SLA (IY+disp)

FD CB d 26
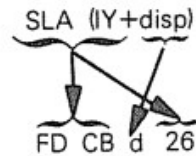
This instruction is identical to SLA (IX+disp), but uses the IY register instead of the IX register.

SRA reg — ARITHMETIC SHIFT RIGHT CONTENTS OF REGISTER

The illustration shows execution of SRA A:

SRA reg

CB 00101 xxx
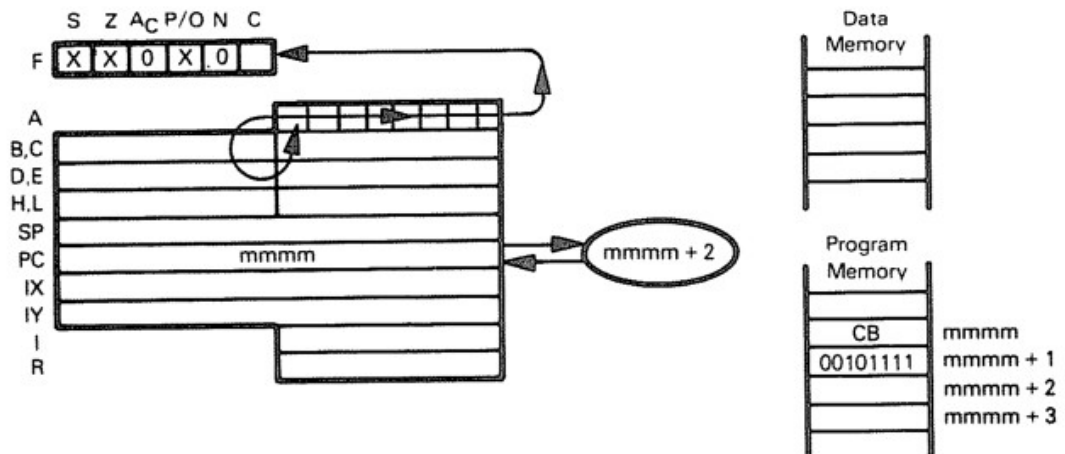
000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A
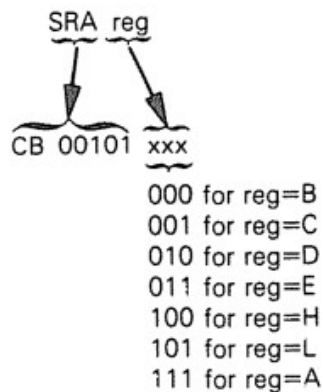
Shift specified register right one bit. Most significant bit is unchanged.

Suppose Register H contains 59_{16}, and Carry=0. After the instruction

SRA H

has executed, Register H will contain $2C_{16}$ and Carry will be 1.

3-154

Before
Register H   C
```
01011001   0
```

After
Register H   C
```
00101100   1
```

0 sets S to 0 ←
3 ones, set P/O to 0

→ Non-zero result, set Z to 0
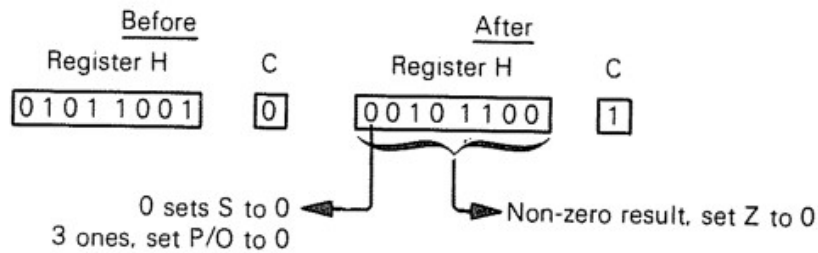
SRA (HL) — ARITHMETIC SHIFT RIGHT CONTENTS OF
SRA (IX+disp) MEMORY POSITION
SRA (IY+disp)
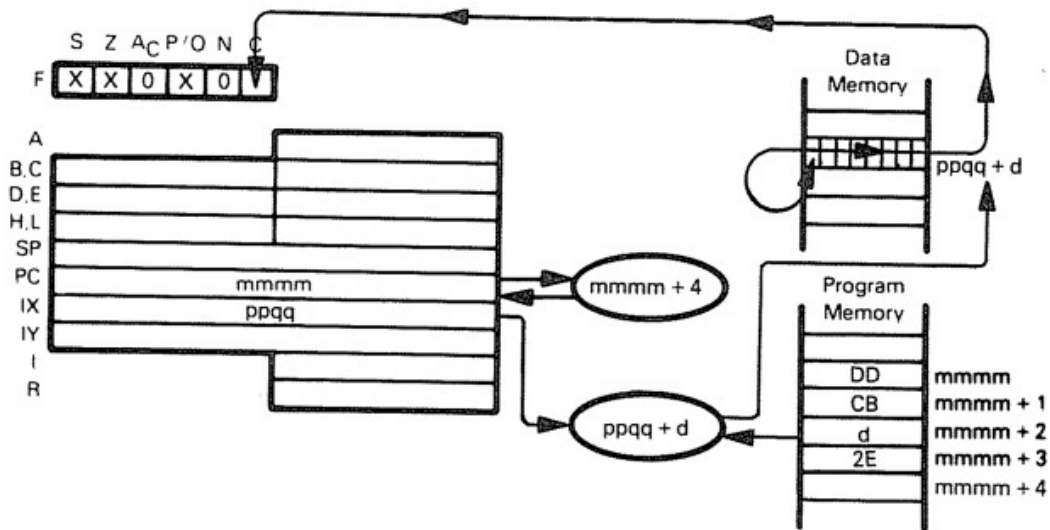
The illustration shows execution of SRA (IX+disp):
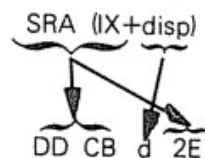
SRA (IX+disp)

DD CB d 2E

Shift contents of memory location (specified by the sum of the contents of Register IX and the displacement value d) right. Most significant bit is unchanged.

Suppose Register IX contains 3400_{16}, memory location $34AA_{16}$ contains 27_{16}, and Carry=1. After execution of

SRA (IX+0AAH)

memory location $34AA_{16}$ will contain 13_{16}, and Carry will be 1.

Before
Memory   Carry
```
00100111   1
```

After
Memory   Carry
```
00010011   1
```

0 sets S to 0 ←
3 ones, set P/O to 0

→ Non-zero result, set Z to 0

SRA (IY+disp)

FD CB d 2E
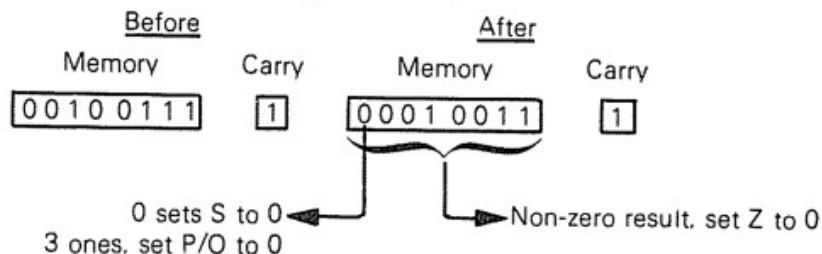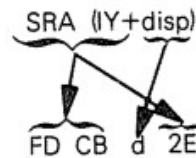
This instruction is identical to SRA (IX+disp), but uses the IY register instead of the IX register.
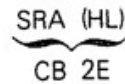
SRA (HL)

CB 2E

Shift contents of memory location (specified by the contents of the HL register pair) right one bit. Most significant bit is unchanged.

SRL reg — SHIFT CONTENTS OF REGISTER RIGHT LOGICAL
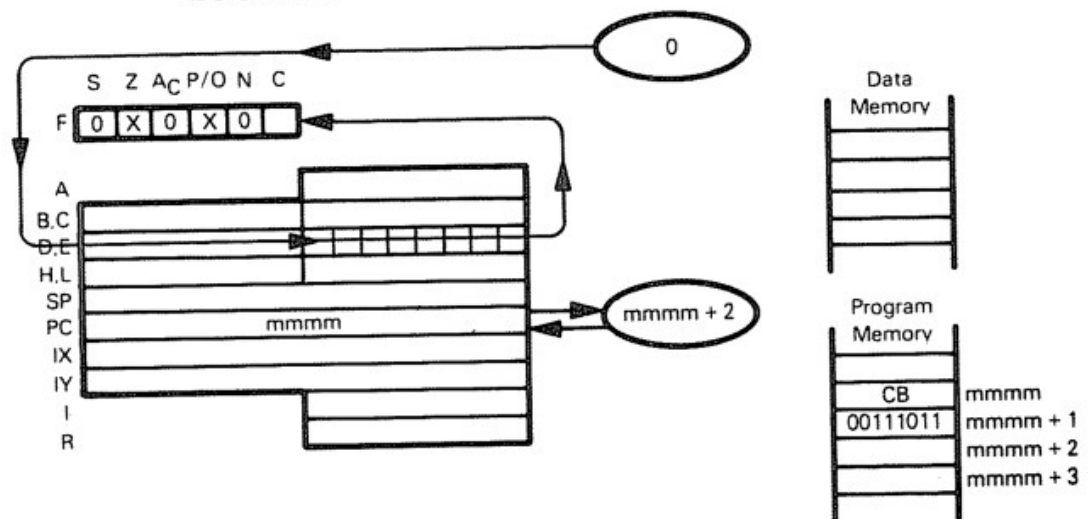
The illustration shows execution of SRL E:
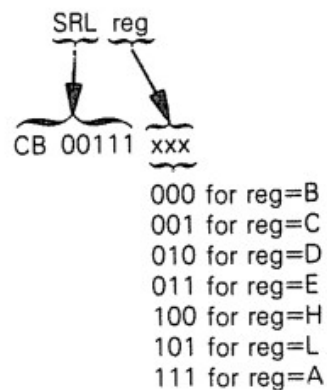
SRL reg

CB 00111 xxx

000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A

Shift contents of specified register right one bit. Most significant bit is reset to 0.

Suppose Register D contains $1F_{16}$, and Carry=0. After execution of

SRL D

Register D will contain $0F_{16}$, and Carry will be 1.

3-156

Register D   Carry   Register D   Carry

| 0 0 0 1 1 1 1 1 | | 0 | | 0 0 0 0 1 1 1 1 | | 1 |

4 ones, set P/O to 1   Non-zero result, set Z to 0
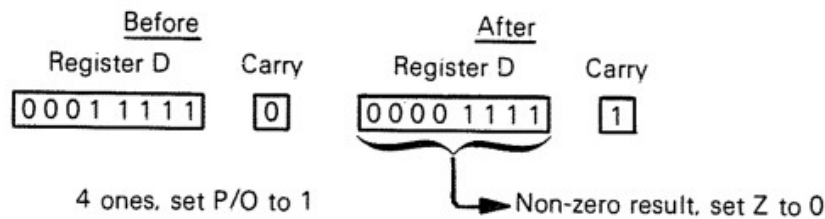
SRL (HL) —
SRL (IX+disp)
SRL (IY+disp)

SHIFT CONTENTS OF MEMORY LOCATION RIGHT LOGICAL
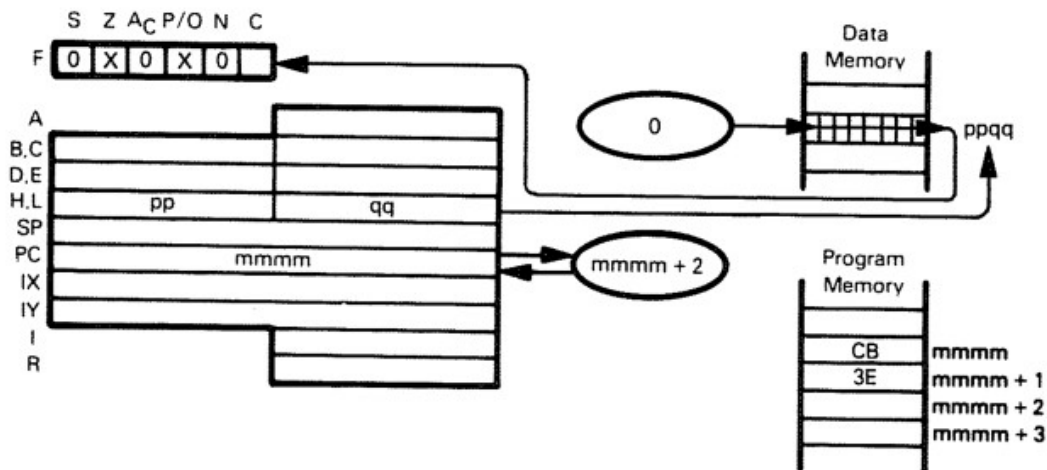
The illustration shows execution of SRL (HL):

$$\underbrace{\text{SRL (HL)}}_{\text{CB 3E}}$$

Shift contents of memory location (specified by the contents of the HL register pair) right one bit. Most significant bit is reset to 0.

Suppose the HL register pair contains 2000_{16}, memory location 2000_{16} contains $8F_{16}$, and Carry=0. After execution of

SRL (HL)

memory location 2000_{16} will contain 47_{16}, and Carry will be 1.

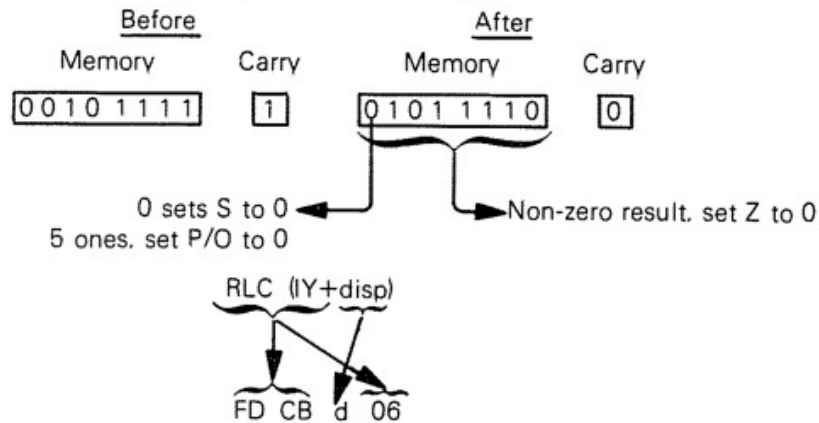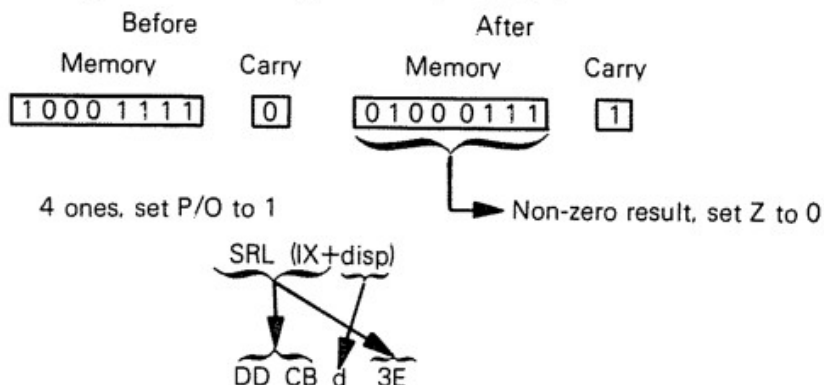Before   After

Memory   Carry   Memory   Carry

| 1 0 0 0 1 1 1 1 | | 0 | | 0 1 0 0 0 1 1 1 | | 1 |

4 ones, set P/O to 1   Non-zero result, set Z to 0

SRL (IX+disp)

DD CB d 3E

Shift contents of memory location (specified by the sum of the contents of the IX register and the displacement value d) right one bit. Most significant bit is reset to 0.

$$\underbrace{SRL}\ \underbrace{(IY+disp)}$$

$$\underbrace{FD}\ \underbrace{DB}\ \underbrace{d}\ \underbrace{3E}$$
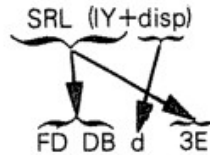
This instruction is identical to SRL (IX+disp), but uses the IY register instead of the IX register.

SUB data — SUBTRACT IMMEDIATE FROM ACCUMULATOR
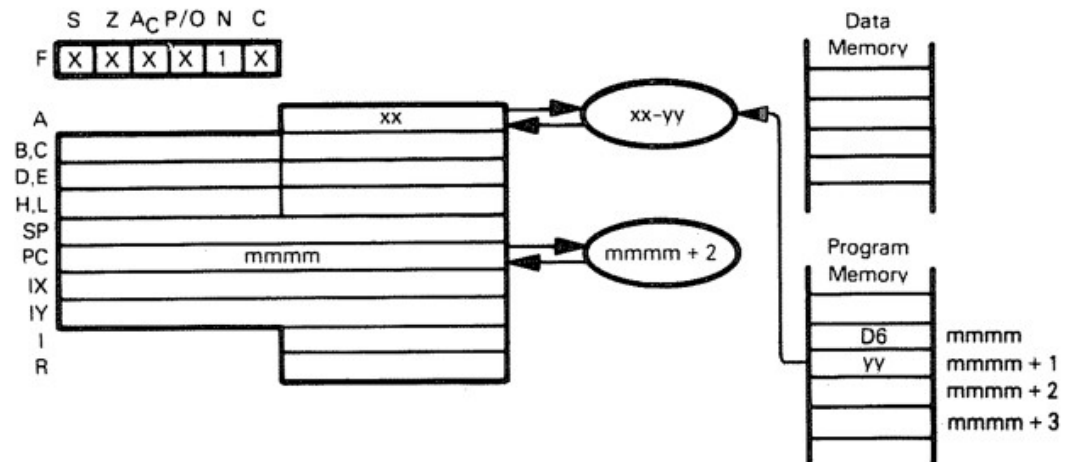
$$\underbrace{SUB}\ \underbrace{data}$$

$$\underbrace{D6}\ \underbrace{yy}$$

Subtract the contents of the second object code byte from the Accumulator.

Suppose $xx=3A_{16}$. After the instruction

SUB 7CH

has executed, the Accumulator will contain BE_{16}.


```
             3A  =  0 0 1 1   1 0 1 0
Two's comp of 7C  =  1 0 0 0   0 1 0 0
                    1,0 1 1   1 1 1 0
```

1 sets S to 1 ◄——

Non-zero result, set Z to 0

Borrow, set C to 1 ◄——

Borrow, set A_C to 1

$0 \not\forall 0 = 0$, set P/O to 0

Subtract instruction, set N to 1
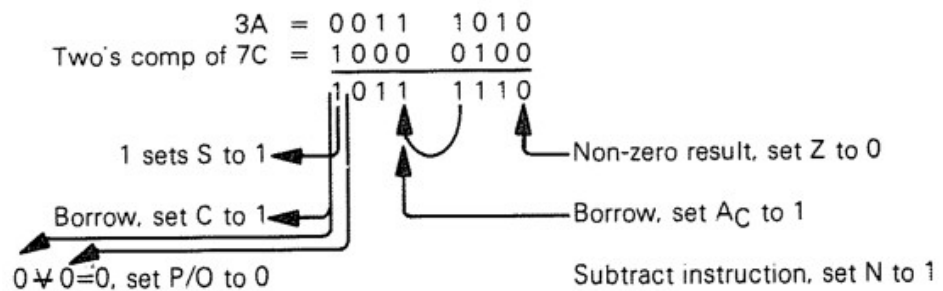
Notice that the resulting carry is complemented.

SUB reg — SUBTRACT REGISTER FROM ACCUMULATOR

```
        S  Z AₐₑP/O N  C
     F  X  X  X  X  1  X
```

```
A
B,C                    xx                 xx-yy
D,E
H,L                         Contents of A, B, C,
SP                          D, E, H or L is yy
PC         mmmm                  mmmm + 1
IX
IY
 I
R
```

Data Memory

Program Memory

```
10010xxx   mmmm
           mmmm + 1
           mmmm + 2
           mmmm + 3
```
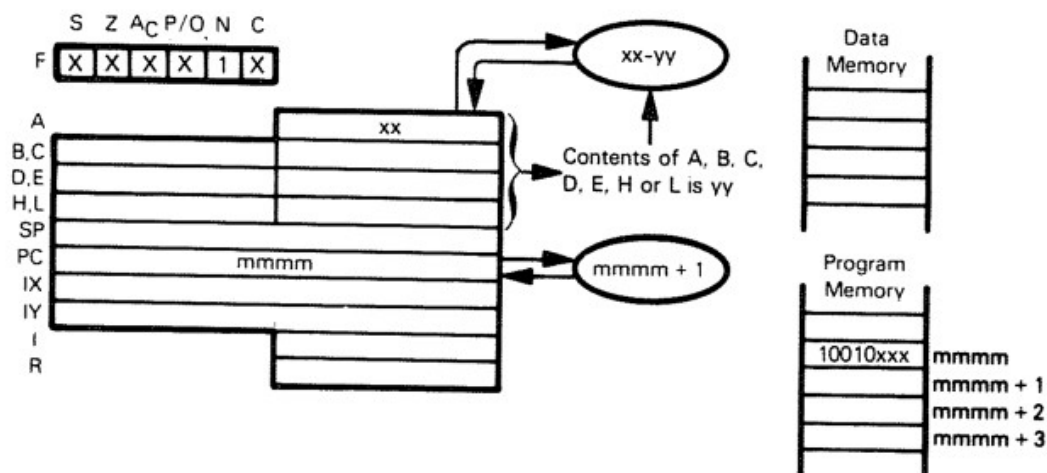
```
SUB      reg
10010    xxx
         000  for reg=B
         001  for reg=C
         010  for reg=D
         011  for reg=E
         100  for reg=H
         101  for reg=L
         111  for reg=A
```

Subtract the contents of the specified register from the Accumulator.

Suppose xx=E3 and Register H contains $A0_{16}$. After execution of

$$SUB \quad H$$

the Accumulator will contain 43_{16}.

```
            E3       = 1 1 1 0   0 0 1 1
     Two's comp of A0 = 0 1 1 0   0 0 0 0
                        0 1 0 0   0 0 1 1
```

0 sets S to 0 ◄— Non-zero result, set Z to 0

No borrow, set C to 0 ◄— No borrow, set Aₑ to 0

1 ⊻ 1=0, set P/O to 0 — Subtract instruction, set N to 1
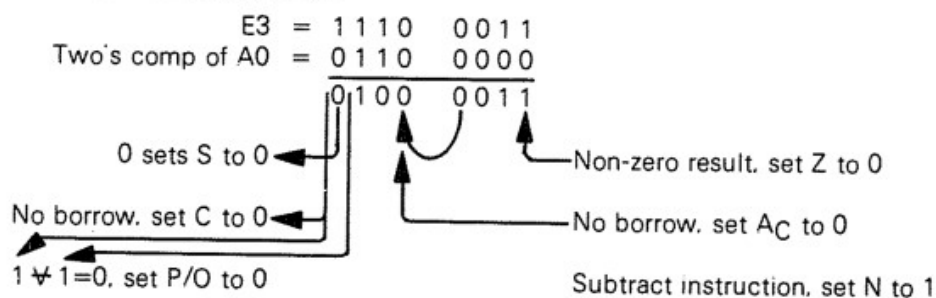
Notice that the resulting carry is complemented.

SUB (HL) — SUBTRACT MEMORY FROM ACCUMULATOR
SUB (IX+disp)
SUB (IY+disp)
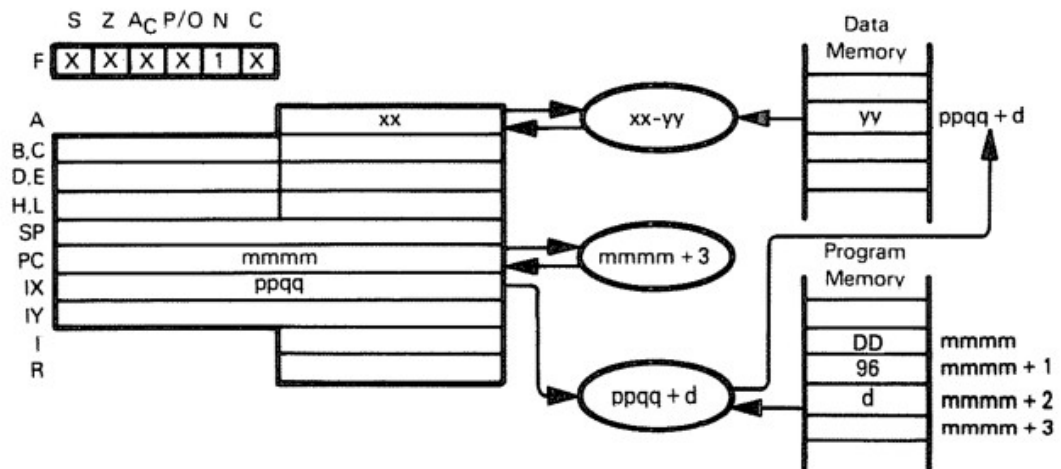
The illustration shows execution of SUB (IX+d):

$$\underbrace{\text{SUB}}\ \underbrace{\text{(IX+disp)}}$$

$$\text{DD 96}\quad\text{d}$$

Subtract contents of memory location (specified by the sum of the contents of the IX register and the displacement value d) from the Accumulator.

Suppose $ppqq=4000_{16}$, $xx=FF_{16}$, and memory location $40FF_{16}$ contains 50_{16}. After execution of

$$\text{SUB (IX+0FFH)}$$

the Accumulator will contain AF_{16}.
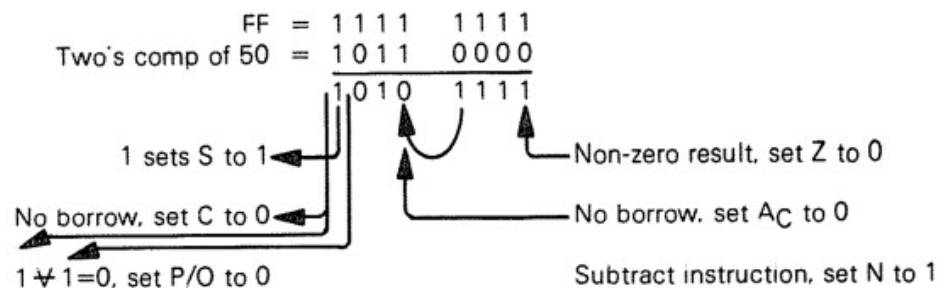
Notice that the resulting carry is complemented.

$$\underbrace{\text{SUB}}\ \underbrace{\text{(IY+disp)}}$$

$$\text{FD 96}\quad\text{d}$$

This instruction is identical to SUB (IX+disp), except that it uses the IY register instead of the IX register.

$$\underbrace{\text{SUB (HL)}}$$

$$96$$

Subtract contents of memory location (specified by the contents of the HL register pair) from the Accumulator.

XOR data — EXCLUSIVE-OR IMMEDIATE WITH ACCUMULATOR

$$\underbrace{XOR}_{EE} \quad \underbrace{data}_{yy}$$
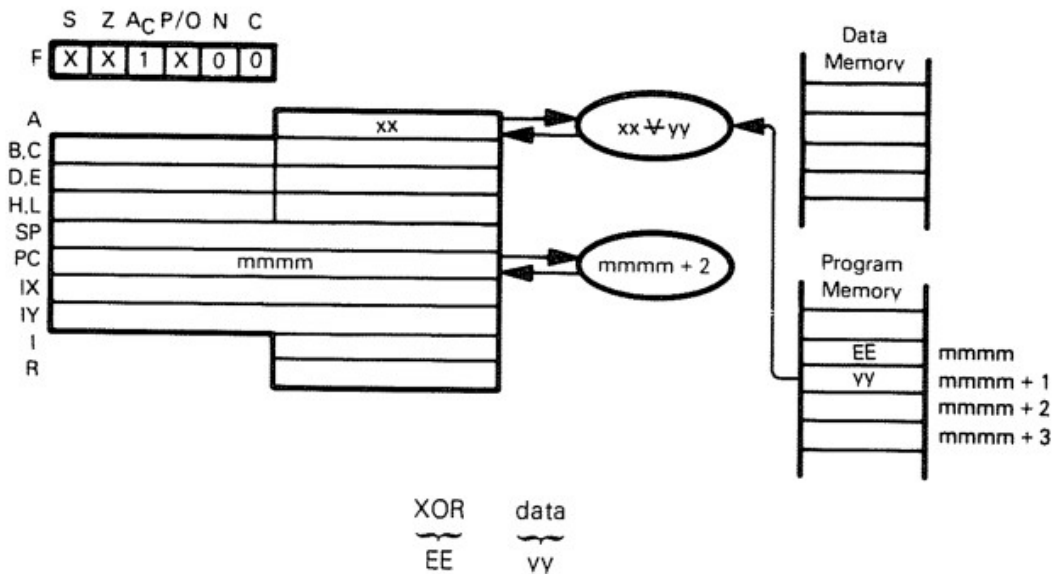
Exclusive-OR the contents of the second object code byte with the Accumulator.

Suppose $xx=3A_{16}$. After the instruction

$$XOR \; 7CH$$

has executed, the Accumulator will contain 46_{16}.

$$
\begin{array}{llll}
3A & = & 0011 & 1010 \\
7C & = & 0111 & 1100 \\
\hline
   &   & 0100 & 0110
\end{array}
$$

0 sets S to 0 ◄———

———Non-zero result, set Z to 0

———Three 1 bits, set P/O to 0

The Exclusive-OR instruction is used to test for changes in bit status.

XOR reg — EXCLUSIVE-OR REGISTER WITH ACCUMULATOR

S   Z A$_C$ P/O N   C

F | X | X | 1 | X | 0 | 0

A
B,C
D,E
H,L
SP
PC   mmmm
IX
IY
I
R

xx ∀ yy

Contents of A, B,
C, D, E, H or L
is yy

mmmm + 1

Data
Memory

Program
Memory

10101xxx | mmmm
mmmm + 1
mmmm + 2
mmmm + 3
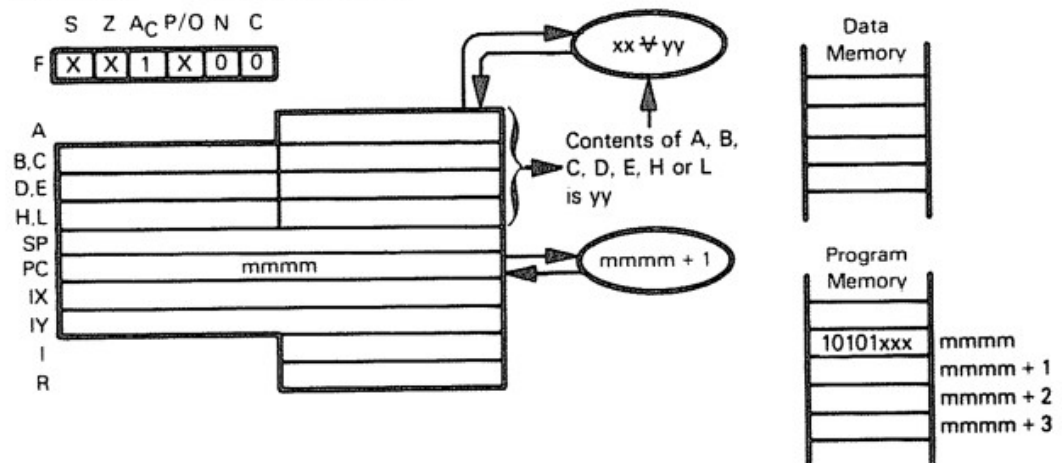
```
XOR      reg
‿‿‿      ‿‿‿
10101    xxx
         ‿‿‿
         000 for reg=B
         001 for reg=C
         010 for reg=D
         011 for reg=E
         100 for reg=H
         101 for reg=L
         111 for reg=A
```

Exclusive-OR the contents of the specified register with the Accumulator.

Suppose xx=E3$_{16}$ and Register E contains A0$_{16}$. After the instruction

XOR E

has executed, the Accumulator will contain 43$_{16}$.

```
E3 = 1110  0011
A0 = 1010  0000
     0100  0011
```

0 sets S to 0 ◄─┘

└─ Non-zero result. set Z to 0

└─ Three 1 bits. set P/O to 0

The Exclusive-OR instruction is used to test for changes in bit status.

XOR (HL) — EXCLUSIVE-OR MEMORY WITH ACCUMULATOR
XOR (IX+disp)
XOR (IY+disp)
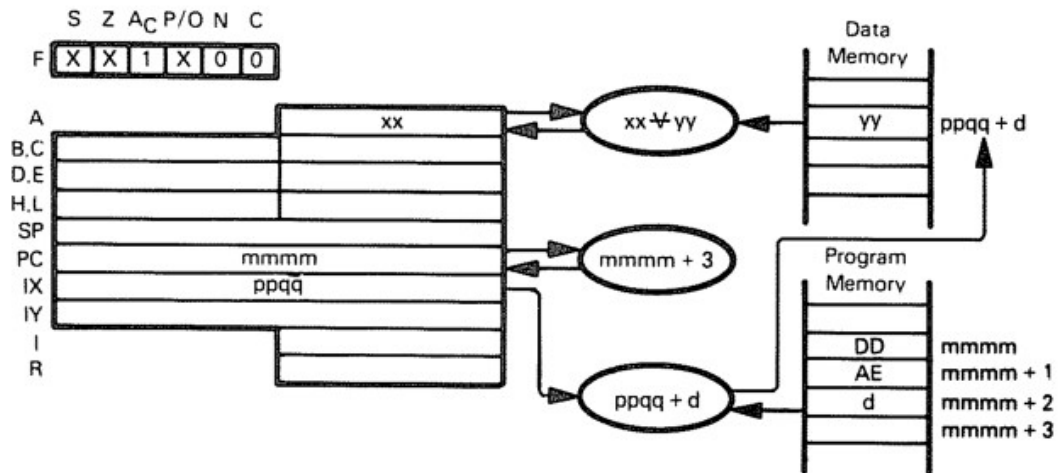
The illustration shows execution of XOR (IX+disp):

$$\underbrace{\text{XOR} \quad \text{(IX+disp)}}_{\text{DD} \quad \text{AE}} \quad \underbrace{}_{\text{d}}$$

Exclusive-OR contents of memory location (specified by the sum of the contents of the IX register and the displacement value d) with the Accumulator.

Suppose $xx=E3_{16}$, $ppqq=4500_{16}$, and memory location $45FF_{16}$ contains $A0_{16}$. After the instruction

$$\text{XOR} \quad \text{(IX+0FFH)}$$

has executed, the Accumulator will contain 43_{16}.

$$
\begin{array}{ll}
E3 = & 1110 \quad 0011 \\
A0 = & 1010 \quad 0000 \\
\hline
& 0100 \quad 0011
\end{array}
$$

0 sets S to 0 ◄────

────Non-zero result. set Z to 0

────Three 1 bits. set P/O to 0

$$\underbrace{\text{XOR} \quad \text{(IY+disp)}}_{\text{FD} \quad \text{AE}} \quad \underbrace{}_{\text{d}}$$

This instruction is identical to XOR (IX+disp), except that it uses the IY register instead of the IX register.

$$\underbrace{\text{XOR} \quad \text{(HL)}}_{\text{AE}}$$

Exclusive-OR contents of memory location (specified by the contents of the HL register pair) with the Accumulator.

8080A/Z80 COMPATIBILITY

Although the Z80 microprocessor can certainly be used on its own merits, one of its important characteristics is its compatibility with the 8080A microprocessor. This compatibility has the following features:

1) All 8080A machine language instructions are also Z80 machine language instructions.

2) All 8080A registers are also Z80 registers (see Table 3-6).

3) Almost all 8080A programs will run on a Z80, with some minor differences to be noted later.

4) The Z80 has instructions, registers, and other features not present on the 8080A, so Z80 programs will not generally run on 8080A processors.

Note that this compatibility does not extend to assembly language source statements since Z80 assemblers and 8080A assemblers use different operation code mnemonics. **Table 3-7 contains a list of the 8080A mnemonic codes and the corresponding Z80 codes, while Table 3-8 is the same list organized by Z80 codes.**

Readers should note the binary coding limitations that this compatibility places on the extra features of the Z80 microprocessor. The 8080A has some unused operation codes (see Table 3-9) that are used for some of the Z80's extra instructions. But there are simply not enough such codes to cover the large number of features in a simple form.

Thus, many of the added Z80 instructions require a 2-byte operation code. The first byte is CB, DD, ED, or FD. Note the following meanings of these codes from Table 3-9:

CB — a register or bit operation

DD — an operation involving register IX

ED — a miscellaneous non-8080A instruction not covered elsewhere

FD — an operation involving register IY

The second byte of the operation code describes the actual operation to be performed.

The end result is that these multi-byte instructions execute rather slowly (and use more memory) because an additional memory access is required. The reader should be aware of this variation in execution times and try to use faster executing instructions when possible. This warning particularly applies to the extra shift instructions (RLC, RRC, RL, RR, SRA, SRL) and to instructions involving the index registers IX and IY.

There are a few minor incompatibilities between the 8080A and the Z80. These are:

1) The Z80 uses the P (or P/O) flag to indicate twos complement overflow after arithmetic operations. The 8080A always uses this flag for parity.

2) The Z80 and 8080A execute the DAA instruction differently. On the Z80, this instruction will correct decimal subtraction as well as decimal addition. On the 8080A, it will correct only decimal addition.

3) The Z80 rotate instructions clear the A_C flag. The 8080A rotate instructions do not affect the A_C flag.

Table 3-6. Register and Flag Correspondence between Z80 and 8080A

Z80 Register	8080A Register
A	A
A'	None
B	B
B'	None
C	C
C'	None
D	D
D'	None
E	E
E'	None
F	Least Significant Half of PSW
F'	None
H	H
H'	None
I	None
IX	None
IY	None
L	L
L'	None
R	None
PC	PC
SP	SP

Z80 Register Pairs	8080A Register Pairs
BC	B
DE	D
HL	H
AF	PSW

Z80 Flags	8080A Flags
C (Carry)	C (Carry)
H (Half-Carry)	AC (Auxiliary Carry)
N (Subtract)	None
P/O (Parity/Overflow)	P (Parity)
S (Sign)	S (Sign)
Z (Zero)	Z (Zero)

The Z80 is not compatible with the extra features of the 8085 microprocessor. The codes used for RIM and SIM on the 8085 are used for relative jumps (NZ and NC) on the Z80.

8085/Z80 INCOMPATIBILITIES

Instruction timings on the 8080A, 8085, and Z80 all differ. Programs that depend on precise instruction timings will therefore execute properly only on the processor for which they were written.

TIMING INCOMPATIBILITIES

The N flag on the Z80 occupies bit 2 of the F register; the corresponding bit in the Processor Status Word of the 8080A is always a logic '1'.

Table 3-7 Correspondence between 8080A and Z80 Mnemonics

8080A Mnemonic		Z80 Mnemonic	
ACI	data	ADC	A,data
ADC	reg or M	ADC	A,reg or (HL)
ADD	reg or M	ADD	A,reg or (HL)
ADI	data	ADD	A,data
ANA	reg or M	AND	reg or (HL)
ANI	data	AND	data
CALL	addr	CALL	addr
CC	addr	CALL	C,addr
CM	addr	CALL	M,addr
CMA		CPL	
CMC		CCF	
CMP	reg or M	CP	reg or (HL)
CNC	addr	CALL	NC,addr
CNZ	addr	CALL	NZ,addr
CP	addr	CALL	P,addr
CPE	addr	CALL	PE,addr
CPI	data	CP	data
CPO	addr	CALL	PO,addr
CZ	addr	CALL	Z,addr
DAA		DAA	
DAD	rp	ADD	HL,rp
DCR	reg or M	DEC	reg or (HL)
DCX	rp	DEC	rp
DI		DI	
EI		EI	
HLT		HALT	
IN	port	IN	A,(port)
INR	reg or M	INC	reg or (HL)
INX	rp	INC	rp
JC	addr	JP	C,addr
JM	addr	JP	M,addr
JMP	addr	JP	addr
JNC	addr	JP	NC,addr
JP	addr	JP	P,addr
JNZ	addr	JP	NZ,addr
JPE	addr	JP	PE,addr
JPO	addr	JP	PO,addr
JZ	addr	JP	Z,addr
LDA	addr	LD	A,(addr)
LDAX	B or D	LD	A,(BC) or (DE)

8080A Mnemonic		Z80 Mnemonic	
LHLD	addr	LD	HL,(addr)
LXI	rp,data16	LD	rp,data16
MOV	reg,reg or M	LD	reg,reg or (HL)
MOV	reg or M,reg	LD	reg or (HL),reg
MVI	reg or M,data	LD	reg or (HL),data
NOP		NOP	
ORA	reg or M	OR	reg or (HL)
ORI	data	OR	data
OUT	port	OUT	(port),A
PCHL		JP	(HL)
POP	pr	POP	pr
PUSH	pr	PUSH	pr
RAL		RLA	
RAR		RRA	
RC		RET	C
RET		RET	
RLC		RLCA	
RM		RET	M
RNC		RET	NC
RNZ		RET	NZ
RP		RET	P
RPE		RET	PE
RPO		RET	PO
RRC		RRCA	
RST	n	RST	n
RZ		RET	Z
SBB	reg or M	SBC	A,reg or (HL)
SBI	data	SBC	A,data
SHLD	addr	LD	(addr),HL
SPHL		LD	SP,HL
STA	addr	LD	(addr),A
STAX	B or D	LD	(BC) or (DE),A
STC		SCF	
SUB	reg or M	SUB	reg or (HL)
SUI	data	SUB	data
XCHG		EX	DE,HL
XRA	reg or M	XOR	reg or (HL)
XRI	data	XOR	data
XTHL		EX	(SP),HL

Table 3-8. Correspondence between Z80 and 8080A Mnemonics

Z80 Mnemonic		8080A Mnemonic		Z80 Mnemonic		8080A Mnemonic	
ADC	A,data	ACI	data	INC	rp	INX	rp
ADC	A,(HL)	ADC	M	INC	xy	—	
ADC	A,reg	ADC	reg	INC	(xy + disp)	—	
ADC	A,(xy + disp)	—		IND		—	
ADC	HL,rp	—		INDR		—	
ADD	A,data	ADI	data	INI		—	
ADD	A,(HL)	ADD	M	INIR		—	
ADD	A,reg	ADD	reg	JP	addr	JMP	addr
ADD	A,(xy + disp)	—		JP	C,addr	JC	addr
ADD	HL,rp	DAD	rp	JP	(HL)	PCHL	
ADD	IX,pp	—		JP	M,addr	JM	addr
ADD	IY,rr	—		JP	NC,addr	JNC	addr
AND	data	ANI	data	JP	NZ,addr	JNZ	addr
AND	(HL)	ANA	M	JP	P,addr	JP	addr
AND	reg	ANA	reg	JP	PE,addr	JPE	addr
AND	(xy + disp)	—		JP	PO,addr	JPO	addr
BIT	b,(HL)	—		JP	Z,addr	JZ	addr
BIT	b,reg	—		JP	xy	—	
BIT	b,(xy + disp)	—		JR	C,disp	—	
CALL	addr	CALL	addr	JR	disp	—	
CALL	C,addr	CC	addr	JR	NC,disp	—	
CALL	M,addr	CM	addr	JR	NZ,disp	—	
CALL	NC,addr	CNC	addr	JR	Z,disp	—	
CALL	NZ,addr	CNZ	addr	LD	A,(addr)	LDA	addr
CALL	P,addr	CP	addr	LD	A,(BC) or (DE)	LDAX	B or D
CALL	PE,addr	CPE	addr	LD	A,i	—	
CALL	PO,addr	CPO	addr	LD	A,R	—	
CALL	Z,addr	CZ	addr	LD	(addr),A	STA	addr
CCF		CMC		LD	(addr),BC or DE	—	
CP	data	CPI	data	LD	(addr),HL	SHLD	addr
CP	(HL)	CMP	M	LD	(addr),SP	—	
CP	reg	CMP	reg	LD	(addr),xy	—	
CP	(xy + disp)	—		LD	(BC) or (DE),A	STAX	B or D
CPD		—		LD	BC or DE,(addr)	—	
CPDR		—		LD	HL,(addr)	LHLD	addr
CPI		—		LD	(HL),data	MVI	M,data
CPIR		—		LD	(HL),reg	MOV	M,reg
CPL		CMA		LD	I,A	—	
DAA		DAA		LD	R,A	—	
DEC	(HL)	DCR	M	LD	reg,data	MVI	reg,data
DEC	reg	DCR	reg	LD	reg,(HL)	MOV	reg,M
DEC	rp	DCX	rp	LD	reg,reg	MOV	reg,reg
DEC	xy	—		LD	reg,(xy + disp)	—	
DEC	(xy + disp)	—		LD	rp,data16	LXI	rp,data16
DI		DI		LD	SP,(addr)	—	
DJNZ	disp	—		LD	SP,HL	SPHL	
EI		EI		LD	SP,xy	—	
EX	AF,AF'	—		LD	xy,data16	—	
EX	DE,HL	XCHG		LD	xy,(addr)	—	
EX	(SP),HL	XTHL		LD	(xy + disp),data	—	
EX	(SP),xy	—		LD	(xy + disp),reg	—	
EXX		—		LDD		—	
HALT		HLT		LDDR		—	
IM	m	—		LDI		—	
IN	A,(port)	IN	port	LDIR		—	
IN	reg,(C)	—		NEG		—	
INC	(HL)	INR	M	NOP		NOP	
INC	reg	INR	reg	OR	data	ORI	data

— indicates that there is no corresponding instruction.

Table 3-8. Correspondence between Z80 and 8080A Mnemonics (Continued)

Z80 Mnemonic		8080A Mnemonic	
OR	(HL)	ORA	M
OR	reg	ORA	reg
OR	(xy + disp)	—	
OTDR		—	
OTIR		—	
OUT	(C),reg	—	
OUT	(port),A	OUT	port
OUTD		—	
OUTI		—	
POP	pr	POP	pr
POP	xv	—	
PUSH	pr	PUSH	pr
PUSH	xy	—	
RES	b,(HL)	—	
RES	b,reg	—	
RES	b,(xy + disp)	—	
RET		RET	
RET	C	RC	
RET	M	RM	
RET	NC	RNC	
RET	NZ	RNZ	
RET	P	RP	
RET	PE	RPE	
RET	PO	RPO	
RET	Z	RZ	
RETI		—	
RETN		—	
RL	(HL)	—	
RL	reg	—	
RL	(xy + disp)	—	
RLA		RAL	
RLC	(HL)	—	
RLC	reg	—	
RLC	(xy + disp)	—	
RLCA		RLC	
RLD		—	

Z80 Mnemonic		8080A Mnemonic	
RR	(HL)	—	
RR	reg	—	
RR	(xy + disp)	—	
RRA		RAR	
RRC	(HL)	—	
RRC	reg	—	
RRC	(xy + disp)	—	
RRCA		RRC	
RRD		—	
RST	n	RST	n
SBC	A,data	SBI	data
SBC	A,(HL)	SBB	M
SBC	A,reg	SBB	reg
SBC	A,(xy + disp)	—	
SBC	HL,rp	—	
SCF		STC	
SET	b,(HL)	—	
SET	b,reg	—	
SET	b,(xy + disp)	—	
SLA	(HL)	—	
SLA	reg	—	
SLA	(xy + disp)	—	
SRA	(HL)	—	
SRA	reg	—	
SRA	(xy + disp)	—	
SRL	(HL)	—	
SRL	reg	—	
SRL	(xy + disp)	—	
SUB	data	SUI	data
SUB	(HL)	SUB	M
SUB	reg	SUB	reg
SUB	(xy + disp)	—	
XOR	data	XRI	data
XOR	(HL)	XRA	M
XOR	reg	XRA	reg
XOR	(xy + disp)	—	

— indicates that there is no corresponding instruction

Table 3-9. Unused 8080A Operation Codes and Their Z80 Meanings

8080A Operation Code	Z80 Use
08	EX AF,AF'
10	DJNZ disp
18	JR disp
20 (RIM on 8085)	JR NZ,disp
28	JR Z,disp
30 (SIM on 8085)	JR NC,disp
38	JR C,disp
CB	BIT, RES, RL, RLC, RR, RRC, SET, SLA, SRA, SRL
D9	EXX
DD	All instructions involving Register IX.
ED	ADC HL,rp — LD A,I — NEG CPD — LD A,R — OTDR CPDR — LD (addr),rp — OTIR CPI — LD I,A — OUT (C),reg CPIR — LD R,A — OUTD IM m — LD rp,(addr) — OUTI IN reg,(C) — LDD — RETI IND — LDDR — RETN INDR — LDI — RLD INI — LDIR — RRD INIR — — SBC HL,rp
FD	All instructions involving Register IY.

ZILOG Z80 ASSEMBLER CONVENTIONS

The standard Z80 assembler is available from Z80 manufacturers and on the major time-sharing networks; it is also part of most development systems. Cross assembler versions are available for most large computers and many minicomputers.

ASSEMBLER FIELD STRUCTURE

The assembly language instructions have the standard field structure (see Table 2-1). The required delimiters are:

1) A colon after a label, except for the pseudo-operations EQU, DEFL, and MACRO, which require a space.

2) A space after the operation code.

3) A comma between operands in the operand field. (Remember this one!)

4) A semicolon before a comment.

5) Parentheses around memory references.

Typical Z80 assembly language instructions are:

```
START:  LD      A,(1000)    ;GET LENGTH
        ADD     HL,DE
        HALT
```

LABELS

The assembler allows six characters in labels; the first character must be a letter, while subsequent characters must be letters, numbers, ?, or the underbar character (_). We will use only capital letters or numbers, although some versions of the assembler allow lower-case letters and other symbols.

RESERVED NAMES

Some names are reserved as keywords and should not be used by the programmer. These are the register names (A, B, C, D, E, H, L, I, R), the double register names (IX, IY, SP), the register names (AF, BC, DE, HL, AF', BC', DE', HL'), and the states of the four testable flags (C, NC, Z, NZ, M, P, PE, PO).

PSEUDO-OPERATIONS

The assembler has the following basic pseudo-operations:

DEFB	-	DEFINE BYTE
DEFL	-	DEFINE LABEL
DEFM	-	DEFINE STRING
DEFS	-	DEFINE STORAGE
DEFW	-	DEFINE WORD
END	-	END
EQU	-	EQUATE
ORG	-	ORIGIN

DEFB, DEFM, and DEFW are the Data pseudo-operations used to place data in ROM. DEFB is used for 8-bit data, DEFW for 16-bit data, and DEFM for ASCII strings (63 or less characters long). The only unusual feature to remember is that DEFW stores the eight least significant bits of data in the first word and the eight most significant bits in the second word. This is the standard 8080A/8085/Z80 procedure for storing addresses in memory, but is contrary to normal practice. You must be aware of the order when storing 16-bit data.

> DEFB,DEFM,
> DEFW
> PSEUDO-
> OPERATIONS

Note that DEFB and DEFW define the value of only a single byte or single word, respectively. Establishing a table of values requires a series of DEFB or DEFW pseudo-operations. one for each byte or word of data.

Examples:

ADDR: DEFW   3165H

results in (ADDR) = 65, and (ADDR+1) = 31 (hexadecimal).

TCONV: DEFB   32

This pseudo-operation places the number 32 in the next byte of ROM and assigns the name TCONV to the address of that byte.

ERROR: DEFM   'ERROR'

This pseudo-operation places the 7-bit ASCII characters E. R. R. O. and R in the next five bytes of ROM and assigns the name ERROR to the address of the first byte.

OPERS: DEFW   FADD
         DEFW   FSUB
         DEFW   FMUL
         DEFW   FDIV

This series of pseudo-operations places the addresses FADD. FSUB. FMUL. and FDIV in the next eight bytes of memory and assigns the name OPERS to the address of the first byte. Note that the first byte contains the least significant bits of address FADD.

DEFS is the Reserve pseudo-operation used to assign locations in RAM: it allocates a specified number of bytes.

> **DEFS PSEUDO-OPERATION**

EQU is the Equate or Define pseudo-operation used to assign values to names.

> **EQU PSEUDO-OPERATION**

DEFL is similar to EQU, except that DEFL allows the name to be redefined later. DEFL is much like the SET directive in other assemblers. It should only be used to define assembly time variables (i.e., those variables used in conditional assembly or conditional macro expansion statements).

> **DEFL PSEUDO-OPERATION**

ORG is the standard Origin pseudo-operation.

Z80 programs usually have several origins; the origins are used as follows:

> **ORG PSEUDO-OPERATION**

1)   To specify the RESET address (usually zero).
2)   To specify interrupt entry points (usually 0 to 66_{16} but may be anywhere in memory.
3)   To specify the starting address of the main program.
4)   To specify the starting addresses of subroutines.
5)   To define areas for RAM storage.
6)   To define an area for the RAM Stack.
7)   To specify addresses used for I/O ports and special functions.

Examples:

```
RESET    EQU    0
         ORG    RESET
```

This sequence places the RESET instruction sequence in memory beginning at address 0.

```
INT1     EQU    38H
         ORG    INT1
```

The instruction sequence that follows is stored in memory beginning at location 38_{16}.

END simply marks the end of the assembly language program.

The special purpose pseudo-operations COND. MACRO, ENDC, and ENDM are described later in this chapter.

LABELS WITH PSEUDO-OPERATIONS

The rules and recommendations for labels with Z80 pseudo-operations are as follows:

1) EQU, DEFL, and MACRO require labels, since the function of these pseudo-operations is to define the meaning of that label.

2) DEFB, DEFM. DEFW, and DEFS usually have labels.

3) ORG, COND, ENDC. ENDM, and END should not have labels, since the meaning of such labels is unclear.

ADDRESSES

The Zilog Z80 assembler allows entries in the address field in any of the following forms:

1) Decimal (the default case)
   Example: 1247

2) Hexadecimal (must start with a digit and end with an H)
   Examples: 142CH. 0E7H

3) Octal (must end with O or Q, but Q is far less confusing)
   Example: 1247Q or 1247O

4) Binary (must end with B)
   Example: 1001001000111B

5) ASCII (enclosed in single quotation marks)
   Example: 'HERE'

6) As an offset from the Program Counter ($)
   Example: $+237H

All arithmetic and logic operations within an address field assume all arguments are 16-bit data; they produce 16-bit results. These operations are allowed as part of expressions in the address field.

When defining address constants, hexadecimal notation should be used. Binary constants of 16 bits are unwieldy and hence error-prone. Octal constants are inconvenient due to the fact that addresses are stored in low-order byte high-order byte format. This division occurs in the middle of an octal digit, which causes you to have to split a digit. For example, to express the address 9D7FH or 116577Q in low-high format you get 7F9DH or 77236Q. As you can see, in hexadecimal notation the digits are simply transposed, while no such simple relationship exists for octal notation.

OPERATOR	FUNCTION	PRIORITY
+	UNARY PLUS	1
-	UNARY MINUS	1
.NOT. or \	LOGICAL NOT	1
.RES.	RESULT	1
••	EXPONENTIATION	2
•	MULTIPLICATION	3
/	DIVISION	3
.MOD.	MODULO	3
.SHR.	LOGICAL SHIFT RIGHT	3
.SHL.	LOGICAL SHIFT LEFT	3
+	ADDITION	4
-	SUBTRACTION	4
.AND. or &	LOGICAL AND	5
.OR. or ↑	LOGICAL OR	6
.XOR.	LOGICAL XOR	6
.EQ. or =	EQUALS	7
.GT. or >	GREATER THAN	7
.LT. or <	LESS THAN	7
.UGT.	UNSIGNED GREATER THAN	7
.ULT.	UNSIGNED LESS THAN	7

In address expressions with more than one operator, the order of evaluation is defined by the priorities given in the list above. Operators having the same priorities are evaluated from left to right. Expressions in parentheses are evaluated first. Remember that enclosing an expression entirely in parentheses indicates a memory address.

Note the following:

1) The Result operator (.RES.) causes overflow to be suppressed; i.e., a change in sign caused by overflow into the sign bit does not result in an assembler error.

2) The shifts have the form:

        .SHR.   op1.op2
        .SHL.   op1.op2

where op1 is the number to be shifted and op2 is the number of shifts. The shifts are logical, i.e., zeros are shifted into the high-order or low-order bits, respectively.

3) The comparison operators produce a result of either logical True (all ones) or logical False (zero).

4) The operators .GT. and .LT. assume signed twos complement numbers, whereas .UGT. and .ULT. assume unsigned operands. This means that, for .GT. and .LT., positive twos complement numbers are larger than negative twos complement numbers, while the opposite is the case for .UGT. and .ULT.

CONDITIONAL ASSEMBLY

The Z80 assembler has a simple conditional assembly capability based on the pseudo-operations COND and ENDC. COND is followed by an expression, for example:

$$\text{COND} \quad \text{BASE - 1000H}$$
or
$$\text{COND} \quad \text{BASE - OPER1}$$

If the expression is not zero, the assembler includes all of the instructions up to the ENDC pseudo-operation in the program; if the expression is zero, the assembler ignores all instructions between COND and ENDC.

We will not use conditional assemblies or refer to this capability again; it is sometimes handy for adding or eliminating debugging instructions, or configuring unique versions of a common program.

MACROS

The standard Z80 assembler has a macro capability that assigns names to instruction sequences. Use the pseudo-operation MACRO to begin the definition and ENDM to end it. The macro may have parameters and may include any assembly language instructions except the definitions of other macros.

The macro capability is often a convenient programming shorthand, but we will not use it.

Note that instruction sequences defined by macros are generally quite short; they should not exceed ten or fifteen instructions. Longer sequences should be made into subroutines to conserve memory space.

Every MACRO pseudo-operation must have a label; the label is the name with which you identify the macro. For a discussion of this subject, see Chapter 2.

Chapter 4
SIMPLE PROGRAMS

The only way to learn assembly language programming is to write assembly language programs. That is what we will do for the next six chapters, which contain examples of typical microprocessor tasks. Problems at the end of each chapter contain variations on the examples given in the text of the chapter. You should try to run the examples on a Z80-based microcomputer system to ensure that you understand the material covered in the chapter.

In this chapter we begin with some very simple programs.

GENERAL FORMAT OF EXAMPLES

Each program example contains the following parts:

1) A title that describes the general problem.

2) A statement of purpose which describes the specific task that the program performs, plus the memory locations that it uses.

3) A sample problem showing input data and results.

4) A flowchart if the program logic is complex.

5) The source program or assembly language listing of the program.

6) The object program or hexadecimal machine language listing of the program.

7) Explanatory notes that discuss the instructions and methods used in the program.

The problems at the end of the chapter are similar to the examples; problems should be programmed on a Z80-based microcomputer system using the examples as guidelines.

The source programs in the examples have been constructed as follows:

1) Standard Zilog Z80 assembler notation is used, as summarized in Chapter 3.

2) The forms in which data and addresses appear are selected for clarity rather than for consistency. We use hexadecimal numbers for memory addresses, instruction codes, and BCD data; decimal for numeric constants; binary for logical masks; and ASCII for characters.

3) Frequently used instructions and programming techniques are emphasized.

4) Examples illustrate tasks that microprocessors perform in communications, instrumentation, computer, business equipment, industrial, and military applications.

5) Detailed comments are included.

6) Simple and clear structures are emphasized, but programs are as efficient as possible within this guideline. The notes often describe more efficient procedures.

7) Programs use consistent memory allocations. Each program starts in memory location 0000 (the RESET location) and ends with the HALT instruction. If your

microcomputer has no monitor and no interrupts, you may prefer to end programs with an endless loop instruction, e.g.:

```
HERE:    JR       HERE
```

The hexadecimal version is 18 followed by FE. You may replace the HALT or JR HERE instruction with a RESTART or JP instruction that transfers control back to the monitor in some Z80-based microcomputers.

Consult the user's manual for your microcomputer to determine the required memory allocations and terminating instruction for your particular system.

GUIDELINES FOR PROBLEMS

When tackling the problems at the end of each chapter, try to work within the following guidelines:

<div style="float:right; border:1px solid">PROGRAMMING GUIDELINES</div>

1) Comment each program so that others can understand it.
   The comments can be brief and ungrammatical; they should explain the purpose of a section or instruction in the program. Comments should not describe the operation of instructions; that description is available in manuals. You do not have to comment each statement or explain the obvious. You may follow the format of the examples but provide less detail.

2) Emphasize clarity, simplicity, and good structure in programs. While programs should be reasonably efficient, do not worry about saving a single byte of program memory or a few microseconds.

3) Make programs reasonably general. Do not confuse parameters (such as the number of elements in an array) with fixed constants (such as π or ASCII C).

4) Never assume fixed initial values for parameters, i.e., use an instruction to load an initial value into a parameter.

5) Use assembler notation as shown in the examples and defined in Chapter 3.

6) Use hexadecimal notation for addresses. Use the clearest possible form for data.

7) If your microcomputer allows it, start all programs in memory location 0000 and use memory locations starting with 0040_{16} for data and temporary storage. Otherwise, establish equivalent addresses for your microcomputer and use them consistently. Again, consult the user's manual.

8) Use meaningful names for labels and variables, e.g., SUM or CHECK rather than X, Y, or Z.

9) Execute each program on your microcomputer. There is no other way of ensuring that your program is correct. We have provided sample data with each problem. Be sure that the program works for special cases.

We now summarize some useful information that you should keep in mind when writing programs.

Almost all processing instructions (e.g., ADD, SUBTRACT, AND, OR) use the Accumulator. In most cases you will load data into the Accumulator with LD, using either LD A,(addr) to load data from any memory location or using LD A,(HL) to load data from the address specified in Registers H and L. Remember that the parentheses indicate a memory address rather than data.

<div style="float:right; border:1px solid">USING THE ACCUMULATOR</div>

The preferred method of accessing memory is using implied addressing via Registers H and L, that is, using (HL). This code causes the Z80 to perform a memory access using the address stored in Registers H and L. You can use LD HL, data16 to load a fixed num-

<div style="float:right; border:1px solid">USING REGISTER PAIR HL</div>

ber into Registers H and L or LD HL,(addr) to load the contents of two successive memory locations into H and L. You can use INC HL or DEC HL to increment or decrement (by 1) the address in Registers H and L.

The 8-bit arithmetic and logical operations all use the data in the Accumulator as one of their operands and place their result into the Accumulator.

Some of the 8-bit arithmetic and logical operations have special uses, for example:

```
SPECIAL
INSTRUCTIONS
```

SUB A (or XOR A) clears the Accumulator.

ADD A,A shifts the Accumulator left one bit logically. This instruction also multiplies the contents of the Accumulator by 2. AND A (or OR A) clears the Carry flag while preserving the contents of the Accumulator.

A logical AND can mask off parts of a word. The required mask has '1' bits in the positions that you want to reserve and '0' bits in the positions that you want to clear.

PROGRAM EXAMPLES

Ones Complement

Purpose: Logically complement the contents of memory location 0040 and place the result into memory location 0041.

Sample Problem:

$$(0040) = 6A$$

Result: $(0041) = 95$

Source Program:

```
LD      A,(40H)     :GET DATA
CPL                 :COMPLEMENT
LD      (41H),A     :STORE RESULT
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3A	LD	A,(40H)
0001	40		
0002	00		
0003	2F	CPL	
0004	32	LD	(41H),A
0005	41		
0006	00		
0007	76	HALT	

The LD A,(addr) and LD (addr),A instructions contain addresses to determine the source or destination of the data. The addresses are 16 bits long, with the eight least significant bits in the word immediately following the instruction code and the eight most significant bits in the next word (this order is contrary to normal computer practice). CPL is a one-word instruction that inverts each bit of the Accumulator. It replaces each '0' with a '1' and each '1' with a '0', just like a set of inverter gates.

HALT is used to end all the examples.

Note that we could also place an address into Registers H and L and then use that address throughout the program. This is shown in the following program.

Source Program:

```
LD    HL,40H    ;POINT TO OPERAND
LD    A,(HL)    ;GET DATA
CPL             ;COMPLEMENT
INC   HL        ;POINT TO DESTINATION
LD    (HL),A    ;STORE RESULT
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	2F	CPL	
0005	23	INC	HL
0006	77	LD	(HL),A
0007	76	HALT	

Which version do you think is better?

The two versions require the same number of bytes of memory even though the second version is two instructions longer. This is because the second version uses fewer explicit addresses.

8-Bit Addition

Purpose: Add the contents of memory locations 0040 and 0041, and place the result into memory location 0042.

Sample Problem:

```
(0040)  =  38
(0041)  =  2B
```

Result: (0042) = 63

Source Program:

```
LD    A,(40H)    ;GET FIRST OPERAND
LD    B,A        ;SAVE FIRST OPERAND
LD    A,(41H)    ;GET SECOND OPERAND
ADD   A,B        ;ADD OPERANDS
LD    (42H),A    ;STORE SUM
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3A	LD	A,(40H)
0001	40		
0002	00		
0003	47	LD	B,A
0004	3A	LD	A,(41H)
0005	41		
0006	40		
0007	80	ADD	A,B
0008	32	LD	(42H),A
0009	42		
000A	00		
000B	76	HALT	

Here again, we could alternatively use Registers H and L as the source for all addresses.

Source Program:

```
LD      HL,40H
LD      A,(HL)      ;GET FIRST OPERAND
INC     HL
ADD     A,(HL)      ;ADD SECOND OPERAND
INC     HL
LD      (HL),A      ;STORE RESULT
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	23	INC	HL
0005	86	ADD	A,(HL)
0006	23	INC	HL
0007	77	LD	(HL),A
0008	76	HALT	

In this case, the program using Registers H and L is shorter than the one using direct addressing. Why?

LD HL,40H loads the contents of the following two words of program memory into Register Pair HL. The first word goes into Register L, the second into Register H.

The code (HL) means that data is obtained from or sent to the memory location addressed by Registers H and L. Thus, LD A,(HL) loads the Accumulator with the contents of the addressed memory location; LD (HL),A loads the addressed memory location with the contents of the Accumulator. ADD A,(HL) adds the contents of the location addressed by HL to the contents of the Accumulator. Remember that H and L contain a 16-bit address, but the memory location with that address contains eight bits of data. Note the difference between ADD A,(HL) and ADD A,H or ADD A,L.

INC HL performs a 16-bit increment in one instruction cycle. The CPU doesn't use the 8-bit arithmetic unit for the increment; it uses the incrementer that it normally uses to increment the 16-bit Program Counter.

LD A,(HL) and LD (HL),A are preferable to LD A,(addr) and LD (addr),A whenever you use the same memory location repeatedly or use adjacent locations, because LD A,(HL) and LD (HL),A require less program memory and time. Note, however, that you must load Registers H and L before you can use (HL).

Shift Left One Bit

Purpose: Shift the contents of memory location 0040 left one bit and place the result into memory location 0041. Clear the empty bit position. This type of shift is known as a logical shift. In a logical shift, a value of zero is always shifted in.

Sample Problem:

$$(0040) = 6F$$

Result: $(0041) = DE$

Source Program:

```
LD      A,(40H)      :GET DATA
ADD     A,A          :SHIFT LEFT
LD      (41H),A      :STORE RESULT
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3A	LD	A,(40H)
0001	40		
0002	00		
0003	87	ADD	A,A
0004	32	LD	(41H),A
0005	41		
0006	00		
0007	76	HALT	

ADD A,A simply adds the contents of the Accumulator to itself. The result, of course, is twice the original data, which is the same result that a logical left shift would produce. The least significant bit of the result is zero, since $0+0 = 1+1 = 0$; $1+1$ also produces a Carry to the next bit.

Alternatively, we could replace ADD A,A with SLA A, certainly the more obvious choice. However, SLA A requires two words of program memory and eight clock cycles, while ADD A,A requires one word of program memory and four clock cycles. The difference is caused by the fact that SLA A is one of the extra instructions added to the original 8080A set (remember the comparison presented earlier).

Mask Off Most Significant Four Bits

Purpose: Place the least significant four bits of memory location 0040 into the least significant four bits of memory location 0041. Clear the most significant four bits of memory location 0041.

Sample Problem:

$$(0040) = 3D$$

Result: $(0041) = 0D$

Source Program:

```
LD      A,(40H)      ;GET DATA
AND     00001111B    ;MASK 4 LSB'S
LD      (41H),A      ;STORE RESULT
HALT
```

Note: B means binary in standard Z80 assembler notation.

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3A	LD	A,(40H)
0001	40		
0002	00		
0003	E6	AND	00001111B
0004	0F		
0005	32	LD	(41H),A
0006	41		
0007	00		
0008	76	HALT	

The mask (00001111) is written in binary to make its function clearer to the reader. Binary notation for masks is generally much clearer than hexadecimal notation, although the results are the same. Hexadecimal notation should be used for masks longer than four bits. The comments should explain the masking operation.

When the argument in the address field is a number, AND logically ANDs the contents of the Accumulator with the contents of the word of program memory immediately following the instruction. AND may be used to clear bits that are not in use. The four least significant bits could be an input from a switch or an output to a numeric display.

Clear a Memory Location

Purpose: Clear memory location 0040.

Source Program:

```
SUB     A
LD      (40H),A      ;CLEAR LOCATION 40
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	97	SUB	A
0001	32	LD	(40H),A
0002	40		
0003	00		
0004	76	HALT	

SUB A subtracts the number in the Accumulator from itself. The result is to clear the Accumulator. SUB A, XOR A, or LD A,0 can all clear the Accumulator. LD A,0 takes more time and memory but doesn't affect the status flags.

Word Disassembly

Purpose: Divide the contents of memory location 0040 into two 4-bit sections and store them in memory locations 0041 and 0042. Place the four most significant bits of memory location 0040 into the four least significant bit positions

of memory location 0041; place the four least significant bits of memory location 0040 into the four least significant bit positions of memory location 0042. Clear the four most significant bit positions of memory locations 0041 and 0042.

Sample Problem:

$$(0040) = 3F$$

Result: $(0041) = 03$
$(0042) = 0F$

Source Program:

```
LD      HL,40H
LD      A,(HL)          ;GET DATA
LD      B,A
RRA                     ;SHIFT DATA RIGHT 4 TIMES
RRA
RRA
RRA
AND     00001111B       :MASK OFF MSB'S
INC     HL
LD      (HL),A          ;STORE MSB'S
LD      A,B             ;RESTORE ORIGINAL DATA
AND     00001111B       :MASK OFF LSB'S
INC     HL
LD      (HL),A          ;STORE LSB'S
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	47	LD	B,A
0005	1F	RRA	
0006	1F	RRA	
0007	1F	RRA	
0008	1F	RRA	
0009	E6	AND	00001111B
000A	0F		
000B	23	INC	HL
000C	77	LD	(HL),A
000D	78	LD	A,B
000E	E6	AND	00001111B
000F	0F		
0010	23	INC	HL
0011	77	LD	(HL),A
0012	76	HALT	

Instructions using the address in Registers H and L occupy only one word of program memory. However, HL must be loaded before the address can be used. Thus, implied memory addressing saves time and memory, as compared to direct memory addressing, only when the program repeatedly uses the same address or consecutive addresses.

RRC shifts the Accumulator right one bit circular, with the least significant bit going to the most significant bit position and to the Carry. Shifting the Accumulator right four times requires four RRCs. We could use SRL A to provide a logical shift directly (no final AND would then be necessary). However, SRL A requires twice as much time and memory as RRC. Try substituting SRL A for RRC and see the difference. Another alternative would be to use the RLD instruction to replace both the mask and the store. However, this solution is not optimal in terms of either storage or execution speed due to the constraint that the high-order nibble of each result must equal zero.

Many Z80 instructions affect a pair of 8-bit registers. The pairs are HL (H and L), DE (D and E), and BC (B and C). Registers B, D, and H are the most significant eight bits of the pairs; Registers C, E, and L are the least significant eight bits. The common instructions that use pairs of registers are LD rp (Load Register Pair), INC rp (Increment Register Pair), DEC rp (Decrement Register Pair), and ADD HL,rp (Add Register Pair to H and L).

Find Larger of Two Numbers

Purpose: Place the larger of the contents of memory locations 0040 and 0041 into memory location 0042. Assume that the contents of memory locations 0040 and 0041 are unsigned binary numbers.

Sample Problems:

a.
        (0040) = 3F
        (0041) = 2B

    Result: (0042) = 3F

b.
        (0040) = 75
        (0041) = A8

    Result: (0042) = A8

Source Program:

```
        LD      HL.40H
        LD      A.(HL)      ;GET FIRST OPERAND
        INC     HL
        CP      (HL)        ;IS SECOND OPERAND LARGER?
        JR      C.DONE
        LD      A.(HL)      ;YES. GET SECOND OPERAND INSTEAD
DONE:   INC     HL
        LD      (HL).A      ;STORE LARGER OPERAND
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	21		LD	HL,40H
0001	40			
0002	00			
0003	7E		LD	A.(HL)
0004	23		INC	HL
0005	BE		CP	(HL)
0006	30		JR	NC.DONE
0007	01			
0008	7E		LD	A.(HL)
0009	23	DONE:	INC	HL
000A	77		LD	(HL).A
000B	76		HALT	

4-9

CP (HL) sets the flags as if the contents of the memory location addressed by H and L had been subtracted from the contents of the Accumulator. However, the contents of the Accumulator are left unchanged for later comparisons or other processing.

If A is the contents of the Accumulator and X is the second operand for a CP instruction, then the flags are set as follows:

1) Zero = 1 if A = X
   Zero = 0 if A ≠ X
2) Carry = 1 if A < X
   Carry = 0 if A ≥ X
   (A, X are unsigned binary numbers)

CP sets the Carry to 1 if a borrow would be necessary to actually perform the subtraction. i.e., if the number being subtracted from the contents of the Accumulator is greater than those contents. Thus, the sequence CP, JR NC,DONE causes a jump to DONE if the contents of the Accumulator are greater than or equal to the other number.

JR NC,DONE causes a jump to memory location DONE if the Carry flag = 0. Otherwise (if Carry = 1), the computer continues with the next sequential memory location after the JR instruction.

DONE is a label, a name which you assign to a location in memory so that it is easier to remember. Note that labels are followed by a colon on the line where they are defined.

The label makes the destination of the branch clearer, particularly when relative addressing is being used. The assembler calculates the required offset (caution: some Z80 assemblers will not do this). Using a label is preferable to just specifying the offset (i.e., JR NC,$+3) since the Z80's instructions vary in length. You could therefore easily make an error in determining an offset.

If the branch conditions are not satisfied, the processor simply proceeds to the next sequential location in program memory (i.e., it executes the instruction LD A,(HL)).

The Z80 assemblers allow six characters in labels — the first must be a letter, while the others may be letters or numbers (some special characters are allowed but we will not use them).

The JR instruction uses relative addressing in which the second word of the instruction is an 8-bit twos complement number that the CPU adds to the address of the next instruction to find the target address. In the example, the relative offset is 0009 (target address) minus 0008 (address immediately following the branch) or 01.

We should note that some Z80 assemblers will not calculate the offset in the form shown. These assemblers require an offset in the address field, rather than the label of the target instruction. If you have such an assembler, use the form JR NC,DONE-$. Remember that $ means "the address of the current instruction".

The Z80 has two sets of jump instructions, JP (Jump) and JR (Jump Relative). The JP instructions require a complete memory address; they occupy three bytes of memory and execute in ten clock cycles. The JR instructions require only a one-word offset; they occupy two bytes of memory and execute in 12 cycles if a jump is actually performed and in 7 if not. So the JR instructions use less memory than JP instructions but may require a little extra time if a jump is performed (the extra time is used to execute the required 16-bit addition of program counter and offset).

16-Bit Addition

Purpose: Add the 16-bit number in memory locations 0040 and 0041 to the 16-bit number in memory locations 0042 and 0043. The most significant eight bits are in memory locations 0041 and 0043. Store the result in memory locations 0044 and 0045, with the most significant bits in 0045.

Sample Problem:

```
(0040)  =  2A
(0041)  =  67
(0042)  =  F8
(0043)  =  14
```

Result: $672A + 14F8 = 7C22$
```
(0044)  =  22
(0045)  =  7C
```

Source Program:

```
LD      HL,(40H)     ;GET FIRST 16-BIT NUMBER
LD      DE,(42H)     ;GET SECOND 16-BIT NUMBER
ADD     HL,DE        ;16-BIT ADDITION
LD      (44H),HL     ;STORE 16-BIT RESULT
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	2A	LD	HL,(40H)
0001	40		
0002	00		
0003	ED	LD ·	DE,(42H)
0004	5B		
0005	42		
0006	00		
0007	19	ADD	HL,DE
0008	22	LD	(44H),HL
0009	44		
000A	00		
000B	76	HALT	

LD HL,(addr) loads Registers H and L from two memory locations, the one specified in the instruction and the next consecutive one. The contents of the first addressed location go to Register L. The contents of the next location go to Register H. Thus, LD HL,(40H) means L = (40), H = (41). The actual transfer proceeds one byte at a time and takes 16 clock cycles. The advantage of the 16-bit Load instruction over two 8-bit Load instructions is that the CPU has to fetch only one instruction from memory.

Note the difference between LD HL,(addr), which loads the contents of the two RAM locations at addr and addr+1 into H and L, and LD HL,data16, which loads the contents of the next two bytes pointed to by the instruction counter into H and L. Since these two bytes immediately follow the op-code, loads of this type are referred to as load immediate instructions.

LD DE,(addr) is similar to LD HL,(addr) except that it takes one extra word of memory and four more clock cycles. This is one of the instructions that is present in the Z80 set but not in the 8080/8085 sets. An alternative approach is:

```
EX      DE,HL        ;SAVE FIRST 16-BIT NUMBER IN DE
LD      HL,(42H)     ;GET SECOND 16-BIT NUMBER
```

EX DE,HL exchanges the contents of Registers D and E with H and L. No numbers are changed or destroyed. The advantage of EX DE,HL will become obvious if you try to replace it with a series of LD instructions.

ADD HL,DE adds the 16-bit number in Registers D and E to the 16-bit number in Registers H and L. The result is placed into Registers H and L. ADD HL,DE actually adds one byte at a time. It executes in 11 clock cycles.

LD (addr),HL stores the contents of Registers H and L into two memory locations, the one specified in the instruction and the next consecutive one. The contents of L go into the specified location and the contents of H go into the next location. Thus, LD (44H),HL means (44) = L, (45) = H. As with LD HL,(addr), the actual transfer proceeds one byte at a time and requires 16 clock cycles.

Although the Z80 is an 8-bit processor, it has instructions that handle 16-bit numbers. These instructions are intended primarily for handling addresses, but you can also use them for 16-bit data. The most common ones and their uses are:

1) ADD HL,rp — 16-Bit Add
    Used to access tables and to add 16-bit data units
2) DEC rp — 16-Bit Decrement
    Used to subtract one from the contents of a register pair
3) INC rp — 16-Bit Increment
    Used to add one to the contents of a register pair
4) LD rp,data16 — 16-Bit Load Immediate
    Used to initialize a register pair with a fixed value, e.g., the starting address of an array or table
5) LD HL,(addr) — 16-Bit Load HL Direct
    Used to place variable addresses into the main address register (H and L)
6) LD (addr),HL — 16-Bit Store HL Direct
    Used to store addresses to memory from the main address register (H and L).

Table of Squares

Purpose: Calculate the square of the contents of memory location 0040 from a table and place it into memory location 0041. Assume that memory location 0040 contains a number between 0 and 7 inclusive ($0 \leq (0041) \leq 7$).

The table occupies memory locations 0050 to 0057.

Memory Address (Hex)	Entry	
	(Hex)	(Decimal)
0050	00	0 (0^2)
0051	01	1 (1^2)
0052	04	4 (2^2)
0053	09	9 (3^2)
0054	10	16 (4^2)
0055	19	25 (5^2)
0056	24	36 (6^2)
0057	31	49 (7^2)

Sample Problems:

a.   (0041) = 03

         Result: (0042) = 09

b.   (0041) = 06

         Result: (0042) = 24

Source Program:

```
        LD      A,(40H)     ;GET DATA
        LD      L,A         ;MAKE DATA INTO 16-BIT INDEX
        LD      H,0
        LD      DE,SQTAB    ;GET STARTING ADDRESS OF TABLE
        ADD     HL,DE       ;INDEX TABLE WITH DATA
        LD      A,(HL)      ;GET SQUARE OF DATA
        LD      (41H),A
        HALT

        ORG     50H         ;SQUARE TABLE
SQTAB:  DEFB    0
        DEFB    1
        DEFB    4
        DEFB    9
        DEFB    16
        DEFB    25
        DEFB    36
        DEFB    49
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	3A		LD	A,(40H)
0001	40			
0002	00			
0003	6F		LD	L,A
0004	26		LD	H,0
0005	00			
0006	11		LD	DE,SQTAB
0007	50			
0008	00			
0009	19		ADD	HL,DE
000A	7E		LD	A,(HL)
000B	32		LD	(41H),A
000C	41			
000D	00			
000E	76			
0050	00	SQTAB:	DEFB	0
0051	01		DEFB	1
0052	04		DEFB	4
0053	09		DEFB	9
0054	10		DEFB	16
0055	19		DEFB	25
0056	24		DEFB	36
0057	31		DEFB	49

Note that you must also enter the table of squares into memory (the assembler pseudo-operation DEFB will handle this). The table of squares is constant data, not parameters that may change; that is why you can initialize the table using the DEFB pseudo-operation, rather than by executing instructions to load values into the table. Remember that the table is part of the program memory (ROM in most systems).

LD L,A moves the data in the Accumulator to Register L. The data is the eight least significant bits of the index. You cannot always assume that the data presented to your

program is in the proper range. It is always a good practice to range check all critical values. Range checking consists of testing a value to ensure that it is within the proper lower and upper limits. Any byte can have a value in the range 0 to 255. If the value stored in the byte at location 0040H is greater than seven, the program will reference an undefined byte beyond the end of the square table, causing the program to generate erroneous results Range checking will prevent this error from occurring.

LD H,0 clears Register H so that it does not interfere with the 16-bit addition of starting address and index. Never assume that a register contains zero at the start of a program.

LD DE,SQTAB loads the starting address of the table into Registers D and E. We use D and E for the starting address since the ADD HL instruction does not change D and E. Thus, the starting address of the table will still be in D and E after the addition, in the event that we want another element from the table.

ADD HL,DE adds the starting address and the index; the result in H and L is thus the address of the correct entry. LD A,(HL) then moves that entry to the Accumulator.

Arithmetic that a microprocessor cannot do directly in a few instructions is often best performed with lookup tables. Lookup tables simply contain all the possible answers to the problem; they are organized so that the answer to a particular problem can be found easily. The arithmetic problem now becomes an accessing problem — how do we get the correct answer from the table? We must know two things: the position of the answer in the table (called the index) and the base, or starting, address of the table. The address of the answer is then the base address plus the index.

The base address, of course, is a fixed number for a particular table. How can we determine the index? In simple cases, where a single piece of data is involved, we can organize the table so that the data is the index. In the table of squares, the 0th entry in the table contains zero squared, the first entry one squared, etc. In more complex cases, where the spread of input values is very large or there are several data items involved (e.g., roots of a quadratic or number of permutations), we must use more complicated methods to determine indexes.

The basic tradeoff in using a table is time vs. memory. Tables are faster, since no computations are required, and simpler, since no mathematical methods must be devised and tested. However, tables can occupy a large amount of memory if the range of the input data is large. We can often reduce the size of a table by limiting the accuracy of the results, scaling the input data, or organizing the table cleverly. Tables are often used to compute transcendental and trigonometric functions, linearize inputs, convert codes, and perform other mathematical tasks.

16-Bit Ones Complement

Purpose: Place the ones complement of the 16-bit number in memory locations 0040 and 0041 into memory locations 0042 and 0043. The most significant bytes are in locations 0041 and 0043.

Sample Problem:

$$
\begin{array}{rll}
& (0040) & = & 67 \\
& (0041) & = & E2 \\
\text{Result:} & (0042) & = & 98 \\
& (0043) & = & 1D
\end{array}
$$

The ones complement inverts each bit of the original number; the sum of the original number and its ones complement will always be all 1 bits.

Source Program:

```
LD    HL,(40H)    :GET DATA
LD    A,L         :COMPLEMENT 8 LSB'S
CPL
LD    L,A
LD    A,H         :COMPLEMENT 8 MSB'S
CPL
LD    H,A
LD    (40H),HL    :STORE ONES COMPLEMENT
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	2A	LD	HL,(40H)
0001	40		
0002	00		
0003	7D	LD	A,L
0004	2F	CPL	
0005	6F	LD	L,A
0006	7C	LD	A,H
0007	2F	CPL	
0008	67	LD	H,A
0009	22	LD	(42H),HL
000A	42		
000B	00		
000C	76	HALT	

Despite the Z80's 16-bit instructions, you must use 8-bit instructions to perform most arithmetic and logical operations. The 16-bit instructions can, however, be used to load and store data and occasionally to do a few 16-bit arithmetic operations, such as addition, subtraction, incrementing, and decrementing. You will soon learn that the 16-bit instructions are far from a complete set and you may often run into awkward problems if using them to manipulate 16-bit data.

PROBLEMS

1) Twos Complement

Purpose: Place the twos complement of the contents of memory location 0040 into memory location 0041. The twos complement is the ones complement plus one.

Sample Problem:

$$(0040) = 3E$$

Result: $(0041) = C2$

The sum of the original number and its twos complement is zero (try the sample case).

2) 8-Bit Subtraction

Purpose: Subtract the contents of memory location 0041 from the contents of memory location 0040. Place the result into memory location 0042.

Sample Problem:

$$(0040) = 77$$
$$(0041) = 39$$

Result: $(0042) = 3E$

4-15

3) Shift Left Two Bits

Purpose: Shift the contents of memory location 0040 left two bits and place the result into memory location 0041. Clear the two least significant bit positions.

Sample Problem:

$$(0040) = 5D$$

Result: $(0041) = 74$

4) Mask Off Least Significant Four Bits

Purpose: Place the four most significant bits of the contents of memory location 0040 into memory location 0041. Clear the four least significant bits of memory location 0041.

Sample Problem:

$$(0040) = C4$$

Result: $(0041) = C0$

5) Set a Memory Location to All Ones

Purpose: Memory location 0040 is set to all ones (FF hex).

6) Word Assembly

Purpose: Combine the four least significant bits of memory locations 0040 and 0041 into a word and store them in memory location 0042. Place the four least significant bits of memory location 0040 into the four most significant bit positions of memory location 0042; place the four least significant bits of memory location 0041 into the four least significant bit positions of memory location 0042.

Sample Problem:

$$(0040) = 6A$$
$$(0041) = B3$$

Result: $(0042) = A3$

7) Find Smaller of Two Numbers

Purpose: Place the smaller of the contents of memory locations 0040 and 0041 into memory location 0042. Assume that 0040 and 0041 contain unsigned binary numbers.

Sample Problems:

a.
$$(0040) = 3F$$
$$(0041) = 2B$$

Result: $(0042) = 2B$

b.
$$(0040) = 75$$
$$(0041) = A8$$

Result: $(0042) = 75$

8) 24-Bit Addition

Purpose: Add the 24-bit number in memory locations 0040, 0041, and 0042 to the 24-bit number in memory locations 0043, 0044, and 0045. The most significant eight bits are in memory locations 0042 and 0045; the least significant eight bits are in memory locations 0040 and 0043. Store the result in memory locations 0046, 0047, and 0048 with the most significant bits in 0048 and the least significant bits in 0046.

4-16

```
          (0040)  =  2A
          (0041)  =  67
          (0042)  =  35
          (0043)  =  F8
          (0044)  =  A4
          (0045)  =  51
Result:   (0046)  =  22
          (0047)  =  0C
          (0048)  =  87
```

that is, 35672A
         +51A4F8
          870C22

9) Sum of Squares

Purpose: Calculate the squares of the contents of memory locations 0040 and 0041 and add them together. Place the result into memory location 0042. Assume that memory locations 0040 and 0041 both contain numbers between 0 and 7 inclusive ($0 \leq (0040) \leq 7$ and $0 \leq (0041) \leq 7$). Use the table of squares from the example entitled Table of Squares.

Sample Problem:

```
          (0040)  =  03
          (0041)  =  06
Result:   (0042)  =  2D
```

that is, $3^2 + 6^2 = 9 + 36 = 45$ (decimal)
                        $= 2D$ (hex)

10) 16-Bit Twos Complement

Purpose: Place the twos complement of the 16-bit number in memory locations 0040 and 0041 (most significant bits in 0041) into memory locations 0042 and 0043 (most significant bits in 0043).

Sample Problems:

a.
```
          (0040)  =  00
          (0041)  =  58
Result:   (0042)  =  00
          (0043)  =  A8
```

b.
```
          (0040)  =  72
          (0041)  =  00
Result:   (0042)  =  8E
          (0043)  =  FF
```

Chapter 5
SIMPLE PROGRAM LOOPS

The program loop is the basic structure that forces the CPU to repeat a sequence of instructions. Loops have four sections:

1) The initialization section, which establishes the starting values of counters, address registers (pointers), and other variables.

2) The processing section, where the actual data manipulation occurs. This is the section that does the work.

3) The loop control section, which updates counters and pointers for the next iteration.

4) The concluding section, which analyzes and stores the results.

Note that the computer performs Sections 1 and 4 once, while it may perform Sections 2 and 3 many times. Thus, the execution time of the loop will mainly depend on the execution time of Sections 2 and 3. You will want Sections 2 and 3 to execute as quickly as possible; do not worry about the execution time of Sections 1 and 4. A typical program loop can be flowcharted as shown in Figure 5-1, or the positions of the processing and loop control sections may be reversed as shown in Figure 5-2. The processing section in Figure 5-1 is always executed at least once, while the processing section in Figure 5-2 may not be executed at all. Figure 5-1 seems more natural, but Figure 5-2 is often more efficient and avoids the problem of what to do when there is no data (a bugaboo for computers, and the frequent cause of silly situations like the computer dunning someone for a bill of $0.00).

The loop structure can be used to process entire blocks of data. To accomplish this, the program must increment an address register (usually register pair HL) after each iteration so that the address register points to the next element in the data block. The next iteration will then perform the same operations on the data in the next memory location. The computer can handle blocks of any length with the same set of instructions.

Implied addressing through register pairs (particularly HL) is the key to processing a block of data with the Z80, since it allows you to vary the actual memory address by changing the contents of registers. Indexed addressing, while longer and slower on the Z80 than implied addressing, may be handy when processing more than one block of data. Note that in the immediate and direct addressing modes, the addresses that are used are completely determined by the instruction (and thus fixed if the program memory is read-only).
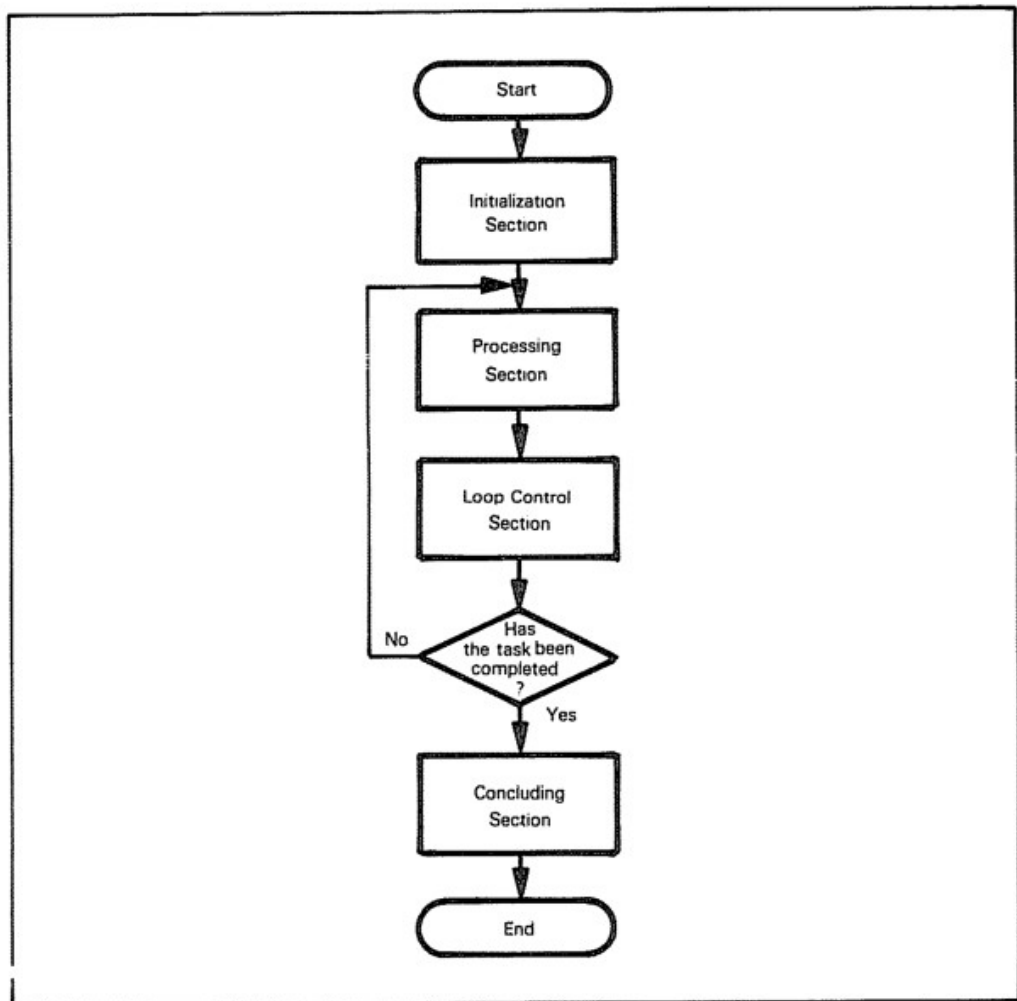
Figure 5-1. Flowchart of a Program Loop
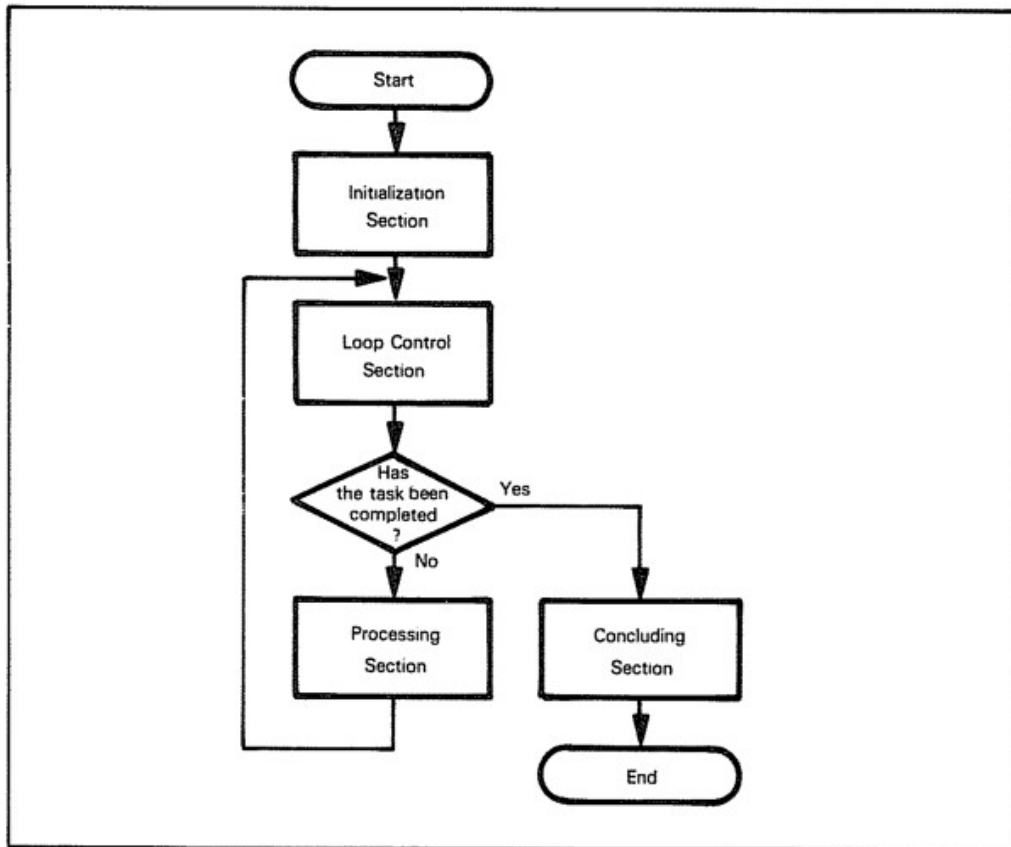
Figure 5-2. A Program Loop that Allows Zero Iterations

EXAMPLES

Sum of Data

Purpose: Calculate the sum of a series of numbers. The length of the series is in memory location 0041. and the series begins in memory location 0042. Store the sum in memory location 0040. Assume that the sum is an 8-bit number so that you can ignore carries.

```
8-BIT
SUMMATION
```

Sample Problem:

$$(0041) = 03$$
$$(0042) = 28$$
$$(0043) = 55$$
$$(0044) = 26$$

Result: $(0040) = (0042) + (0043) + (0044)$
$$= 28+55+26$$
$$= A3$$

There are three entries in the sum, since (0041)=03.

Flowchart:

```
                    ┌──────────────┐
                    │    Start     │
                    └──────┬───────┘
                           │
                           ▼
                ┌──────────────────────┐
                │  Pointer  =  41      │
                │  Count  =  (Pointer) │
                │  Sum  =  0           │
                └──────────┬───────────┘
                           │
          ┌──────────────►▼
          │     ┌──────────────────────┐
          │     │ Pointer = Pointer + 1│
          │     │ Sum = Sum            │
          │     │         + (Pointer)  │
          │     └──────────┬───────────┘
          │                │
          │                ▼
          │     ┌──────────────────────┐
          │     │  Count  =  Count - 1 │
          │     └──────────┬───────────┘
          │                │
          │                ▼
          │              ◇ Is
       No │         Count zero
          └──────────      ?
                           │ Yes
                           ▼
                ┌──────────────────────┐
                │    (40)  =  Sum      │
                └──────────┬───────────┘
                           │
                           ▼
                    ┌──────────────┐
                    │     End      │
                    └──────────────┘
```
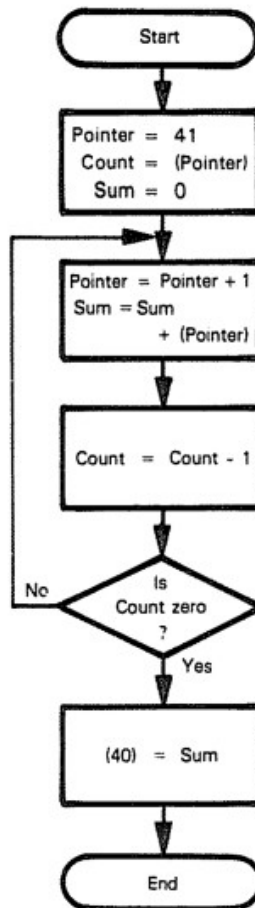
Note: (Pointer) is the contents of the memory location addressed by Pointer. Remember that on the Z80, Pointer is a 16-bit address, while (Pointer) is an 8-bit byte of data.

Source Program:

```
        LD      HL,41H
        LD      B,(HL)          :COUNT = LENGTH OF SERIES
        SUB     A               :SUM = ZERO
SUMD:   INC     HL
        ADD     A,(HL)          :SUM = SUM + DATA
        DEC     B
        JR      NZ,SUMD
        LD      (40H),A         :STORE SUM
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	46		LD	B,(HL)
0004	97		SUB	A
0005	23	SUMD:	INC	HL
0006	86		ADD	A,(HL)
0007	05		DEC	B
0008	20		JR	NZ,SUMD
0009	FB			
000A	32		LD	(40H),A
000B	40			
000C	00			
000D	76		HALT	

The initialization section of the program is the first three instructions which set the sum, counter, and data pointer to their starting values.

Note that you can use LD to transfer data between memory and any of the primary general purpose registers (i.e., A, B, C, D, E, H, L) using the address in Registers H and L. However, the only transfers allowed using direct addressing are those that move data to or from the Accumulator (i.e., LD A,(addr) and LD (addr),A — there is no instruction LD E,(addr), for example).

The processing section of the program is the single instruction ADD A,(HL) which adds the contents of the memory location being addressed by Registers H and L to the contents of the Accumulator, and stores the result in the Accumulator. This instruction does the real work of the program.

The loop control section of the program consists of the instructions INC HL and DEC B. INC HL updates the pointer so that the next iteration adds the next number to the sum. DEC B decrements the counter that keeps track of how many iterations are left.

The instruction JR NZ causes a branch if the Zero flag is zero. The offset is a twos complement number, and the count begins from the memory location immediately following the JR instruction. In this case, the required jump is from memory location 000A to memory location 0005. So the offset is:

$$\begin{array}{r} 0005 \\ -000A \end{array} = \begin{array}{r} 05 \\ +F6 \\ \hline FB \end{array}$$

If the Zero flag is one, the CPU executes the next instruction in sequence (i.e., LD (40H),A). Since DEC B was the last instruction before JR to affect the Zero flag, JR NZ,SUMD causes a jump to SUMD if DEC B does not produce a zero result, i.e.:

$$PC = \begin{cases} SUMD \text{ if } B \neq 0 \\ PC+2 \text{ if } B = 0 \end{cases}$$

(The 2 is caused by the two-word JR instruction).

The loop control sequence DEC followed by JR NZ is so common that the Z80 has a special instruction that both decrements the counter and performs the jump. This instruction is DJNZ, Decrement and Jump on Not Zero, which decrements Register B and then jumps by the specified relative offset if the remainder is not zero. So we could change the end of the example to:

```
DJNZ    SUMD
LD      (40H),A
HALT
```

Which has the object form:

07	10	DJNZ	SUMD
08	FC		
09	32	LD	(40H),A
0A	40		
0B	00		
0C	76	HALT	

This change saves one byte of memory and three clock cycles. Note, however, that you must use Register B as the counter since this is the register that DJNZ decrements.

Since the offset in Z80 relative jumps is only one byte long, such jumps can go no further than 127 locations forward or 128 locations backward (actually 129 forward or 126 backward, since the count starts at the end of the 2-word instruction). Longer jumps must use the JP instructions.

Most computer loops count down rather than up so that the Zero flag can serve as an exit condition. Remember that the Zero flag is 1 if the result was zero and 0 if the result was not zero. Try rewriting the program so that it counts up rather than down; which method is more efficient?

The order of instructions is often very important. DEC B must come right before JR NZ,SUMD, since otherwise the Zero result set by DEC B could be changed by another instruction. INC HL must come before ADD A,(HL) or else the first number added to the sum will be the contents of memory location 0041 instead of the contents of memory location 0042.

16-Bit Sum of Data

Purpose: Calculate the sum of a series of numbers. The length of the series is in memory location 0042 and the series itself begins in memory location 0043. Store the sum in memory locations 0040 and 0041 (eight least significant bits in 0040).

Sample Problem:

```
            (0042)  =  03
            (0043)  =  C8
            (0044)  =  FA
            (0045)  =  96

Result:  C8 + FA + 96 = 0258
            (0040)  =  58
            (0041)  =  02
```

Flowchart:
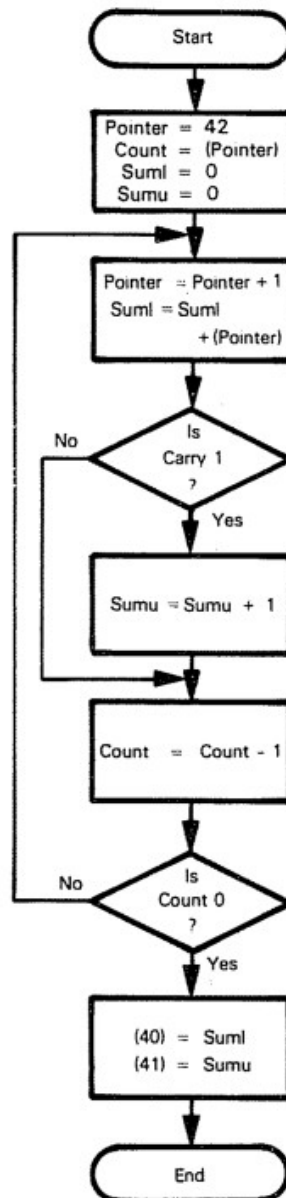
Source Program:

```
        LD      HL,42H
        LD      B,(HL)      :COUNT = LENGTH OF SERIES
        SUB     A           :LSB'S OF SUM = 0
        LD      C,A         :MSB'S OF SUM = 0
DSUMD:  INC     HL
        ADD     A,(HL)      :SUM = SUM + DATA
        JR      NC,CHCNT
        INC     C           :ADD CARRY TO MSB'S OF SUM
CHCNT:  DJNZ    DSUMD
        LD      HL,40H
        LD      (HL),A      :STORE LSB'S OF SUM
        INC     HL
        LD      (HL),C      :STORE MSB'S OF SUM
        HALT
```

5-7

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	21		LD	HL,42H
0001	42			
0002	00			
0003	46		LD	B,(HL)
0004	97		SUB	A
0005	4F		LD	C,A
0006	23	DSUMD:	INC	HL
0007	86		ADD	A,(HL)
0008	30		JR	NC,CHCNT
0009	01			
000A	0C		INC	C
000B	10	CHCNT:	DJNZ	DSUMD
000C	F9			
000D	21		LD	HL,40H
000E	40			
000F	00			
0010	77		LD	(HL),A
0011	23		INC	HL
0012	71		LD	(HL),C
0013	76		HALT	

The structure of this program is the same as the structure of the last one. The most significant bits of the sum now must be initialized and stored. The processing section consists of three instructions (ADD A,(HL); JR NC,CHCNT; and INC C), including a Conditional Jump.

JR NC,CHCNT causes a jump to memory location CHCNT if the Carry = 0. Thus, if there is no carry from the 8-bit addition, the program jumps around the statement that increments the most significant bits of the sum. The relative offset is:

$$\begin{array}{r} 000B \\ -000A \\ \hline 01 \end{array}$$

The relative offset for DJNZ DSUMD is:

$$\begin{array}{r} 0006 \\ -000D \end{array} = \begin{array}{r} 06 \\ +F3 \\ \hline F9 \end{array}$$

INC C adds 1 to the contents of Register C. Note that INC BC is a 16-bit increment that adds 1 to Register C and adds the resulting carry to Register B; INC C is an 8-bit increment that does not account for the carry.

Number of Negative Elements

Purpose: Determine the number of negative elements (most significant bit 1) in a block. The length of the block is in memory location 0041 and the block itself starts in memory location 0042. Place the number of negative elements in memory location 0040.

Sample Problem:

$$
\begin{aligned}
(0041) &= 06 \\
(0042) &= 68 \\
(0043) &= F2 \\
(0044) &= 87 \\
(0045) &= 30 \\
(0046) &= 59 \\
(0047) &= 2A
\end{aligned}
$$

Result: (0040) = 02, since 0043 and 0044 contain numbers with an MSB of 1.

Flowchart:
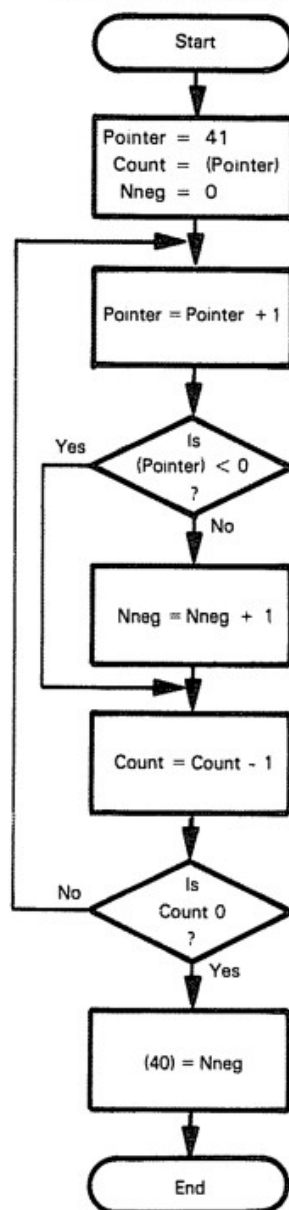
5-9

Source Program :

```
        LD      HL,41H
        LD      B,(HL)          ;COUNT = NUMBER OF ELEMENTS
        LD      C,0             ;NUMBER OF NEGATIVES = ZERO
SRNEG:  INC     HL
        LD      A,(HL)          ;GET NEXT ELEMENT
        AND     A               ;IS MSB ZERO?
        JP      P,CHCNT
        INC     C               ;NO, ADD 1 TO NUMBER OF NEGATIVES
CHCNT:  DJNZ    SRNEG
        LD      A,C             ;STORE NUMBER OF NEGATIVES
        LD      (40H),A
        HALT
```

Object Program :

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL,41H
0001	41		
0002	00		
0003	46	LD	B,(HL)
0004	0E	LD	C,0
0005	00		
0006	23	SRNEG: INC	HL
0007	7E	LD	A,(HL)
0008	A7	AND	A
0009	F2	JP	P,CHCNT
000A	0D		
000B	00		
000C	0C	INC	C
000D	10	CHCNT: DJNZ	SRNEG
000E	F7		
000F	79	LD	A,C
0010	32	LD	(40H),A
0011	40		
0012	00		
0013	76	HALT	

AND A simply sets the flag bits according to the contents of the Accumulator without affecting those contents; OR A has the same effect. This is necessary since merely loading the Accumulator does not affect the flags.

JP P,CHCNT requires a full 16-bit address. There is no relative jump on the Sign flag like there is on the Carry and Zero flags.

Note that all we really want to do is test the value of bit 7 of the memory location addressed by Registers H and L. The Z80 has a special bit testing instruction, BIT, that is designed specifically for this purpose. BIT sets the Z flag to the complement of the indicated bit within the indicated register or memory location. For example, BIT 5,D will set Z to 1 if bit 5 of Register D is zero, and to 0 if bit 5 of Register D is one. An implementation of this alternative is as follows.

Source Program:

```
        LD      HL.41H
        LD      B,(HL)          ;COUNT = NUMBER OF ELEMENTS
        LD      C.0             ;NUMBER OF NEGATIVES = ZERO
SRNEG:  INC     HL
        BIT     7,(HL)          ;IS NEXT ELEMENT NEGATIVE?
        JR      Z.CHCNT
        INC     C               ;YES, ADD 1 TO NUMBER OF NEGATIVES
CHCNT:  DJNZ    SRNEG
        LD      A,C             ;STORE NUMBER OF NEGATIVES
        LD      (40H),A
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	46		LD	B,(HL)
0004	0E		LD	C.0
0005	00			
0006	23	SRNEG:	INC	HL
0007	CB		BIT	7,(HL)
0008	7E			
0009	28		JR	Z.CHCNT
000A	01			
000B	0C		INC	C
000C	10	CHCNT:	DJNZ	SRNEG
000D	F8			
000E	79		LD	A.C
000F	32		LD	(40H),A
0010	40			
0011	00			
0012	76		HALT	

BIT 7,(HL) sets the Z bit if bit 7 of the memory location addressed by Registers H and L is zero, and clears the Z bit if bit 7 of that location is one. BIT does not affect any registers or memory locations.

This program uses JR Z,CHCNT since no incrementing is necessary if the addressed bit is zero.

Still another approach would be to use the instruction RLC (HL) to shift the sign bit of the data in memory to the Carry. The required jump would then be JR NC,CHCNT. However, this approach uses extra time (RLC (HL) takes 15 cycles as compared to the 12 needed by BIT 7,(HL)) and also changes the data in memory which may be needed for other purposes. Note that these disadvantages are related: the extra time is needed to return the result to the memory location.

Find Maximum

Purpose: Find the largest element in a block of data. The length of the block is in memory location 0041 and the block itself begins in memory location 0042. Store the maximum in memory location 0040. Assume that the numbers in the block are all 8-bit unsigned binary numbers.

Sample Problem:

$$(0041) = 05$$
$$(0042) = 67$$
$$(0043) = 79$$
$$(0044) = 15$$
$$(0045) = E3$$
$$(0046) = 72$$

Result: $(0040) = E3$. since this is the largest of the five unsigned numbers.

Flowchart:

```
                        ┌──────────────┐
                        │    Start     │
                        └──────┬───────┘
                               ▼
                    ┌──────────────────────┐
                    │ Pointer  =  41       │
                    │ Count  =  (Pointer)  │
                    │ Max  =  0            │
                    └──────────┬───────────┘
                               ▼
                    ┌──────────────────────┐
             ┌──────│ Pointer = Pointer + 1│
             │      └──────────┬───────────┘
             │                 ▼
             │              ╱Is      ╲
        Yes  │   ◄────────╱ Max > (Pointer) ╲
             │            ╲      ?   ╱
             │              ╲      ╱
             │                │ No
             │                ▼
             │      ┌──────────────────────┐
             │      │   Max = (Pointer)    │
             │      └──────────┬───────────┘
             │                 ▼
             │      ┌──────────────────────┐
             │      │  Count = Count - 1   │
             │      └──────────┬───────────┘
             │                 ▼
             │              ╱Is     ╲
          No └───────────╱ Count 0  ╲
                         ╲     ?   ╱
                          ╲      ╱
                            │ Yes
                            ▼
                  ┌──────────────────────┐
                  │     (40) = Max       │
                  └──────────┬───────────┘
                             ▼
                  ┌──────────────────────┐
                  │        End           │
                  └──────────────────────┘
```
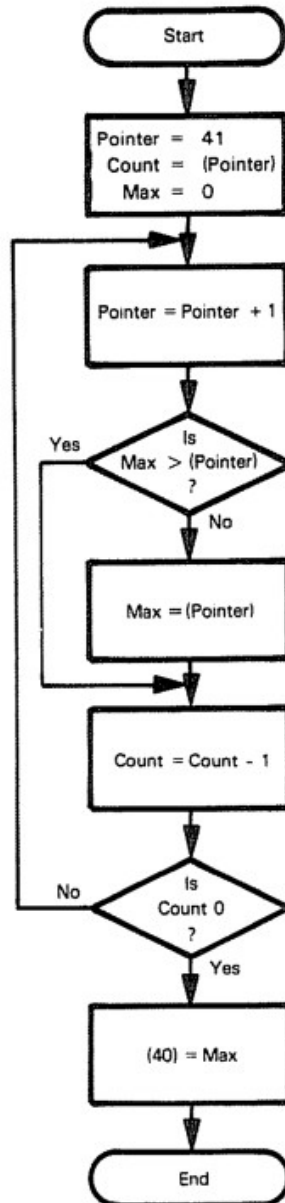
Source Program:

```
        LD      HL,41H       ;POINT TO COUNT
        LD      B,(HL)       ;COUNT = NUMBER OF ELEMENTS
        SUB     A            ;MAXIMUM = MINIMUM POSSIBLE VALUE (ZERO)
NEXTE:  INC     HL
        CP      (HL)         ;IS NEXT ELEMENT ABOVE MAXIMUM?
        JR      NC.DECNT
        LD      A,(HL)       ;YES, REPLACE MAXIMUM WITH ELEMENT
DECNT:  DJNZ    NEXTE
        LD      (40H),A      ;SAVE MAXIMUM
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL,41H
0001	41		
0002	00		
0003	46	LD	B,(HL)
0004	97	SUB	A
0005	23	NEXTE: INC	HL
0006	BE	CP	(HL)
0007	30	JR	NC.DECNT
0008	01		
0009	7E	LD	A,(HL)
000A	10	DECNT: DJNZ	NEXTE
000B	F9		
000C	32	LD	(40H),A
000D	40		
000E	00		
000F	76	HALT	

The relative offset for JR NC.DECNT is:

$$\begin{array}{r} 000A \\ -0009 \\ \hline 01 \end{array}$$

The relative offset for DJNZ NEXTE is:

$$\begin{array}{r} 0005 \\ -000C \end{array} = \begin{array}{r} 05 \\ +F4 \\ \hline F9 \end{array}$$

The first three instructions of this program form the initialization section.

This program takes advantage of the fact that zero is the smallest 8-bit unsigned binary number. When you set the register that contains the maximum value — in this case the Accumulator — to the minimum possible value before you enter the loop, then the program will set the Accumulator to a larger value unless all the elements in the array are zeros.

The program works properly if there are two elements, but not if there are one or none at all. Why? How could you solve this problem?

The instruction CP (HL) sets the Carry flag as follows (ELEMENT is the contents of the address in Registers H and L and MAX is the contents of the Accumulator):

$$CARRY = 1 \text{ if ELEMENT} > MAX$$
$$CARRY = 0 \text{ if ELEMENT} \leq MAX$$

If CARRY = 0, the program proceeds to DECNT and does not change the maximum. If CARRY = 1, the program replaces the old maximum with the current element by executing the instruction LD A,(HL).

The program does not work if the numbers are signed because negative numbers will appear to be larger than positive numbers. The problem is somewhat tricky because overflow could make the result appear to have the wrong sign.

Remember that overflow occurs when the magnitude of a result affects its sign bit. The Z80 has a Parity/Overflow flag that indicates when twos complement overflow has occurred. Arithmetic operations that result in overflow set this flag. You can then test its value with the instructions JP PE,ADDR (Jump on Parity Even — or Jump on Overflow) or JP PO,ADDR (Jump on Parity Odd — or Jump on No Overflow). One thing you may have to watch is that this Z80 usage is inconsistent with the 8080A or 8085 microprocessors, which always use the P flag to indicate parity. The 8080A and 8085 microprocessors have no overflow indicator.

Justify a Binary Fraction

Purpose: Shift the contents of memory location 0040 left until the most significant bit of the number is 1. Store the result in memory location 0041 and the number of left shifts required in memory location 0042. If the contents of memory location 0040 are zero, clear both 0041 and 0042.

Note: The process is just like converting a number to a scientific notation; for example:

$$0.0057 = 5.7 \times 10^{-3}$$

Sample Problems:

a.		(0040)	=	22
	Result:	(0041)	=	88
		(0042)	=	02
b.		(0040)	=	01
	Result:	(0041)	=	80
		(0042)	=	07
c.		(0040)	=	CB
	Result:	(0041)	=	CB
		(0042)	=	00
d.		(0040)	=	00
	Result:	(0041)	=	00
		(0042)	=	00

Flowchart:
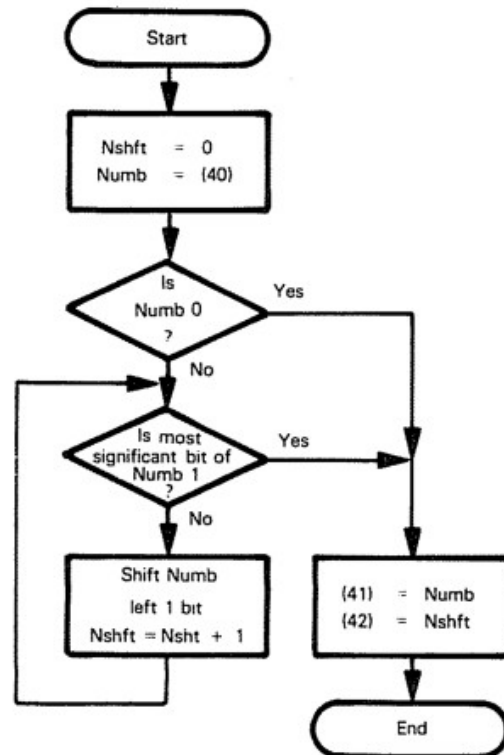
Source Program:

```
        LD      B.0         ;NUMBER OF SHIFTS =ZERO
        LD      HL.40H
        LD      A.(HL)      ;GET DATA
        AND     A           ;IS DATA ZERO?
        JR      Z.DONE      ;YES, DONE
CHKMS:  JP      M.DONE      ;DONE IF SIGN BIT IS ONE
        INC     B           ;ADD 1 TO NUMBER OF SHIFTS
        ADD     A.A         ;SHIFT LEFT ONE BIT
        JP      CHKMS
DONE:   INC     HL
        LD      (HL).A      ;SAVE JUSTIFIED DATA
        INC     HL
        LD      (HL).B      ;SAVE NUMBER OF SHIFTS
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	06	LD	B,0
0001	00		
0002	21	LD	HL,40H
0003	40		
0004	00		
0005	7E	LD	A,(HL)
0006	A7	AND	A
0007	28	JR	Z,DONE
0008	08		
0009	FA	CHKMS: JP	M,DONE
000A	11		
000B	00		
000C	04	INC	B
000D	87	ADD	A,A
000E	C3	JP	CHKMS
000F	09		
0010	00		
0011	23	DONE: INC	HL
0012	77	LD	(HL),A
0013	23	INC	HL
0014	70	LD	(HL),B
0015	76	HALT	

JP M,DONE causes a jump to location DONE if the Sign bit is 1. This condition may mean that the last result was a negative number or may just mean that its most significant bit was 1 — the computer supplies only the results; the programmer must provide the interpretation.

ADD A,A adds the number in the Accumulator to itself. The program uses this instruction, rather than RLA or RLCA, because ADD A affects the Sign bit while RLA and RLCA do not.

We could reorganize this program so as to eliminate an extraneous JP and use relative rather than absolute jumps. One reorganized version would be:

```
        LD    B,0         ;NUMBER OF SHIFTS = ZERO
        LD    HL,40H
        LD    A,(HL)       ;GET DATA
        AND   A            ;IS DATA ZERO?
        JR    Z,DONE       ;YES, DONE
        DEC   B            ;ADJUST NUMBER OF SHIFTS BACK ONE
CHKMS:  INC   B            ;ADD 1 TO NUMBER OF SHIFTS
        RLA                ;SHIFT LEFT ONE BIT
        JR    NC,CHKMS     ;CONTINUE IF MSB NOT ONE
        RRA                ;ADJUST DATA BACK
DONE:   INC   HL
        LD    (HL),A       ;SAVE JUSTIFIED DATA
        INC   HL
        LD    (HL),D       ;SAVE NUMBER OF SHIFTS
        HALT
```

Show that this version also works. What are its advantages and disadvantages as compared to the previous program?

PROBLEMS

1) Checksum of Data

Purpose: Calculate the checksum of a series of numbers. The length of the series is in memory location 0041 and the series itself begins in memory location 0042. Store the checksum in memory location 0040. The checksum is formed by Exclusive-ORing all the numbers in the series together.

Note: Such checksums are often used in paper tape and cassette systems to ensure that the data has been read correctly. The calculated checksum is compared to the one stored with the data — if the two checksums do not agree, the system will usually either indicate an error to the operator or automatically read the data again.

Sample Problem:

$$
\begin{aligned}
(0041) &= 03 \\
(0042) &= 28 \\
(0043) &= 55 \\
(0044) &= 26
\end{aligned}
$$

$$
\begin{aligned}
\text{Result:} \quad (0040) &= (0042) \oplus (0043) \oplus (0044) \\
&= 28 \oplus 55 \oplus 26 \\
&= 00101000 \\
\oplus\ & \underline{01010101} \\
& 01111101 \\
\oplus\ & \underline{00100110} \\
& 01011011 \\
&= 5B
\end{aligned}
$$

2) Sum of 16-Bit Data

Purpose: Calculate the sum of a series of 16-bit numbers. The length of the series is in memory location 0042 and the series itself begins in memory location 0043. Store the sum in memory locations 0040 and 0041 (eight most significant bits in 0041). Each 16-bit number occupies two memory locations, with the eight most significant bits in the higher address. Assume that the sum can be contained in 16 bits.

Sample Problem:

$$
\begin{aligned}
(0042) &= 03 \\
(0043) &= F1 \\
(0044) &= 28 \\
(0045) &= 1A \\
(0046) &= 30 \\
(0047) &= 89 \\
(0048) &= 4B
\end{aligned}
$$

Result: $28F1 + 301A + 4B89 = A494$

$$
\begin{aligned}
(0040) &= 94 \\
(0041) &= A4
\end{aligned}
$$

3) Number of Zero, Positive, and Negative Numbers

Purpose: Determine the number of zero, positive (most significant bit zero but entire number not zero), and negative (most significant bit 1) elements in a block. The length of the block is in memory location 0043 and the block itself starts in memory location 0044. Place the number of negative elements in memory location 0040, the number of zero elements in memory location 0041, and the number of positive elements in memory location 0042.

Sample Problem:

$$(0043) = 06$$
$$(0044) = 68$$
$$(0045) = F2$$
$$(0046) = 87$$
$$(0047) = 00$$
$$(0048) = 59$$
$$(0049) = 2A$$

Result: 2 negative, 1 zero, and 3 positive, so
$$(0040) = 02$$
$$(0041) = 01$$
$$(0042) = 03$$

4) Find Minimum

Purpose: Find the smallest element in a block of data. The length of the block is in memory location 0041 and the block itself begins in memory location 0042. Store the minimum in memory location 0040. Assume that the numbers in the block are 8-bit unsigned binary numbers.

Sample Problem:

$$(0041) = 05$$
$$(0042) = 67$$
$$(0043) = 79$$
$$(0044) = 15$$
$$(0045) = E3$$
$$(0046) = 72$$

Result: $(0040) = 15$, since this is the smallest of the five unsigned numbers.

5) Count 1 Bits

Purpose: Determine how many bits in memory location 0040 are one and place the result in memory location 0041.

Sample Problem:

$$(0040) = 3B = 00111011$$

Result: $(0041) = 05$

Chapter 6
CHARACTER-CODED DATA

Microprocessors often handle character-coded data. Not only do keyboards, teletypewriters, communications devices, displays, and computer terminals expect or provide character-coded data; many instruments, test systems, and controllers also require data in this form. The most commonly used code is ASCII. Baudot and EBCDIC are found less frequently. We will assume all of our character-coded data to be 7-bit ASCII with the most significant bit zero (see Table 6-1).

Some principles to remember in handling ASCII-coded data are:

```
┌─────────────┐
│ HANDLING    │
│ DATA IN     │
│ ASCII       │
└─────────────┘
```

1) The codes for the numbers and letters form ordered sub-sequences. The codes for the decimal numbers are hex 30 through 39, so that you can convert between decimal and ASCII with a simple additive factor. The codes for the upper-case letters are hex 41 through 5A, so that you can do alphabetic ordering by sorting the data in increasing numerical order.

2) The computer draws no distinction between printing and non-printing characters. This distinction is made only by I/O devices.

3) An ASCII device will handle only ASCII data. To print a 7 on an ASCII printer, the microprocessor must send hex 37 to the printer; hex 07 is the 'bell' character. Similarly, the microprocessor will receive the character 9 from an ASCII keyboard as hex 39; hex 09 is the 'tab' character.

4) Some ASCII devices do not use the full character set. For example, control characters and lower-case letters may be ignored or printed as spaces or question marks.

5) Some widely used ASCII characters are:

   $0A_{16}$ - line feed (LF)

   $0D_{16}$ - carriage return (CR)

   20_{16} - space

   $3F_{16}$ - ? (question mark)

   $7F_{16}$ - rubout or delete character

6) Each ASCII character occupies seven bits. This allows a large character set but is wasteful when the data is limited to a small subset such as the decimal numbers. An 8-bit byte, for example, can hold only one ASCII-coded decimal digit, while it can hold two BCD-coded digits.

Table 6-1. Hex-ASCII Table

Hex MSD \ Hex LSD	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	''	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	•	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

EXAMPLES

Length of a String of Characters

Purpose: Determine the length of a string of ASCII characters (seven bits with most significant bit zero). The string starts in memory location 0041: the end of the string is marked by a carriage return character ('CR', hex 0D). Place the length of the string (excluding the carriage return) into memory location 0040.

Sample Problems:

a.

(0041) = 0D

Result: (0040) = 00 since the first character is a carriage return.

b.

(0041) = 52 'R'
(0042) = 41 'A'
(0043) = 54 'T'
(0044) = 48 'H'
(0045) = 45 'E'
(0046) = 52 'R'
(0047) = 0D CR

Result: (0040) = 06

Flowchart:
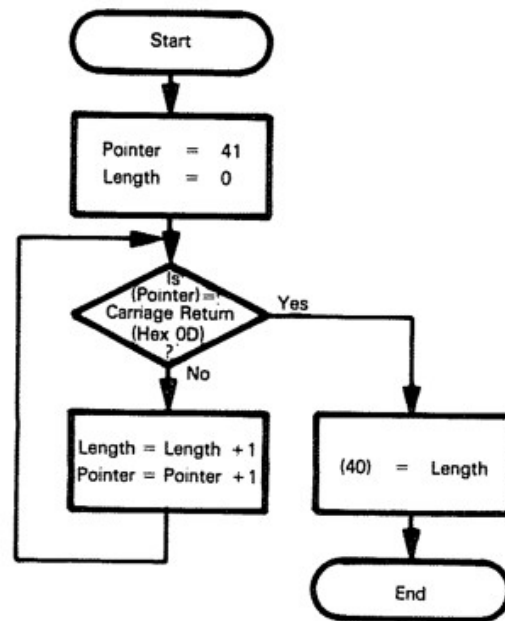
Source Program:

```
        LD      HL.41H      ;POINTER = START OF STRING
        LD      B,0         ;STRING LENGTH = ZERO
        LD      A,0DH       ;GET ASCII CARRIAGE RETURN TO COMPARE
CHKCR:  CP      (HL)        ;IS CHARACTER A CARRIAGE RETURN?
        JR      Z.DONE      ;YES, DONE
        INC     B           ;NO. ADD 1 TO STRING LENGTH
        INC     HL
        JR      CHKCR       ;TRY NEXT CHARACTER
DONE:   LD      A,B         ;SAVE STRING LENGTH
        LD      (40H),A
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	06		LD	B,0
0004	00			
0005	3E		LD	A,0DH
0006	0D			
0007	BE	CHKCR:	CP	(HL)
0008	28		JR	Z,DONE
0009	04			
000A	04		INC	B
000B	23		INC	HL
000C	18		JR	CHKCR
000D	F9			
000E	78	DONE:	LD	A,B
000F	32		LD	(40H),A
0010	40			
0011	00			
0012	76		HALT	

The carriage return (CR) is just another ASCII character (hex 0D) as far as the computer is concerned. The fact that the output device treats the carriage return as a control character rather than as a printing character does not affect the computer.

The Compare instruction, CP, sets the flags as if a subtraction had been performed, but leaves the carriage return character in the Accumulator for later comparisons. The Zero (Z) flag is affected as follows:

$Z = 1$ if the character in the string is a carriage return

$Z = 0$ if it is not a carriage return

The instruction INC B adds 1 to the string length counter in Register B. LD B,0 initializes this counter to zero before the loop begins. Remember to initialize variables before using them in a loop.

This loop does not terminate because a counter is decremented to zero. The computer will simply continue examining characters until it finds a carriage return. You may have to place a maximum count in a loop like this to avoid problems with erroneous strings that do not contain a carriage return. What would happen if the example program were used with such a string?

Note that, by rearranging the logic and changing the initial conditions, you can shorten the program and decrease its execution time. If we adjust the flowchart so that the program increments the counter and pointer before it looks for the carriage return, only one Jump instruction is necessary instead of two. The new flowchart and program are as follows:

Flowchart:
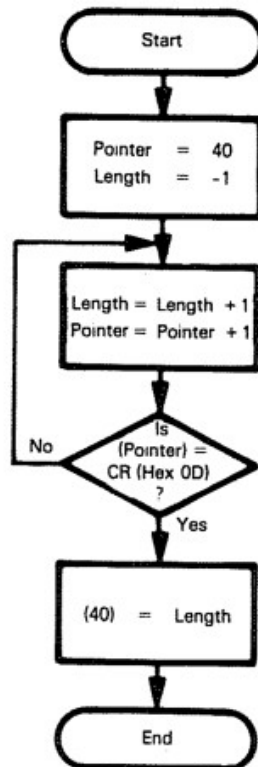
Source Program:

```
        LD     HL,40H       :POINTER = BYTE BEFORE STRING
        LD     B,0FFH       :LENGTH = -1
        LD     A,0DH        :GET ASCII CARRIAGE RETURN TO COMPARE
CHKCR:  INC    HL
        INC    B            :ADD 1 TO STRING LENGTH
        CP     (HL)         :IS CHARACTER A CARRIAGE RETURN?
        JR     NZ,CHKCR     :NO. CHECK NEXT CHARACTER
        LD     A,B          :YES. SAVE STRING LENGTH
        LD     (40H),A
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	21		LD	HL,40H
0001	40			
0002	00			
0003	06		LD	B,0FFH
0004	FF			
0005	3E		LD	A,0DH
0006	0D			
0007	23	CHKCR:	INC	HL
0008	04		INC	B
0009	BE		CP	(HL)
000A	20		JR	NZ,CHKCR
000B	FB			
000C	78		LD	A,B
000D	32		LD	(40H),A
000E	40			
000F	00			
0010	76		HALT	

The task of looking for a particular value in a list, table, or string is a common one. The Z80 microprocessor has, in fact, special instructions that simplify this task.

These special instructions are called Block Search Instructions; they operate as follows:

BLOCK SEARCH INSTRUCTIONS

CPI compares the contents of the memory location addressed by HL with the contents of the Accumulator (just like CP (HL)). It then increments HL and decrements the byte counter (register pair BC). The Parity/Overflow bit is reset if the byte counter is decremented to zero and set otherwise. CPD is the same instruction except that it decrements HL instead of incrementing it.

CPIR and CPDR are the repeated forms of the Block Search instructions. These instructions repeat the basic Search instruction until either BC is decremented to zero or a true comparison occurs (i.e., A = (HL)). Remember that decrementing BC to zero resets the Parity/Overflow bit, while finding a match sets the Zero bit.

Note that BC contains a 16-bit counter. Thus, the Block Search Instructions can handle strings of any length.

A version of the previous program using CPI is shown below.

Source Program:

```
        LD      HL,41H      ;POINTER = START OF STRING
        LD      BC,0        ;BYTE COUNTER = ZERO
        LD      A,0DH       ;GET ASCII CARRIAGE RETURN TO COMPARE
CHKCR:  CPI                 ;IS CHARACTER A CARRIAGE RETURN?
        JR      NZ,CHKCR    ;NO, CHECK NEXT CHARACTER
        LD      A,0FFH      ;YES, CALCULATE STRING LENGTH
        SUB     C
        LD      (40H),A     ;SAVE STRING LENGTH
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	21		LD	HL.41H
0001	41			
0002	00			
0003	01		LD	BC.0
0004	00			
0005	00			
0006	3E		LD	A.0DH
0007	0D			
0008	ED	CHKCR:	CPI	
0009	A1			
000A	20		JR	NZ.CHKCR
000B	FC			
000C	3E		LD	A.0FFH
000D	FF			
000E	91		SUB	C
000F	32		LD	(40H).A
0010	40			
0011	00			
0012	76		HALT	

A little manipulation is necessary to calculate the string length, since CPI decrements the byte counter (BC) instead of incrementing it as we did with INC B in the earlier program. Also, the byte counter is decremented one extra time when the carriage return is found. How could you adjust the initial conditions to handle this problem?

In fact, we can improve the program even further by using CPIR to remove the need for the relative jump JR. CPIR does everything that CPI does, but it also automatically repeats the comparison procedure unless A = (HL) or BC has been decremented to zero. The program using CPIR is shown below.

Source Program:

```
LD    HL.41H    :POINTER = START OF STRING
LD    BC.0      :BYTE COUNTER = ZERO
LD    A.0DH     :GET ASCII CARRIAGE RETURN TO COMPARE
CPIR            :SEARCH FOR CARRIAGE RETURN
LD    A.0FFH    :CALCULATE STRING LENGTH FROM COUNTER
SUB   C
LD    (40H).A   :SAVE STRING LENGTH
HALT
```

6-7

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL,41H
0001	41		
0002	00		
0003	01	LD	BC,0
0004	00		
0005	00		
0006	3E	LD	A,0DH
0007	0D		
0008	ED	CPIR	
0009	B1		
000A	3E	LD	A,0FFH
000B	FF		
000C	91	SUB	C
000D	32	LD	(40H),A
000E	40		
000F	00		
0010	76	HALT	

The multiple operation instructions like CPI and CPIR have the same effect as the sequences they replace. The savings in execution time and memory come about because the processor needs fewer instructions for each pass through the loop. Thus, the real savings is in loop execuion.

All these programs assume that the string is less than 256 bytes long. How would you change them to handle longer strings?

Find First Non-Blank Character

Purpose: Search a string of ASCII characters (seven bits with most significant bit zero) for a non-blank character. The string starts in memory location 0042. Place the address of the first non-blank character into memory locations 0040 and 0041 (most significant bits in 0041). A blank character is hex 20 in ASCII.

Sample Problems:

a.
      (0042) = 37 '7'

    Result: (0040) = 42, since memory location 0042 contains a non-blank character.

      (0041) = 00

b.
      (0042) = 20 SP
      (0043) = 20 SP
      (0044) = 20 SP
      (0045) = 46 F
      (0046) = 20 SP

    Result: (0040) = 45, since the three previous memory locations all contain blanks.

      (0041) = 00

Flowchart:
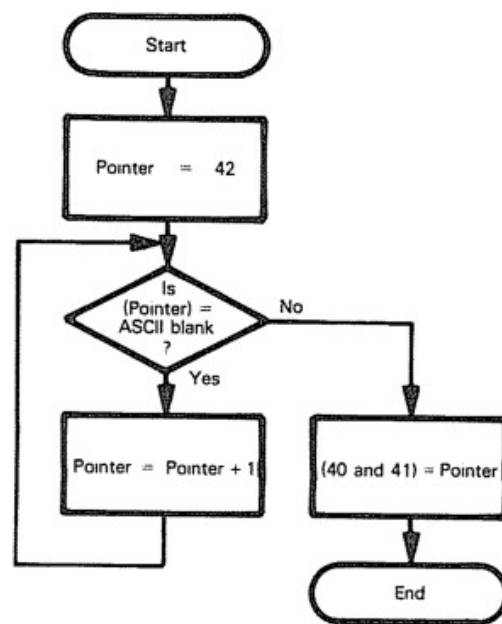
Source Program:

```
        LD      HL,42H      :POINTER = START OF STRING
        LD      A,20H       :GET ASCII SPACE FOR COMPARISON
CHBLK:  CP      (HL)        :IS CHARACTER AN ASCII SPACE?
        JR      NZ,DONE     :NO, THROUGH
        INC     HL
        JR      CHBLK       :YES, EXAMINE NEXT CHARACTER
DONE:   LD      (40H),HL    :NO, SAVE ADDRESS OF FIRST NON-BLANK
                            ;   CHARACTER
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL,42H
0001	42		
0002	00		
0003	3E	LD	A,20H
0004	20		
0005	BE	CHBLK. CP	(HL)
0006	20	JR	NZ,DONE
0007	03		
0008	23	INC	HL
0009	18	JR	CHBLK
000A	FA		
000B	22	DONE: LD	(40H),HL
000C	40		
000D	00		
000E	76	HALT	

Looking for spaces in strings is a common task. Spaces often are eliminated from strings when they are used simply to increase readability or to fit particular formats. It is obviously wasteful to store and transmit beginning, ending or extra spaces, particularly if you are paying for the communications capability and memory required. Data and program entry, however, are much simpler if extra spaces are tolerated. Microcomputers are often used in situations like this to convert data between forms that are easy for humans to use and forms that are efficiently handled on computers and communications lines.

The instruction LD (addr),HL is convenient for storing addresses in the Z80 format (least significant byte first). LD (40H),HL stores the contents of Register L in memory location 0040 and the contents of Register H in memory location 0041.

Again, if we alter the initial conditions so that the loop control section precedes the processing section, we can reduce the number of bytes in the program and decrease the loop's execution time. The rearranged flowchart is:
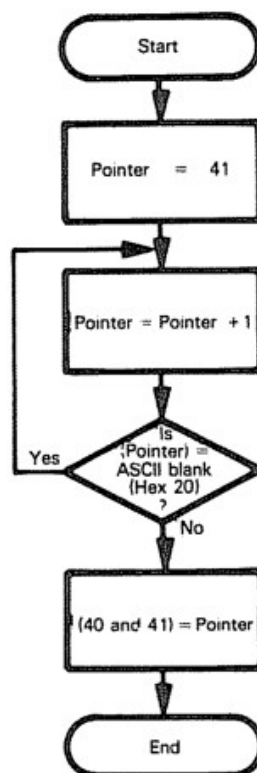
Source Program:

```
        LD    HL,41H      ;POINT TO BYTE BEFORE STRING
        LD    A,20H       ;GET ASCII SPACE FOR COMPARISON
CHBLK:  INC   HL
        CP    (HL)        ;IS CHARACTER AN ASCII SPACE?
        JR    Z,CHBLK     ;YES, KEEP EXAMINING CHARACTERS
        LD    (40H),HL    ;NO, SAVE ADDRESS OF FIRST NON-BLANK
                          ;  CHARACTER
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	3E		LD	A,20H
0004	20			
0005	23	CHBLK:	INC	HL
0006	BE		CP	(HL)
0007	28		JR	Z,CHBLK
0008	FC			
0009	22		LD	(40H),HL
000A	40			
000B	00			
000C	76		HALT	

As in the previous example, we could replace the sequence INC HL, CP (HL) with the single instruction CPI. However, since we do not need the byte counter in this program, CPI takes just as much memory (two bytes) and more time (16 clock cycles instead of 13) than the instructions it replaces. We could not use CPIR here since we want the program to terminate when the characters are **not** the same.

Replace Leading Zeros with Blanks

Purpose: Edit a string of ASCII decimal characters by replacing all leading zeros with blanks. The string starts in memory location 0041; assume that it consists entirely of ASCII-coded decimal digits. The length of the string is in memory location 0040.

Sample Problems:

a.
$$\begin{aligned}(0040) &= 02 \\ (0041) &= 36 \quad \text{'6'}\end{aligned}$$

The program leaves the string unchanged, since the leading digit is not zero.

b.
$$\begin{aligned}(0040) &= 08 \\ (0041) &= 30 \quad \text{'0'} \\ (0042) &= 30 \quad \text{'0'} \\ (0043) &= 38 \quad \text{'8'}\end{aligned}$$

Result:
$$\begin{aligned}(0041) &= 20 \quad \text{SP} \\ (0042) &= 20 \quad \text{SP}\end{aligned}$$

Flowchart:
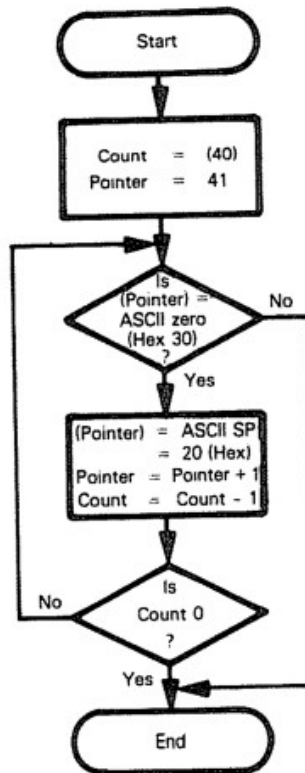
Source Program:

```
        LD      HL.40H
        LD      B,(HL)        :COUNT = STRING LENGTH
        LD      A,'0'         :GET ASCII ZERO FOR COMPARISON
CHKZ:   INC     HL
        CP      (HL)          :IS LEADING DIGIT ZERO?
        JR      NZ.DONE       :NO. THROUGH
        LD      (HL).20H      :REPLACE LEADING ZERO WITH BLANK
        DJNZ    CHKZ          :EXAMINE NEXT DIGIT IF ANY
DONE:   HALT
```

Single quotation marks around characters indicate ASCII.

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	21		LD	HL,40H
0001	40			
0002	00			
0003	46		LD	B,(HL)
0004	3E		LD	A,'0'
0005	30			
0006	23	CHKZ:	INC	HL
0007	BE		CP	(HL)
0008	20		JR	NZ,DONE
0009	04			
000A	36		LD	(HL),20H
000B	20			
000C	10		DJNZ	CHKZ
000D	F8			
000E	76	DONE:	HALT	

You will frequently want to edit decimal strings before they are printed or displayed to improve their appearance. Common editing tasks include eliminating leading zeros, justifying numbers, adding signs or other identifying markers, and rounding. Clearly, printed numbers like 0006 or $27.34382 can be confusing and annoying.

Here the loop has two exits — one if the processor finds a non-zero digit and the other if it has examined the entire string.

The instruction LD (HL),20H places 20 (hex) into the memory location addressed by Registers H and L. You could also initialize Register C to 20 hex (i.e., LD C,20H) and use LD (HL),C to replace the leading zero with a blank. Note the tradeoffs involved in this example. LD (HL),C executes faster than LD (HL),20H and would thus decrease the inner loop's execution time. The overhead required, however, is an LD C,20H instruction in the initialization section of the routine. If this example were to be used in a cash register application, which sequence would you choose and why?

All digits in the string are assumed to be ASCII; that is, the digits are hex 30 through 39 rather than the ordinary decimal 0 to 9. The conversion from decimal to ASCII is simply a matter of adding hex 30 to the decimal digit.

You may have to be careful, when blanking leading zeros, to leave one zero in the event that all the digits are zero. How would you do this?

Note that each ASCII digit requires eight bits, as compared to four for a BCD digit. Therefore, ASCII is an expensive format in which to store or transmit numerical data.

Add Even Parity to ASCII Characters

Purpose: Add even parity to a string of 7-bit ASCII characters. The length of the string is in memory location 0040 and the string itself begins in memory location 0041. Place even parity in the most significant bit of each character by setting the most significant bit to 1 if that makes the total number of 1 bits in the word an even number.

Sample Problem:

$$
\begin{array}{rll}
(0040) &=& 06 \\
(0041) &=& 31 \\
(0042) &=& 32 \\
(0043) &=& 33 \\
(0044) &=& 34 \\
(0045) &=& 35 \\
(0046) &=& 36 \\
\end{array}
$$

Result:
$$
\begin{array}{rll}
(0041) &=& B1 \\
(0042) &=& B2 \\
(0043) &=& 33 \\
(0044) &=& B4 \\
(0045) &=& 35 \\
(0046) &=& 36 \\
\end{array}
$$

Flowchart:

6-14
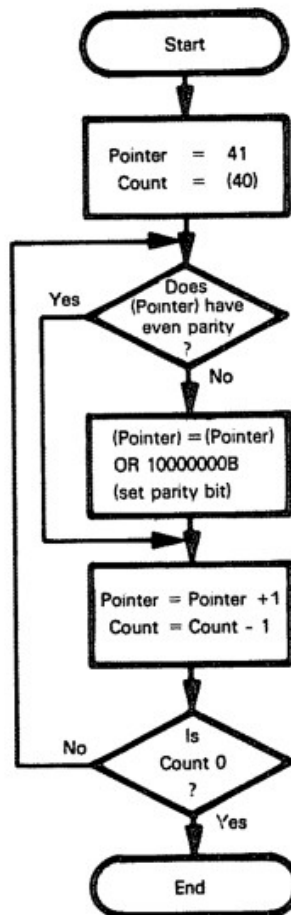
Source Program:

```
        LD      HL,40H
        LD      B,(HL)          :GET STRING LENGTH
        LD      C,10000000B     :GET PARITY BIT OF 1
SETPR:  INC     HL
        LD      A,(HL)          :GET A CHARACTER
        OR      C               :SET PARITY BIT TO 1 AND TEST PARITY
        JP      PO,CHCNT        :IS PARITY NOW EVEN?
        LD      (HL),A          :YES. SAVE CHARACTER WITH EVEN PARITY
CHCNT:  DJNZ    SETPR
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	21		LD	HL,40H
0001	40			
0002	00			
0003	46		LD	B,(HL)
0004	0E		LD	C,10000000H
0005	80			
0006	23	SETPR:	INC	HL
0007	7E		LD	A,(HL)
0008	B1		OR	C
0009	E2		JP	PO,CHCNT
000A	0D			
000B	00			
000C	77		LD	(HL),A
000D	10	CHCNT:	DJNZ	SETPR
000E	F7			
000F	76		HALT	

Parity is often added to ASCII characters before they are transmitted on noisy communication lines, to provide a simple error-checking facility. Parity detects all single-bit errors but does not allow error correction (i.e., you know that an error has occurred when the received parity is wrong, but you cannot tell which bit was changed).

LD C,10000000B saves a parity bit of 1 in Register C. (Note the use of the binary mask: the purpose of the mask is clearer when it is specified in this manner rather than as 80H or 128 decimal.)

The instruction OR C sets the parity (most significant) bit to 1 while retaining all the other bits as they were, as well as setting the Z80 Parity flag.

The following procedure is used to determine if the parity of the byte in memory is odd or even. We OR a parity bit into the byte loaded from memory and then test to see if the parity is odd. If the parity is odd, then the byte in memory has even parity, and we jump down to decrement the count of remaining bytes. If the parity is even, then we know that the byte in memory has odd parity, and therfore we store the byte in the Accumulator into that memory location.

The conditional jumps JP PO (Jump on Parity Odd) and JP PE (Jump on Parity Even) are seldom used except in parity generation and checking. Note that there are no **relative** jumps conditional on the value of the Parity bit, just as there are none conditional on the value of the Sign bit.

Do not confuse the Parity bit included in each character and the Z80's Parity flag, which is set to 1 if the last arithmetic or Boolean result had even parity.

An alternative approach uses the Z80 SET instruction. This version takes a little longer but does not require a temporary register for the parity bit.

Source Program:

```
        LD      HL.40H
        LD      B.(HL)          :GET STRING LENGTH
SETPR:  INC     HL
        LD      A.(HL)          :GET A CHARACTER
        OR      A               :DOES CHARACTER HAVE EVEN PARITY?
        JP      PE.CHCNT
        SET     7.(HL)          :NO. SET PARITY BIT TO 1
CHCNT:  DJNZ    SETPR
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	21		LD	HL.40H
0001	40			
0002	00			
0003	46		LD	B.(HL)
0004	23	SETPR:	INC	HL
0005	7E		LD	A.(HL)
0006	B7		OR	A
0007	EA		JP	PE.CHCNT
0008	0C			
0009	00			
000A	CB		SET	7.(HL)
000B	FE			
000C	10	CHCNT:	DJNZ	SETPR
000D	F6			
000E	76		HALT	

Pattern Match

Purpose: Compare two strings of ASCII characters to see if they are the same. The length of the strings is in memory location 0041; one string starts in memory location 0042 and the other in memory location 0052. If the two strings match. clear memory location 0040; otherwise. set memory location 0040 to FF hex (all ones).

Sample Problems:

a.
```
        (0041)  =  03

        (0042)  =  43    'C'
        (0043)  =  41    'A'
        (0044)  =  54    'T'

        (0052)  =  43    'C'
        (0053)  =  41    'A'
        (0054)  =  54    'T'
```

Result: (0040) = 00.   since the two strings are the same.

b.
$$(0041) = 03$$

$$(0042) = 52 \quad 'R'$$
$$(0043) = 41 \quad 'A'$$
$$(0044) = 54 \quad 'T'$$

$$(0052) = 43 \quad 'C'$$
$$(0053) = 41 \quad 'A'$$
$$(0054) = 54 \quad 'T'$$

Result: $(0040) = FF$, since the first characters in the strings differ.

Note: The matching process ends as soon as the CPU finds a difference — the rest of the strings need not be examined.

Flowchart:
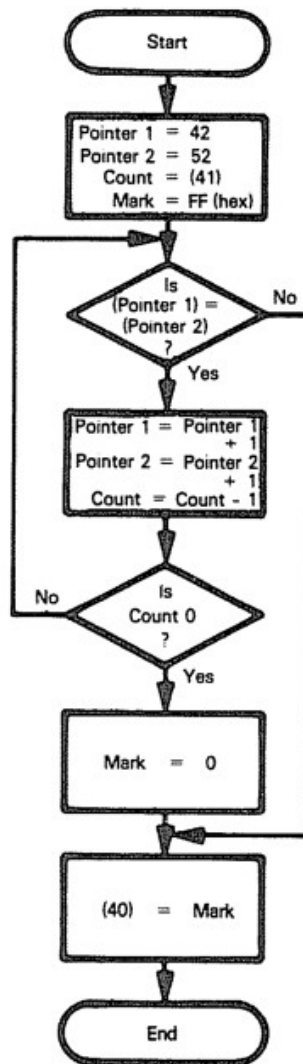
Source Program:

```
        LD      HL,41H
        LD      B,(HL)       :COUNT = LENGTH OF STRINGS
        INC     HL           :POINTER 1 = START OF STRING 1
        LD      DE,52H       :POINTER 2 = START OF STRING 2
        LD      C,0FFH       :MARK = FF (HEX)
CHCAR:  LD      A,(DE)       :GET CHARACTER FROM STRING 2
        CP      (HL)         :IS THERE A MATCH?
        JR      NZ,DONE       NO, DONE
        INC     DE
        INC     HL
        DJNZ    CHCAR        :CHECK NEXT PAIR IF ANY LEFT
        LD      C,0          :MARK = 0 IF ALL CHARACTERS MATCH
DONE:   LD      A,C
        LD      (40H),A      :SAVE MARK
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	46		LD	B,(HL)
0004	23		INC	HL
0005	11		LD	DE,52H
0006	52			
0007	00			
0008	0E		LD	C,0FFH
0009	FF			
000A	1A	CHCAR:	LD	A,(DE)
000B	BE		CP	(HL)
000C	20		JR	NZ,DONE
000D	06			
000E	13		INC	DE
000F	23		INC	HL
0010	10		DJNZ	CHCAR
0011	F8			
0012	0E		LD	C,0
0013	00			
0014	79	DONE:	LD	A,C
0015	32		LD	(40H),A
0016	40			
0017	00			
0018	76		HALT	

Matching strings of ASCII characters is an essential part of looking for commands, recognizing names, identifying variables or operation codes in assemblers and compilers, finding files, and many other tasks.

The program uses two pointers, one in Register Pair HL and the other in Register Pair DE. The only instructions that use the address in DE are LD A,(DE) (Load Accumulator From Memory Location Addressed by DE) and LD (DE),A (Store Accumulator in Memory Location Addressed by DE). Arithmetic and logical operations with memory and transfers to or from other registers (e.g., ADD A,(HL); AND (HL); LD B,(HL); LD (HL),E) can only be performed using the address in Register Pair HL, or using an index register.

The order of operations is very important because of the small number of instructions that use the address in Register Pair DE. You must move a character from the string pointed to by DE to the Accumulator and compare it to a character in the string pointed to by HL. This order of operations is necessary because the Z80 has no instruction which allows a comparison to a character in a string pointed to by DE.

For example, if you replaced LD A,(DE) with LD A,(HL), what would the next instruction be? This asymmetry is peculiar to the Z80 and can cause programming nightmares.

Note that each iteration updates both pointers.

This program could take advantage of the fact that a register is known to contain zero after a particular conditional jump is executed. When the DJNZ CHCAR instruction is executed, if the branch is not performed, then we know that Register B contains zero. Therefore, we can move Register B to Register C, our flag register, to indicate that a match has been found.

We could also use the Z80's SET and RESET instructions to handle the flag if we needed to conserve bits for other purposes.

PROBLEMS

1) Length of a Teletypewriter Message

Purpose: Determine the length of an ASCII message. All characters are 7-bit ASCII with MSB = 0. The string of characters in which the message is embedded starts in memory location 0041. The message itself starts with an ASCII STX character (hex 02) and ends with ETX (hex 03). Place the length of the message (the number of characters between the STX and the ETX but including neither) into memory location 0040.

Sample Problem:

(0041)	=	40	
(0042)	=	02	STX
(0043)	=	47	'G'
(0044)	=	4F	'O'
(0045)	=	03	ETX

Result: (0040) = 02. since there are two characters between the STX in location 0042 and ETX in location 0045.

2) Find Last Non-Blank Character

Purpose: Search a string of ASCII characters for the last non-blank character. The string starts in memory location 0042 and ends with a carriage return character (hex 0D). Place the address of the last non-blank character into memory locations 0040 and 0041 (most significant bits in 0041).

Sample Problems:

a.

(0042)	=	37	'7'
(0043)	=	0D	CR

Result: (0040) = 42. since the last (and only) non-blank character is in memory location 0042.

(0041) = 00

b.

```
(0042) = 41   'A'
(0043) = 20   SP
(0044) = 48   'H'
(0045) = 41   'A'
(0046) = 54   'T'
(0047) = 20   SP
(0048) = 20   SP
(0049) = 0D   CR
```

Result: (0040) = 46
          (0041) = 00

3) Truncate Decimal String to Integer Form

Purpose: Edit a string of ASCII decimal characters by replacing all digits to the right of the decimal point with ASCII blanks (hex 20). The string starts in memory location 0041 and is assumed to consist entirely of ASCII-coded decimal digits and a possible decimal point (hex 2E). The length of the string is in memory location 0040. If no decimal point appears in the string, assume that the decimal point is implicitly at the far right.

Sample Problems:

a.

```
(0040) = 04

(0041) = 37   '7'
(0042) = 2E   '.'
(0043) = 38   '8'
(0044) = 31   '1'
```

Result: (0041) = 37   '7'
          (0042) = 2E   '.'
          (0043) = 20   SP
          (0044) = 20   SP

b.

```
(0040) = 03

(0041) = 26   '6'
(0042) = 37   '7'
(0043) = 31   '1'
```

Result: Unchanged, as number is assumed to be 671.

4) Check Even Parity in ASCII Characters

Purpose: Check even parity in a string of ASCII characters. The length of the string is in memory location 0041, and the string itself begins in memory location 0042. If the parity of all the characters in the string is correct, clear memory location 0040; otherwise, place FF hex (all ones) into memory location 0040.

Sample Problems:

a.   (0041) = 03

                     (0042) = B1
                     (0043) = B2
                     (0044) = 33

    Result: (0040) = 00, since all the characters have even parity.

b.   (0041) = 03

                     (0042) = B1
                     (0043) = B6
                     (0044) = 33

    Result: (0040) = FF   since the character in memory
                         location 0042 does not have even parity.

5) String Comparison

Purpose: Compare two strings of ASCII characters to see which is larger (i.e., which follows the other in 'alphabetical' ordering). The length of the strings is in memory location 0041; one string starts in memory location 0042 and the other in memory location 0052. If the string starting in memory location 0042 is greater than or equal to the other string, clear memory location 0040; otherwise, set memory location 0040 to FF hex (all ones).

Sample Problems:

a.   (0041) = 03

                     (0042) = 43 'C'
                     (0043) = 41 'A'
                     (0044) = 54 'T'

                     (0052) = 42 'B'
                     (0053) = 41 'A'
                     (0054) = 54 'T'

    Result: (0040) = 00. since CAT is 'larger' than BAT.

b.   (0041) = 03

                     (0042) = 43 'C'
                     (0043) = 41 'A'
                     (0044) = 54 'T'

                     (0052) = 43 'C'
                     (0053) = 41 'A'
                     (0054) = 54 'T'

    Result: (0040) = 00, since the two strings are equal.

c.   (0041) = 03

                     (0042) = 43 'C'
                     (0043) = 41 'A'
                     (0044) = 54 'T'

                     (0052) = 43 'C'
                     (0053) = 55 'U'
                     (0054) = 54 'T'

    Result: (0040) = FF, since CUT is 'larger' than CAT.

Chapter 7
CODE CONVERSION

Code conversion is a continual problem in most microcomputer applications. Peripherals provide data in ASCII. BCD. or various special codes. The system must convert the data into some standard form for processing. Output devices may require data in ASCII. BCD. seven-segment, or other codes. Therefore, the system must convert the results to a suitable form after the processing is completed.

There are several ways to approach code conversion:

1) Some conversions can easily be handled by algorithms involving arithmetic or logical functions. The program may, however, have to handle some special cases separately.

2) More complex conversions can be handled with lookup tables. The lookup table method requires little programming and is easy to apply. However, the table may occupy a large amount of memory if the range of input values is large.

3) Hardware is readily available for some conversion tasks. Typical examples are decoders for BCD to seven-segment conversion and Universal Asynchronous Receiver/Transmitters (UARTs) for conversion between parallel (ASCII) and serial (teletypewriter) formats.

In most applications. the program should do as much as possible of the code conversion work. This results in a savings in parts and board space as well as in increased reliability. Furthermore. most code conversions are easy to program and require little execution time.

EXAMPLES

Hex to ASCII

Purpose: Convert the contents of memory location 0040 to an ASCII character. Memory location 0040 contains a single hexadecimal digit (the four most significant bits are zero). Store the ASCII character in memory location 0041.

Sample Problems:

a.   (0040) = 0C

      Result: (0041) = 43   'C'

b.   (0040) = 06

      Result: (0041) = 36   '6'

Flowchart:
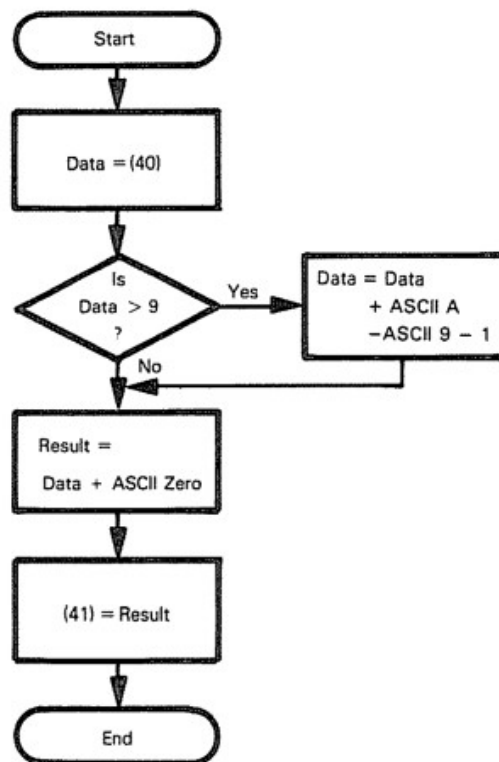
Source Program:

```
        LD      A,(40H)         ;GET DATA
        CP      10              ;IS DATA 10 OR MORE?
        JR      C,ASCZ
        ADD     A,'A'-'9'-1     ;YES, ADD OFFSET FOR LETTERS
ASCZ:   ADD     A,'0'           ;ADD OFFSET FOR ASCII
        LD      (4IH),A         ;STORE ASCII RESULT
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3A	LD	A,(40H)
0001	40		
0002	00		
0003	FE	CP	10
0004	0A		
0005	38	JR	C,ASCZ
0006	02		
0007	C6	ADD	A,'A'-'9'-1
0008	07		
0009	C6	ASCZ: ADD	A,'0'
000A	30		
000B	32	LD	(41H),A
000C	4I		
000D	00		
000E	76	HALT	

In this program, the basic idea is to add ASCII 0 to all the hexadecimal digits. This addition converts the decimal digits correctly; however, there is a break between ASCII 9 (39 hex) and ASCII A (41 hex) which must be considered. This break must be added to the nondecimal digits A, B, C, D, E, and F. This is accomplished by the ADD A instruction which adds the offset 'A'-'9'-1 to the contents of the Accumulator. Can you explain why the offset is 'A'-'9'-1?

Note that the addition terms are placed in the assembly language program in ASCII form (apostrophes surround an ASCII character or string of characters). The offset for the letters is left as an arithmetic expression. The effort is to make the purpose of the terms as clear as possible in the assembly language listing. The extra assembly time is a very small price to pay for a large increase in clarity.

This routine could be used in a variety of programs; for example, monitor programs must convert hexadecimal digits to ASCII in order to display the contents of memory locations in hexadecimal on an ASCII printer or video display.

Another (quicker) conversion method that requires no conditional jumps at all is the following program, described by Allison in Computer magazine.[1]

```
LD      A,(40H)      ;GET HEX DIGIT
ADD     A,90H        ;DEVELOP EXTRA 6 AND CARRY
DAA
ADC     A,40H        ;ADD IN CARRY, ASCII OFFSET
DAA
LD      (4IH),A      ;STORE ASCII DIGIT
HALT
```

Try this program on some digits. Can you explain why it works?

Decimal to Seven-Segment

Purpose: Convert the contents of memory location 0040 to a seven-segment code in memory location 0042. If memory location 0040 does not contain a single decimal digit, clear memory location 0042.

Seven-segment table: The following table can be used to convert decimal numbers to seven-segment code. The seven-segment code is organized with the most significant bit always zero followed by the code (1 = on, 0 = off) for segments g, f, e, d, c, b, and a (see Figure 7-1).

Digit	Code
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D
7	07
8	7F
9	6F
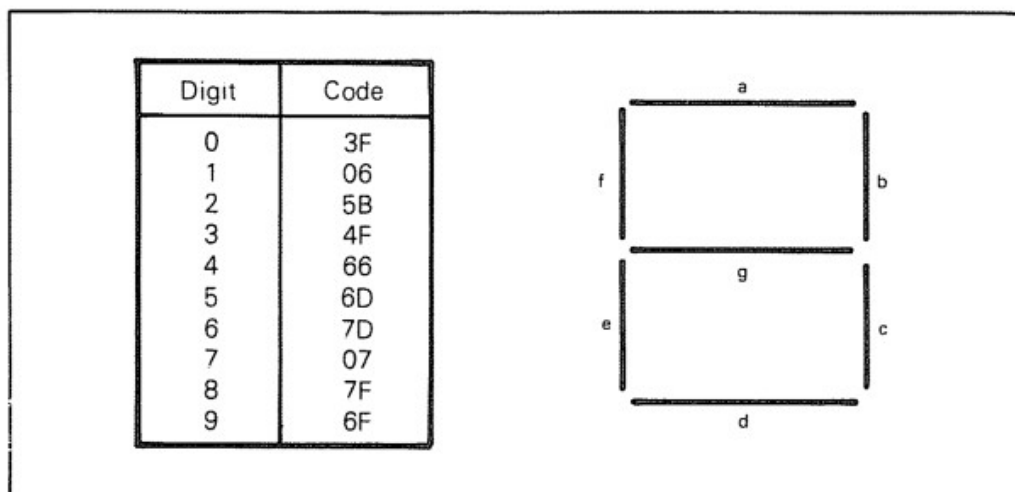
Figure 7-1. Seven-segment Arrangement

Note that the table uses 7D for 6 rather than the alternative 7C (top bar off) to avoid confusion with lower case b. and 6F for 9 rather than 67 (bottom bar off), for no particular reason.

Sample Problems:

a.   (0040) = 03

       Result: (0042) = 4F

b.   (0040) = 28

       Result: (0042) = 00

Flowchart:

```
                    ┌──────────────┐
                    │    Start      │
                    └──────────────┘
                           │
                    ┌──────────────┐
                    │  Data = (40)  │
                    └──────────────┘
                           │
                         ╱ Is ╲         Yes
                        ◇ Data > 9 ◇─────────┐
                         ╲   ?  ╱            │
                           │                 │
                          No                 │
                    ┌──────────────┐   ┌──────────────┐
                    │  Result =     │   │  Result = 0   │
                    │ (SSEG + Data) │   └──────────────┘
                    └──────────────┘         │
                           │◄────────────────┘
                    ┌──────────────┐
                    │ (42) = Result │
                    └──────────────┘
                           │
                    ┌──────────────┐
                    │     End       │
                    └──────────────┘
```
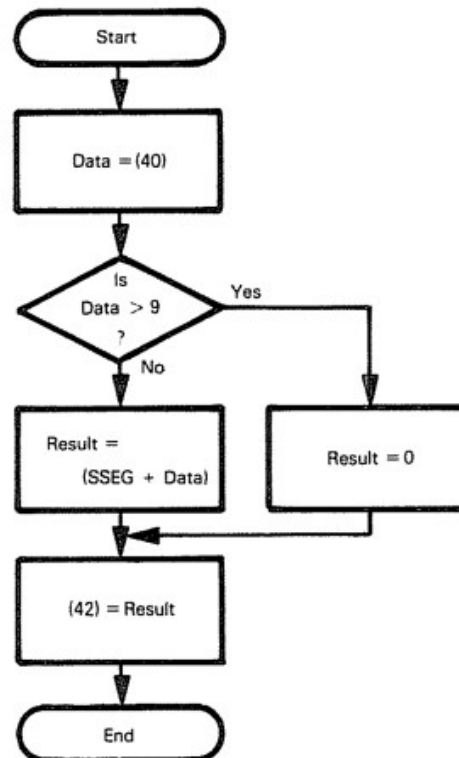
Note that the addition of base address SSEG and index (DATA) produces the address that contains the answer.

Source Program:

```
            LD      B,0          ;GET ERROR CODE TO BLANK DISPLAY
            LD      A,(40H)      ;GET DATA
            CP      10           ;IS DATA A DECIMAL DIGIT?
            JR      NC,DONE      ;NO, KEEP ERROR CODE
            LD      L,A          ;YES, MAKE DATA INTO A 16-BIT INDEX
            LD      H,0
            LD      DE,SSEG      ;GET BASE ADDRESS OF 7-SEGMENT TABLE
            ADD     HL,DE        ;FIND ELEMENT BY INDEXING
            LD      B,(HL)       ;GET 7-SEGMENT CODE FROM TABLE
DONE:       LD      A,B          ;SAVE 7-SEGMENT CODE OR ERROR CODE
            LD      (42H),A
            HALT
            ORG     20H          ;SEVEN-SEGMENT CODE TABLE
SSEG:       DEFB    3FH
            DEFB    06H
            DEFB    5BH
            DEFB    4FH
            DEFB    66H
            DEFB    6DH
            DEFB    7DH
            DEFB    07H
            DEFB    7FH
            DEFB    6FH
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	06		LD	B,0
0001	00			
0002	3A		LD	A,(40H)
0003	40			
0004	00			
0005	FE		CP	10
0006	0A			
0007	30		JR	NC,DONE
0008	08			
0009	6F		LD	L,A
000A	26		LD	H,0
000B	00			
000C	11		LD	DE,SSEG
000D	20			
000E	00			
000F	19		ADD	HL,DE
0010	46		LD	B,(HL)
0011	78	DONE:	LD	A,B
0012	32		LD	(42H),A
0013	42			
0014	00			
0015	76		HALT	
0020	3F	SSEG:	DEFB	3FH
0021	06		DEFB	06H
0022	5B		DEFB	5BH
0023	4F		DEFB	4FH
0024	66		DEFB	66H
0025	6D		DEFB	6DH
0026	7D		DEFB	7DH
0027	07		DEFB	07H
0028	7F		DEFB	7FH
0029	6F		DEFB	6FH

The program calculates the memory address of the desired code by adding the index (i.e., the digit to be displayed) to the base address of the seven-segment code table. This procedure is known as a table lookup.

The assembly language pseudo-operation DEFB (Define Byte) places constant data into program memory. Such data may include tables, headings, error messages, priming messages, format characters, thresholds, etc. The label attached to a DEFB pseudo-operation is assigned the value of the address into which the byte of data is placed.

Tables are often used to perform code conversions that are more complex than the previous example. Such tables typically contain all the results organized according to the input data, e.g., the first entry is the code corresponding to the number zero.

Seven-segment displays provide recognizable forms of the decimal digits and a few letters and other characters. Calculator-type seven-segment displays are inexpensive, easy to combine, and use little power. However, the seven-segment coded digits are somewhat difficult to read.

The assembler simply places the data for the table into memory. Note that one DEFB pseudo-operation fills one byte of memory. We have left some memory space between the program and the table to allow for later additions or corrections.

An alternative approach would be to use one of the Z80's index registers, say IX. The programmer must be aware of the following features of the Z80's index registers:

1) The fixed offset in program memory is only eight bits long and so cannot hold a complete memory address. It must be used either as a short displacement or to hold the eight least significant bits of a memory address.

2) The index registers are 16 bits long. Either IX or IY can be loaded from memory just like a register pair — from two consecutive memory addresses with the least significant eight bits at the lower address.

3) All operations involving the index registers take extra time and memory because one word of the operation code simply declares that an index register is to be used.

The following program uses Register IX to perform the table lookup:

Source Program:

```
        LD      B,0             :GET ERROR CODE TO BLANK DISPLAY
        LD      A,(40H)         :GET DATA
        CP      10              :IS DATA A DECIMAL DIGIT?
        JR      NC,DONE         :NO, KEEP ERROR CODE
        LD      HL,41H          :SAVE TABLE PAGE NUMBER IN MEMORY
        LD      (HL),0
        LD      IX,(40H)        :GET TABLE OFFSET
        LD      B,(IX+SSEG)     :GET 7-SEGMENT CODE FROM TABLE
DONE.   LD      A,B             :SAVE 7-SEGMENT CODE OR ERROR CODE
        LD      (42H),A
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	06	LD	B,0
0001	00		
0002	3A	LD	A,(40H)
0003	40		
0004	00		
0005	FE	CP	10
0006	0A		
0007	30	JR	NC,DONE
0008	0C		
0009	21	LD	HL,41H
000A	41		
000B	00		
000C	36	LD	(HL),0
000D	00		
000E	DD	LD	IX,(40H)
000F	2A		
0010	40		
0011	00		
0012	DD	LD	B,(IX+SSEG)
0013	46		
0014	20		
0015	78	DONE. LD	A,B
0016	32	LD	(42H),A
0017	42		
0018	00		
0019	76	HALT	

The indexed load instruction LD B,(IX + SSEG) adds the index (i.e., the digit to be displayed) to the base of the seven-segment table to get the address of the desired code. Note that the 16-bit index register contains the data as its eight least significant bits and the most significant bits of the starting address of the table as its eight most significant bits. This odd arrangement is necessary because the offset included with the indexed instruction is only eight bits long and can therefore hold only the eight least significant bits of the starting address of the table.

A more general program would allow the table to be placed anywhere in memory. If the table starting address is SSEGM (eight MSBs) and SSEGL (eight LSBs), the instruction LD (HL),0 must be replaced by LD (HL),SSEGM. Why is this change necessary?

Note that all operations involving Index Register IX have a 2-word operation code in which the first word is DD.

Clearly this is not a very efficient use of the index registers. These registers really become useful when you must access several data in a block. The block might contain the characteristics of a message, the parameters of an equation, the current state of a process or machine, or the data for a video display. You could, for example, take the contents of the twelfth location in the block and move them to the twentieth location with either of the following programs, assuming that the starting address of the block is stored in memory locations PTR and PTR+1.

MOVING DATA WITHIN A BLOCK

1) Using DE and HL.

```
        LD      DE,(PTR)        ;GET STARTING ADDRESS
        LD      HL,12           ;CALCULATE SOURCE ADDRESS
        ADD     HL,DE
        LD      A,(HL)          ;GET DATA FROM SOURCE
        LD      HL,20           ;CALCULATE DESTINATION ADDRESS
        ADD     HL,DE
        LD      (HL),A          ;MOVE DATA TO DESTINATION
```

2) Using IX.

```
        LD      IX,(PTR)        ;GET STARTING ADDRESS
        LD      A,(IX+12)       ;GET DATA FROM SOURCE
        LD      (IX+20),A       ;MOVE DATA TO DESTINATION
```

The program using the index registers is far shorter and clearer. Its only limitation is that the offsets must be small enough to fit into an 8-bit byte.

ASCII to Decimal

Purpose: Convert the contents of memory location 0040 from an ASCII character to a decimal digit and store the result in memory location 0041. If the contents of memory location 0040 are not the ASCII representation of a decimal digit, set the contents of memory location 0041 to FF (hex).

Sample Problems:

a.   (0040) = 37 '7'

            Result: (0041) = 07

b.   (0040) = 55

            Result: (0041) = FF

Flowchart:
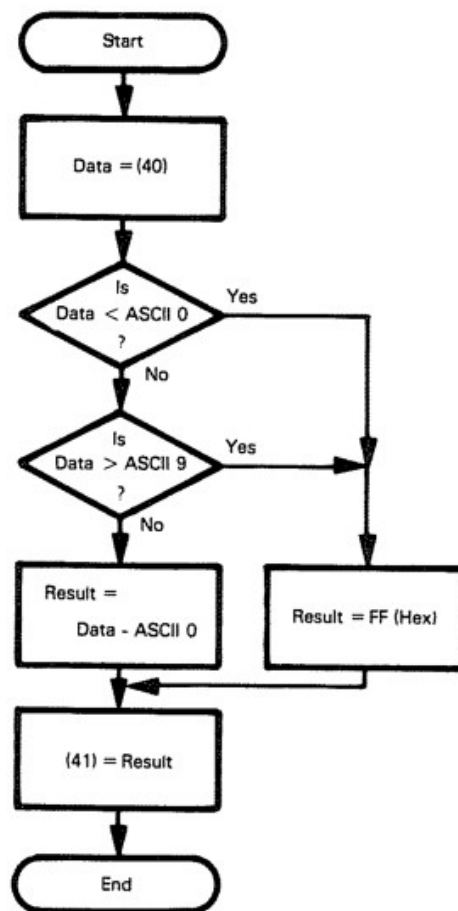
Source Program:

```
        LD      B,0FFH      :GET ERROR MARKER
        LD      A,(40H)     :GET DATA
        SUB     '0'         :IS DATA BELOW ASCII ZERO?
        JR      C.DONE      :YES, NOT A DIGIT
        CP      '9'+1       :IS DATA ABOVE ASCII NINE
        JR      NC.DONE     :YES, NOT A DIGIT
        LD      B,A         :SAVE DIGIT IF VALID
DONE:   LD      A,B         :SAVE DIGIT OR ERROR MARKER
        LD      (41H),A
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	06	LD	B,0FFH
0001	FF		
0002	3A	LD	A,(40H)
0003	40		
0004	00		
0005	D6	SUB	'0'
0006	30		
0007	38	JR	C,DONE
0008	05		
0009	FE	CP	'9'+1
000A	3A		
000B	30	JR	NC,DONE
000C	01		
000D	47	LD	B,A
000E	78	DONE: LD	A,B
000F	32	LD	(41H),A
0010	41		
0011	00		
0012	76	HALT	

This program handles ASCII-coded characters just like ordinary numbers. Note that the decimal digits and the letters form groups of consecutive codes. Strings of letters (like names) can be alphabetized by placing their ASCII representations in increasing numerical order (ASCII B = ASCII A + 1 for example).

Subtracting ASCII zero (30 hex) from any ASCII decimal digit gives the BCD representation of that digit.

ASCII to decimal conversion is necessary when decimal numbers are being entered from an ASCII device like a teletypewriter or video terminal.

The basic idea of the program is to determine if the character is between ASCII 0 and ASCII 9, inclusive. If the character is, it's an ASCII decimal digit, since the digits form a sequence. It may then be converted to decimal simply by subtracting hex 30 (ASCII 0), e.g., ASCII 7 - ASCII 0 = 37-30 = 7.

Note that one comparison is done with an actual subtraction (SUB '0') since the subtraction is necessary to convert ASCII to decimal. The other comparison is done with an implied subtraction (CP '9'+1) since the final result is now in the Accumulator if the original number was valid.

BCD to Binary

Purpose: Convert two BCD digits in memory locations 0040 and 0041 to a binary number in memory location 0042. The most significant BCD digit is in memory location 0040.

Sample Problems:

a.   (0040) = 02
                (0041) = 09

    Result: (0042) = 1D (hex) = 29 (decimal)

b.   (0040) = 07
                (0041) = 01

    Result: (0042) = 47 (hex) = 71 (decimal)

Note: No flowchart is included since the program multiplies the most significant digit by 10 simply by using the formula $10x = 8x + 2x$. Multiplying by 2 requires one arithmetic left shift and multiplying by 8 requires three such shifts.

Source Program:

```
LD      HL,40H      :GET MOST SIGNIFICANT DIGIT (MSD)
LD      A,(HL)
ADD     A,A         ;MSD TIMES TWO
LD      B,A         ;SAVE MSD TIMES TWO
ADD     A,A         ;MSD TIMES FOUR
ADD     A,A         ;MSD TIMES EIGHT
ADD     A,B         ;MSD TIMES TEN
INC     HL          ;POINT TO LEAST SIGNIFICANT DIGIT
ADD     A,(HL)      ;ADD TO FORM BINARY EQUIVALENT
INC     HL
LD      (HL),A      :STORE BINARY EQUIVALENT
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	87	ADD	A,A
0005	47	LD	B,A
0006	87	ADD	A,A
0007	87	ADD	A,A
0008	80	ADD	A,B
0009	23	INC	HL
000A	86	ADD	A,(HL)
000B	23	INC	HL
000C	77	LD	(HL),A
000D	76	HALT	

BCD entries are converted to binary in order to save on storage and to simplify calculations. However, the conversion may offset some of the advantages of binary storage and arithmetic.

This program multiplies the BCD digit in memory location 0040 by ten using repeated additions.[2] Note that ADD A,A multiplies the contents of the Accumulator by 2. This allows you to multiply the contents of the Accumulator by small decimal numbers in a few instructions. How would you use this procedure to multiply by 16? by 12? by 7?

BCD numbers require about 20% more storage than do binary numbers. Representing 0 to 999 requires 12 bits in BCD form but only 10 bits in binary (since $2^{10} = 1024 \approx 1000$).

Convert Binary Number to ASCII String

Purpose: Convert the 8-bit binary number in memory location 0041 to eight ASCII characters (either ASCII 0 or ASCII 1) in memory locations 0042 through 0049 (the most significant bit is in 0042).

Sample Problem:

$$(0041) = D2 = 11010010$$

Result:
(0042)	= 31	'1'
(0043)	= 31	'1'
(0044)	= 30	'0'
(0045)	= 31	'1'
(0046)	= 30	'0'
(0047)	= 30	'0'
(0048)	= 31	'1'
(0049)	= 30	'0'

Flowchart:
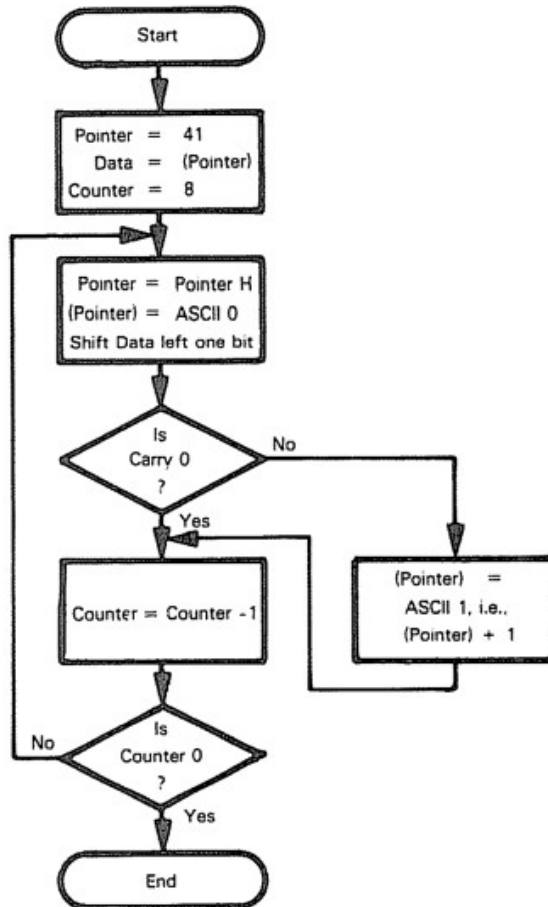
Source Program:

```
        LD      HL,41H
        LD      A,(HL)          ;GET DATA
        LD      B,8             ;COUNTER = NUMBER OF BITS IN WORD
        LD      C,'0'           ;GET ASCII ZERO TO STORE IN STRING
CONV:   INC     HL
        LD      (HL),C          ;PUT ASCII ZERO IN STRING
        RLA                     ;IS NEXT BIT OF DATA 1?
        JR      NC,COUNT        ;YES, MAKE STRING ELEMENT ASCII ONE
        INC     (HL)
COUNT:  DJNZ    CONV
        HALT
```

7-12

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	7E		LD	A,(HL)
0004	06		LD	B,8
0005	08			
0006	0E		LD	C,'0'
0007	30			
0008	23	CONV:	INC	HL
0009	71		LD	(HL),C
000A	17		RLA	
000B	30		JR	NC,COUNT
000C	01			
000D	34		INC	(HL)
000E	10	COUNT:	DJNZ	CONV
000F	F8			
0010	76		HALT	

The ASCII digits form a sequence so ASCII 1 = ASCII 0+1. Remember that the Z80 registers have special uses. You should place the loop counter into Register B so that you can use the DJNZ instruction.

Be careful of the difference between INC HL, which adds one to the 16-bit contents of Register Pair HL, and INC (HL), which adds one to the 8-bit contents of the memory location addressed by Register Pair HL.

Binary-to-ASCII conversion is necessary when numbers are printed in binary form on an ASCII device.

The conversion to ASCII simply involves adding ASCII 0 (hex 30).

PROBLEMS

1) ASCII to Hex

Purpose: Convert the contents of memory location 0040 to a hexadecimal digit and store the result in memory location 0041. Assume that memory location 0040 contains the ASCII representation of a hexadecimal digit (7 bits with MSB 0).

Sample Problems:

a.   (0040) = 43 'C'

   Result: (0041) = 0C

b.   (0040) = 36 '6'

   Result: (0041) = 06

2) Seven-Segment to Decimal

Purpose: Convert the contents of memory location 0040 from a seven-segment code to a decimal number in memory location 0041. If memory location 0040 does not contain a valid seven-segment code, set memory location 0041 to FF (hex). Use the seven-segment table given under the Decimal to Seven-Segment example and try to match codes.

Sample Problems:

a.   (0040) = 4F

    Result: (0041) = 03

b.   (0040) = 28

    Result: (0041) = FF

3)   Decimal to ASCII

Purpose: Convert the contents of memory location 0040 from a decimal digit to an ASCII character and store the result in memory location 0041. If the number in memory location 0040 is not a decimal digit, set the contents of memory location 0041 to an ASCII blank character (20 hex).

Sample Problems:

a.   (0040) = 07

    Result: (0041) = 37   '7'

b.   • (0040) = 55

    Result: (0041) = 20   SP

4)   Binary to BCD

Purpose: Convert the contents of memory location 0040 to two BCD digits in memory locations 0041 and 0042 (most significant digit in 0041). The number in memory location 0040 is unsigned and less than 100.

Sample Problems:

a.   (0040) = 1D   (29 decimal)

    Result: (0041) = 02
            (0042) = 09

b.   (0040) = 47   (71 decimal)

    Result: (0041) = 07
            (0042) = 01

5)   ASCII String to Binary Number

Purpose: Convert the eight ASCII characters in memory locations 0042 through 0049 to an 8-bit binary number in memory location 0041 (the most significant bit is in 0042). Clear memory location 0040 if all the ASCII characters are either ASCII 1 or ASCII 0 and set it to FF otherwise.

Sample Problems:

a.   (0042) = 31   '1'
               (0043) = 31   '1'
               (0044) = 30   '0'
               (0045) = 31   '1'
               (0046) = 30   '0'
               (0047) = 30   '0'
               (0048) = 31   '1'
               (0049) = 30   '0'

    Result: (0041) = D2
            (0040) = 00

b. same as 'a' except:
               (0045) = 37   '7'

    Result: (0040) = FF

REFERENCES

1. Allison, D.R.. "A Design Philosophy for Microcomputer Architectures," Computer, February 1977, pp. 35-41. This is an excellent article which we recommend highly.

2. Other BCD-to-binary conversion methods are discussed in J.A. Tabb and M.L. Roginsky, "Microprocessor Algorithms Make BCD-Binary Conversions Super-fast," EDN. January 5, 1977. pp. 46-50 and in J.B. Peatman, Microcomputer-based Design, McGraw-Hill. New York, 1977. pp. 400-406.

Chapter 8
ARITHMETIC PROBLEMS

Most arithmetic in microprocessor applications consists of multiple-word binary or decimal manipulations. A decimal correction (decimal adjust) or some other means for performing decimal arithmetic is frequently the only arithmetic instruction provided besides basic addition and subtraction. You must implement other arithmetic operations with sequences of instructions.

Multiple-precision binary arithmetic requires simple repetitions of the basic single-word instructions. The Carry bit transfers information between words. Add with Carry and Subtract with Carry use the information from the previous arithmetic operations. You must be careful to clear the Carry before operating on the first words (obviously there is no carry into or borrow from the least significant bits).

Decimal arithmetic is a common enough task for microprocessors that most have special instructions for this purpose. These instructions may either perform decimal operations directly or correct the results of binary operations to the proper decimal form. Decimal arithmetic is essential in such applications as point-of-sale terminals, calculators, check processors, order entry systems, and banking terminals.

You can implement multiplication and division as series of additions and subtractions respectively, much as they are done by hand. Double-word operations are necessary since a multiplication produces a result twice as long as the operands, while a division similarly contracts the length of the result. Multiplications and divisions are time-consuming when done in software because of the repeated arithmetic and shift operations that are necessary. Of course, multiplying or dividing by a power of 2 is simple because such operations can be implemented with an appropriate number of left or right arithmetic shifts.

EXAMPLES
Multiple-Precision Addition

Purpose: Add two multiple-word binary numbers. The length of the numbers (in bytes) is in memory location 0040, the numbers themselves start (least significant bits first) in memory locations 0041 and 0051, respectively, and the sum replaces the number starting in memory location 0041.

Sample Problem:

(0040) = 04

(0041) = C3
(0042) = A7
(0043) = 5B
(0044) = 2F

(0051) = B8
(0052) = 35
(0053) = DF
(0054) = 14

Result: (0041) = 7B
(0042) = DD
(0043) = 3A
(0044) = 44

that is,

$$\begin{array}{r} 2F5BA7C3 \\ +\ \ 14DF35B8 \\ \hline 443ADD7B \end{array}$$

Flowchart:

```
              ┌──────────────┐
              │    Start     │
              └──────┬───────┘
                     │
      ┌──────────────────────────┐
      │ Count    =   (40)        │
      │ Pointer 1  =  41         │
      │ Pointer 2  =  51         │
      │ Carry    =   0           │
      └──────────┬───────────────┘
                 │
    ┌──────────┐ │
    │          ▼
    │  ┌──────────────────┐
    │  │ (Pointer 1) =    │
    │  │   (Pointer 1) +  │      (This step also produces a new Carry)
    │  │   (Pointer 2) +  │
    │  │   Carry          │
    │  └────────┬─────────┘
    │           │
    │  ┌──────────────────────┐
    │  │ Pointer 1= Pointer 1 │
    │  │         + 1          │
    │  │ Pointer 2= Pointer 2 │
    │  │         + 1          │
    │  │ Count  =  Count - 1  │
    │  └────────┬─────────────┘
    │           │
    │           ▼
    │        ╱────────╲
    │  No   ╱   Is     ╲
    └──────╱  Count 0   ╲
           ╲    ?      ╱
            ╲────────╱
                │ Yes
                ▼
         ┌──────────────┐
         │    End       │
         └──────────────┘
```
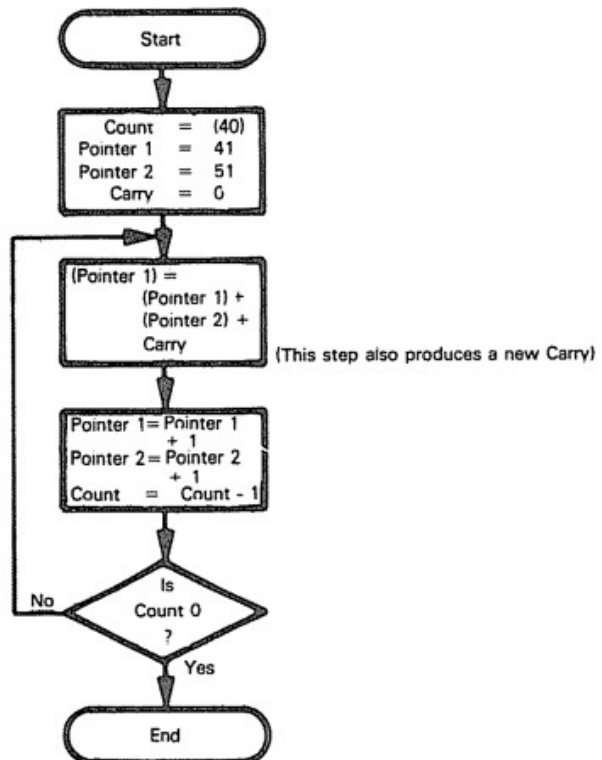
Source Program:

```
        LD    HL,40H    ;COUNT = LENGTH OF STRINGS (IN BYTES)
        LD    B,(HL)
        INC   HL        ;POINTER 1 = FIRST WORD OF STRING 1
        LD    DE,51H    ;POINTER 2 = FIRST WORD OF STRING 2
        AND   A         ;CLEAR CARRY TO START
ADDW:   LD    A,(DE)    ;GET WORD FROM STRING 2
        ADC   A,(HL)    ;ADD WORD FROM STRING 1
        LD    (HL),A    ;STORE RESULT IN STRING 1
        INC   DE
        INC   HL
        DJNZ  ADDW
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	46	LD	B,(HL)
0004	23	INC	HL
0005	11	LD	DE,51H
0006	51		
0007	00		
0008	A7	AND	A
0009	1A	ADDW: LD	A,(DE)
000A	8E	ADC	A,(HL)
000B	77	LD	(HL),A
000C	13	INC	DE
000D	23	INC	HL
000E	10	DJNZ	ADDW
000F	F9		
0010	76	HALT	

The relative address for DJNZ ADDW is:

$$\begin{array}{ccc} 09 & & 09 \\ -10 & = & +F0 \\ \hline & & F9 \end{array}$$

The instruction AND A is used to clear the Carry bit. Any other logical operation would have the same effect. The Carry must be cleared, since there is no carry involved in the addition of the least significant bytes.

The instruction ADC, Add with Carry, includes the Carry from the previous words in the addition. ADC is the only instruction in the loop that affects the Carry. Remember that neither INC nor DJNZ does.

Both the pointer in Register Pair DE and the one in HL must be updated during each iteration.

This procedure can add binary numbers of up to 256 bytes in length. Note that the ten binary bits correspond to three decimal digits, since $2^{10} = 1024 \simeq 1000$. So, you can calculate the number of bits required to give a certain accuracy in decimal digits. For example, ten decimal digit accuracy requires:

$$(10 \text{ digits}) \times \left(\frac{10 \text{ bits}}{3 \text{ digits}} \right) = 33 \text{ bits}$$

If we were only transferring the data from one place in memory to another and not also processing it, we could use the Z80's powerful block transfer instruction LDIR. This single instruction moves a byte of data from the address in HL to the address in

DE, increments the pointers in HL and DE, and decrements the byte counter in BC. It repeats the move operation until BC is decremented to zero. LDI is the same instruction without the repetition factor; LDD and LDDR are non-repeated and repeated moves, respectively, that decrement the pointers rather than incrementing them.

A program to transfer a fixed number of bytes (LENGTH) from one place in memory (starting at PTR1) to another place in memory (starting at PTR2) is the following.

Block Move

Purpose: Move a block of data BC characters long from the address in HL to the address in DE.

Sample Problem:

```
        (HL)   =  40
        (DE)   =  50
        (BC)   =   3

      (0040)   =  31
      (0041)   =  32
      (0042)   =  33

      (0050)   =   0
      (0051)   =   0
      (0052)   =   0

Result: (0050) =  31
      (0051)   =  32
      (0052)   =  33
```

Source Program:

```
    LD      BC,LENGTH   :COUNT = LENGTH OF TRANSFER (IN BYTES)
    LD      HL,PTR1     :POINTER 1 = START OF DATA SOURCE AREA
    LD      DE,PTR2     :POINTER 2 = START OF DATA DESTINATION
                        :   AREA

    LDIR
    HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	01	LD	BC,LENGTH
0001			
0002	LENGTH		
0003	21	LD	HL,PTR1
0004			
0005	PTR1		
0006	11	LD	DE,PTR2
0007			
0008	PTR2		
0009	ED	LDIR	
000A	B0		
000B	76	HALT	

Try to implement the same program without the LDIR instruction. How many bytes of memory and clock cycles does it require each way?

Decimal Addition

Purpose: Add two multiple-word decimal (BCD) numbers. The length of the numbers is in memory location 0040, the numbers themselves start (least significant bits first) in memory locations 0041 and 0051, respectively, and the sum replaces the number starting in memory location 0041.

Sample Problem:

$$(0040) = 04$$

$$(0041) = 85$$
$$(0042) = 19$$
$$(0043) = 70$$
$$(0044) = 36$$

$$(0051) = 59$$
$$(0052) = 34$$
$$(0053) = 66$$
$$(0054) = 12$$

Result: $(0041) = 44$
$(0042) = 54$
$(0043) = 36$
$(0044) = 49$

that is,

```
  36701985
+12663459
─────────
  49365444
```

Flowchart:

```
                    ┌──────────┐
                    │  Start   │
                    └────┬─────┘
                         │
                         ▼
              ┌────────────────────┐
              │ Count    =  (40)   │
              │ Pointer 1 =  41    │
              │ Pointer 2 =  51    │
              │ Carry    =   0     │
              └─────────┬──────────┘
                        │
         ┌──────────────▼──────────────┐
         │  (Pointer 1) =              │
         │    (Pointer 1) +            │        (This step also produces a new Carry)
         │    (Pointer 2) +            │
         │    Carry +                  │
         │    Decimal correction       │
         └──────────────┬──────────────┘
                        │
         ┌──────────────▼──────────────┐
         │  Pointer 1  =               │
         │      Pointer 1 + 1          │
         │  Pointer 2  =               │
         │      Pointer 2 + 1          │
         │  Count  =   Count - 1       │
         └──────────────┬──────────────┘
                        │
                        ▼
                  ╱───────────╲
      No         ╱     Is      ╲
  ◄─────────────┤   Count 0     │
                 ╲     ?       ╱
                  ╲───────────╱
                        │ Yes
                        ▼
                 ┌──────────────┐
                 │     End      │
                 └──────────────┘
```
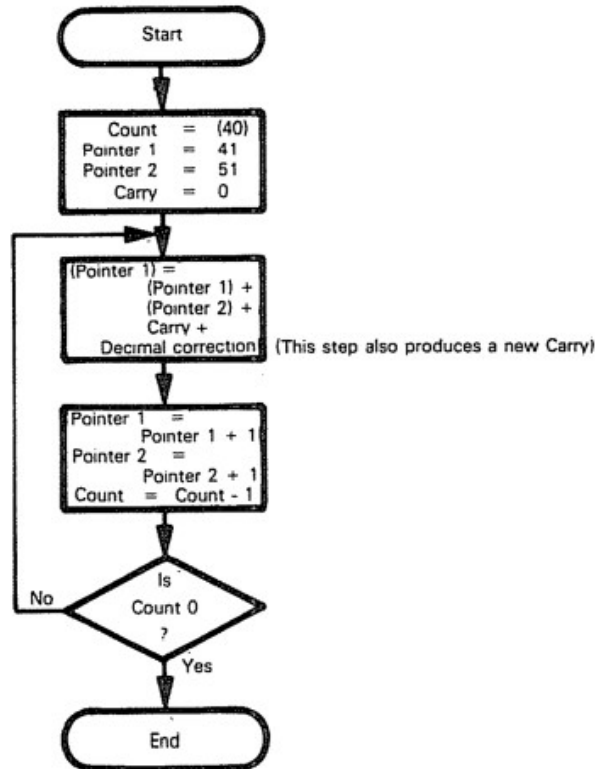
Source Program:

```
        LD      HL.40H
        LD      B,(HL)        ;COUNT = LENGTH OF STRINGS (IN BYTES)
        INC     HL            ;POINTER 1 = FIRST WORD OF STRING 1
        LD      DE,51H        ;POINTER 2 = FIRST WORD OF STRING 2
        AND     A             ;CLEAR CARRY TO START
DECAD:  LD      A,(DE)        ;GET 2 DECIMAL DIGITS FROM STRING 2
        ADC     A,(HL)        ;ADD PAIR OF DIGITS FROM STRING 1
        DAA                   ;MAKE ADDITION DECIMAL
        LD      (HL).A        ;STORE RESULT IN STRING 1
        INC     DE
        INC     HL
        DJNZ    DECAD
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	21		LD	HL,40H
0001	40			
0002	00			
0003	46		LD	B,(HL)
0004	23		INC	HL
0005	11		LD	DE,51H
0006	51			
0007	00			
0008	A7		AND	A
0009	1A	DECAD:	LD	A,(DE)
000A	8E		ADC	A,(HL)
000B	27		DAA	
000C	77		LD	(HL),A
000D	13		INC	DE
000E	23		INC	HL
000F	10		DJNZ	DECAD
0010	F8			
0011	76		HALT	

The Decimal Adjust instruction (DAA) uses the Carry (C) and Half Carry (H) bits to correct the following situations:

DECIMAL ADJUST

1) The sum of two digits is between 10 and 15, inclusive. In this case, six must be added to the sum to give the right result, i.e.

$$
\begin{array}{r}
0101 \quad (5) \\
+ \ 1000 \quad (8) \\
\hline
1101 \quad (D) \\
+ \ 0110 \\
\hline
0001 \ 0011 \quad (\text{BCD } 13, \text{ which is correct})
\end{array}
$$

2) The sum of two digits is 16 or more. In this case the result is a proper BCD number but six less than it should be, i.e.

$$
\begin{array}{r}
1000 \quad (8) \\
+ \ 1001 \quad (9) \\
\hline
0001 \ 0001 \quad (\text{BCD } 11) \\
+ \ 0110 \\
\hline
0001 \ 0111 \quad (\text{BCD } 17, \text{ which is correct})
\end{array}
$$

Six must be added in both situations. However, case 1 can be recognized by the fact that the sum is not a BCD digit, it is between 10 and 15 (or A and F hexadecimal). Case 2 can be recognized only by the fact that the Carry (most significant digit) or Half Carry (least significant digit) has been set to 1, since the result is a valid BCD number. DAA is the only instruction that uses the Half Carry. Note that DAA operates only on the Accumulator.

The Z80 microprocessor also has a flag that distinguishes between Add instructions (ADD, ADC) and Subtract instructions (SUB, SBC). This flag, called the Add/Subtract flag or N flag, is

ADD/SUBTRACT FLAG

cleared by all Add instructions and set by all Subtract instructions. The sole use of this flag is to allow the DAA instruction to correctly change binary addition into BCD addition and binary subtraction into BCD subtraction. The 8080 and 8085 microprocessors do not have an N flag, and so their DAA instructions operate properly only after addition.

DAA can be used only after instructions that place their result into the Accumulator and that properly affect the Carry, Half-Carry, and Add/Subtract flags. Thus, you cannot use DAA after INC (since INC does not affect the Carry), DEC, or any of the double-word instructions that place their results into the index registers or Register Pair HL.

This procedure can add decimal (BCD) numbers of any length. Here four binary bits are required for each decimal digit, so ten-digit accuracy requires:

<div style="border: 1px solid black; display: inline-block; padding: 2px;">
ACCURACY IN BINARY AND BCD
</div>

$$10 \times 4 = 40 \text{ bits}$$

as opposed to 33 bits in the binary case. This is essentially five 8-bit words instead of four. The decimal procedure also takes a little longer per word because of the extra DAA instruction.

8-Bit Binary Multiplication

Purpose: Multiply the 8-bit unsigned number in memory location 0040 by the 8-bit unsigned number in memory location 0041. Place the eight least significant bits of the result into memory location 0042 and the eight most significant bits into memory location 0043.

Sample Problems:

a.
  (0040) = 03
  (0041) = 05

  Result: (0042) = 0F
  (0043) = 00

  or in decimal 3 x 5 = 15

b.
  (0040) = 6F
  (0041) = 61

  Result: (0042) = 0F
  (0043) = 2A

  or 111 x 97 = 10,767

You can perform multiplication on a computer in the same way that you do long multiplication by hand. Since the numbers are binary, the only problem is whether to multiply by 0 or 1; multiplying by zero obviously gives zero as a result, while multiplying by one produces the same number that you started with (the multiplicand). So, each step in a binary multiplication can be reduced to the following operation.

If the current bit in the multiplier is 1, add the multiplicand to the partial product.

<div style="border: 1px solid black; display: inline-block; padding: 2px;">
MULTIPLICATION ALGORITHM
</div>

The only remaining problem is to ensure that you line everything up correctly each time. The following operations perform this task.

1) Shift multiplier left one bit so that the bit to be examined is placed into the Carry.

2) Shift product left one bit so that the next addition is lined up correctly.

The complete process for binary multiplication is as follows:[1]

Step 1 - Initialization
  Product = 0
  Counter = 8

Step 2 - Shift Product so as to line up properly
  Product = 2 x Product (LSB = 0)

Step 3 - Shift Multiplier so bit goes to Carry
  Multiplier = 2 x Multiplier

Step 4 - Add Multiplicand to Product if Carry is 1
        If Carry = 1, Product = Product + Multiplicand

Step 5 - Decrement Counter and check for zero
        Counter = Counter - 1
        If Counter ≠ 0 go to Step 2

In the case of Sample Problem b. where the multiplier is 61 (hex) and the multiplicand is 6F (hex) the process works as follows:

Initialization:

Product	0000
Multiplier	61
Multiplicand	6F
Counter	08

After first iteration of steps 2-5:

Product	0000
Multiplier	C2
Multiplicand	6F
Counter	07
Carry from Multiplier	0

After second iteration:

Product	006F
Multiplier	84
Multiplicand	6F
Counter	06
Carry from Multiplier	1

After third iteration:

Product	014D
Multiplier	08
Multiplicand	6F
Counter	05
Carry from Multiplier	1

After fourth iteration:

Product	029A
Multiplier	10
Multiplicand	6F
Counter	04
Carry from Multiplier	0

After fifth iteration:

Product	0534
Multiplier	20
Multiplicand	6F
Counter	03
Carry from Multiplier	0

After sixth iteration:

Product	0A68
Multiplier	40
Multiplicand	6F
Counter	02
Carry from Multiplier	0

After seventh iteration:

Product	14D0
Multiplier	80
Multiplicand	6F
Counter	01
Carry from Multiplier	0

After eighth iteration:

Product	2A0F
Multiplier	00
Multiplicand	6F
Counter	00
Carry from Multiplier	1

Flowchart:
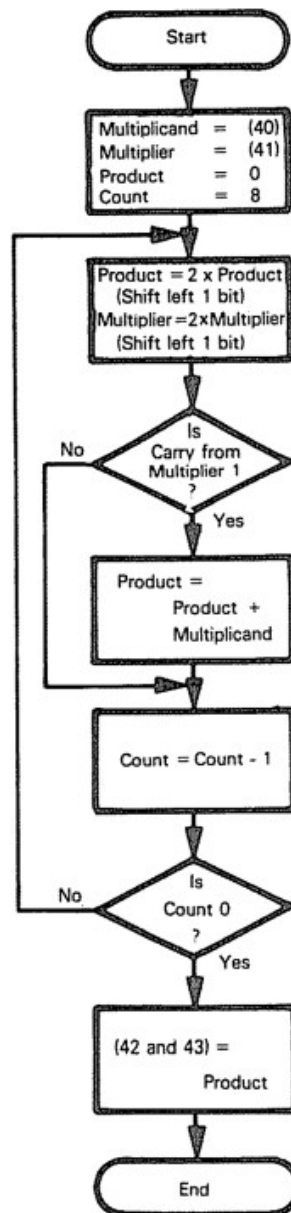
Source Program:

```
        LD      HL.40H
        LD      E.(HL)          ;GET MULTIPLICAND
        LD      D,0             ;EXTEND TO 16 BITS
        INC     HL
        LD      A,(HL)          ;GET MULTIPLIER
        LD      HL,0            ;PRODUCT = ZERO
        LD      B.8             ;COUNT = BIT LENGTH OF MULTIPLIER
MULT:   ADD     HL.HL           ;SHIFT PRODUCT LEFT 1 BIT
        RLA                     ;SHIFT MULTIPLIER LEFT 1 BIT
        JR      NC.CHCNT        ;IS CARRY FROM MULTIPLIER 1?
        ADD     HL.DE           ;YES. ADD MULTIPLICAND TO PRODUCT
CHCNT:  DJNZ    MULT
        LD      (42H),HL        ;SAVE PRODUCT IN MEMORY
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	21		LD	HL.40H
0001	40			
0002	00			
0003	5E		LD	E.(HL)
0004	16		LD	D.0
0005	00			
0006	23		INC	HL
0007	7E		LD	A.(HL)
0008	21		LD	HL.0
0009	00			
000A	00			
000B	06		LD	B.8
000C	08			
000D	29	MULT:	ADD	HL.HL
000E	17		RLA	
000F	30		JR	NC.CHCNT
0010	01			
0011	19		ADD	HL.DE
0012	10	CHCNT:	DJNZ	MULT
0013	F9			
0014	22		LD	(42H).HL
0015	42			
0016	00			
0017	76		HALT	

Note that the multiplicand must be extended to 16 bits by clearing Register D so that it can be added to the product using the ADD HL.DE instruction.

The instruction ADD HL.HL acts as a 16-bit logical left shift for the 16-bit product.

In this program, the Z80 16-bit instructions handle data rather than addresses. LD HL,0 is used to initialize the product; ADD HL.HL to perform a 16-bit logical left shift; ADD HL.DE to add the multiplicand to the partial product; and LD (42H).HL to store the result in memory. You must be careful to extend 8-bit quantities (like the multiplicand in this example) to 16 bits. Note that you cannot use the 16-bit facilities simultaneously for addressing and data manipulation. However, if you have no other need for the alternate registers, you could save the old contents of the regular registers there and restore

them afterward using the EXX instruction. This instruction exchanges the contents of Register Pairs BC. DE. and HL with the contents of their alternate counterparts in just four clock cycles.

Besides its obvious use in calculators and point-of-sale terminals. multiplication is a key part of almost all signal processing and control algorithms. The speed at which multiplications can be performed determines the usefulness of a CPU in process control. signal detection. and signal analysis.

The algorithm takes between 390 and 440 clock cycles to multiply on a Z80 microprocessor. The precise time depends on the number of one bits in the multiplier. Other algorithms may be able to reduce the average execution time somewhat, but 400 clock cycles will still be a typical execution time for a software multiplication.[2]

8-Bit Binary Division

Purpose: Divide the 16-bit unsigned number in memory locations 0040 and 0041 (most significant bits in 0041) by the 8-bit unsigned number in memory location 0042. The numbers are normalized so that 1) the most significant bits of both the dividend and the divisor are zero and 2) the number in memory location 0042 is greater than the number in memory location 0041, i.e., the quotient is an 8-bit number. Store the quotient in memory location 0043 and the remainder in location 0044.

Sample Problems:

a.
    (0040) = 40 (64 decimal)
    (0041) = 00
    (0042) = 08

    Result = (0043) = 08
            (0044) = 00
            i.e., 64/8 = 8

b.
    (0040) = 6D (12,909 decimal)
    (0041) = 32
    (0042) = 47 (71 decimal)

    Result = (0043) = B5 (181 decimal)
            (0044) = 3A (58 decimal)
            i.e.. 12.909/71 = 181 with a remainder of 58

You can perform division on the computer just like you would perform division with pen and paper, i.e., using trial subtractions. Since the numbers are binary, the only question is whether the bit

DIVISION ALGORITHM

in the quotient is 0 or 1. i.e., whether or not the divisor can be subtracted from what is left of the dividend. Each step in a binary division can be reduced to the following operation:

> If the divisor can be subtracted from the eight most significant bits of the dividend without a borrow, the corresponding bit in the quotient is 1: otherwise it is 0.

The only remaining problem is to line up the dividend and quotient properly. You can do this by shifting the dividend and quotient logically left one bit before each trial subtraction. The dividend and quotient can share a 16-bit register, since the procedure clears one bit of the dividend at the same time as it determines one bit of the quotient.

The complete process for binary division is:

Step 1 - Initialization:
Quotient = 0
Counter = 8

Step 2 - Shift Dividend and Quotient so as to line up properly:
Dividend = 2 x Quotient
Quotient = 2 x Quotient

Step 3 - Perform trial Subtraction. If no Borrow add 1 to Quotient:
If 8 MSBs of Dividend $\geq$ Divisor then
MSBs of Dividend = MSBs of Dividend - Divisor
Quotient = Quotient + 1

Step 4 - Decrement counter and check for zero:
Counter = Counter - 1
if Counter $\neq$ 0, go to Step 2
Remainder = 8 MSBs of Dividend

In the case of sample problem b, where the dividend is 326D (hex) and the divisor is 47 (hex), the process works as follows:

Initialization:

Dividend	326D
Divisor	47
Quotient	00
Counter	00

After first iteration of Steps 2 - 4:
(Note that the dividend is shifted prior to the trial subtraction)

Dividend	1DDA
Divisor	47
Quotient	01
Counter	07

After second iteration of Steps 2 - 4:

Dividend	3BB4
Divisor	47
Quotient	02
Counter	06

After third iteration:

Dividend	3068
Divisor	47
Quotient	05
Counter	05

After fourth iteration:

Dividend	19D0
Divisor	47
Quotient	0B
Counter	04

After fifth iteration:

Dividend	33A0
Divisor	47
Quotient	16
Counter	03

After sixth iteration:

Dividend	2040
Divisor	47
Quotient	2D
Counter	02

After seventh iteration:

Dividend	4080
Divisor	47
Quotient	5A
Counter	01

After eighth iteration:

Dividend	3A00
Divisor	47
Quotient	B5
Counter	00

So the quotient is B5 and the remainder is 3A.

The MSBs of dividend and divisor are assumed to be zero so as to simplify calculations (the shift prior to the trial subtraction would otherwise place the MSB of the dividend in the Carry). Problems that are not in this form must be simplified by removing parts of the quotient that would overflow an 8-bit word. For example:

$$\frac{1024}{3} = \frac{400 \text{ (Hex)}}{3} = 100 + \frac{100 \text{ (Hex)}}{3}$$

The last problem is now in the proper form. An extra division may be necessary.

Flowchart:

```
                    ┌──────────────┐
                    │    Start     │
                    └──────┬───────┘
                           │
                           ▼
            ┌──────────────────────────┐
            │ Dividend    =            │
            │             (40 and 41)  │
            │ Divisor     =   (42)     │
            │ Count       =   8        │
            │ Quotient    =   0        │
            └──────────┬───────────────┘
                       │
                       ▼
            ┌──────────────────────────┐
            │ Dividend    =            │
            │           2 x Dividend   │
            │ Quotient    =            │
            │           2 x Quotient   │
            │ (Shift both left 1 bit)  │
            └──────────┬───────────────┘
                       │
                       ▼
                    ╱───────╲
         Yes       ╱   Is    ╲
        ┌─────────╱  Divisor   ╲
        │         ╲  > 8 MSBs of ╲
        │          ╲  Dividend  ╱
        │           ╲    ?    ╱
        │            ╲───────╱
        │                │ No
        │                ▼
        │     ┌──────────────────────────┐
        │     │     8 MSBs of            │
        │     │  Dividend = 8 MSBs       │
        │     │  of Dividend - Divisor   │
        │     │ Quotient = Quotient + 1  │
        │     └──────────┬───────────────┘
        │                │
        └────────────────┤
                         ▼
              ┌──────────────────────┐
              │  Count = Count - 1   │
              └──────────┬───────────┘
                         │
                         ▼
                      ╱───────╲
         No          ╱   Is    ╲
        ┌───────────╱  Count 0  ╲
        │           ╲     ?     ╱
        │            ╲─────────╱
        │                │ Yes
        │                ▼
        │     ┌──────────────────────┐
        │     │  (43) = Quotient     │
        │     │  (44) = 8 MSBs of    │
        │     │         Dividend     │
        │     └──────────┬───────────┘
        │                │
        │                ▼
        │         ┌──────────────┐
        │         │     End      │
        │         └──────────────┘
```
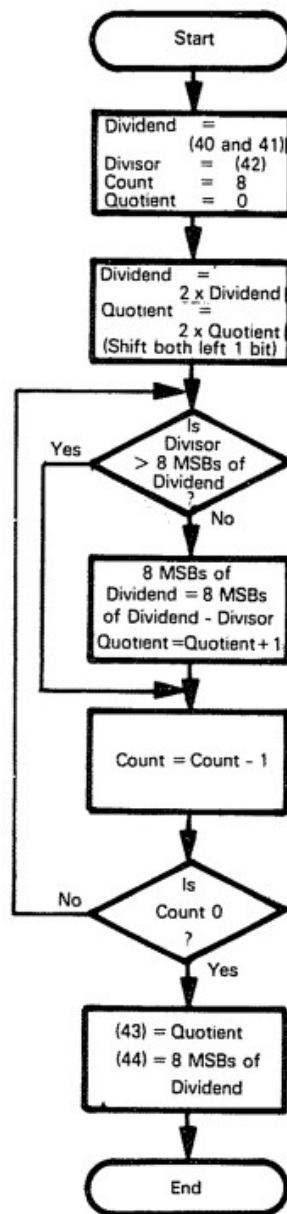
Source Program:

```
            LD      HL,(40H)     ;GET DIVIDEND
            LD      A,(42H)      ;GET DIVISOR
            LD      C,A
            LD      B,8          ;COUNT = NUMBER OF BITS IN DIVISOR
DIV:        ADD     HL,HL        ;SHIFT DIVIDEND, QUOTIENT LEFT 1 BIT
            LD      A,H          ;CAN DIVISOR BE SUBTRACTED?
            SUB     C
            JR      C,CNT        ;NO, GO TO NEXT STEP
            LD      H,A          ;YES, SUBTRACT DIVISOR FROM DIVIDEND
            INC     L            ;ADD 1 TO QUOTIENT
CNT:        DJNZ    DIV
            LD      (43H),HL     ;SAVE QUOTIENT. REMAINDER IN MEMORY
            HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	2A		LD	HL,(40H)
0001	40			
0002	00			
0003	3A		LD	A,(42H)
0004	42			
0005	00			
0006	4F		LD	C,A
0007	06		LD	B,8
0008	08			
0009	29	DIV:	ADD	HL,HL
000A	7C		LD	A,H
000B	91		SUB	C
000C	38		JR	C,CNT
000D	02			
000E	67		LD	H,A
000F	2C		INC	L
0010	10	CNT:	DJNZ	DIV
0011	F7			
0012	22		LD	(43H),HL
0013	43			
0014	00			
0015	76		HALT	

Register Pair HL holds both the dividend and the quotient. The quotient simply replaces the dividend in Register L as the dividend is shifted left logically.

For longer division problems, you could use the instruction SBC HL, which subtracts the contents of a register pair and the contents of the Carry from the contents of Register Pair HL.

The instruction INC L sets the least significant bit of the quotient to 1, since ADD HL,HL has previously cleared that bit.

Division is necessary in calculators, terminals, communications error checking, control algorithms, and many other applications.

This algorithm takes between 400 and 430 clock cycles to divide on a Z80 microprocessor. The precise time depends on the number of one bits in the quotient. Other algorithms may reduce the average execution time somewhat, but 400 clock cycles will still be typical for a software division. Some of the references listed at the end of this chapter discuss faster methods for implementing division.

Self-Checking Numbers
Double Add Double Mod 10

Purpose: Calculate a checksum digit from a string of BCD digits. The length of the string of digits (number of words) is in memory location 0041; the string of digits (2 BCD digits to a word) starts in memory location 0042. Calculate the checksum digit by the Double Add Double Mod 10 technique[3] and store it in memory location 0040.

The Double Add Double Mod 10 technique works as follows:

| SELF-CHECKING |
| NUMBERS |

1) Clear the checksum to start.

2) Multiply the leading digit by two and add the result to the checksum.

3) Add the next digit to the checksum.

4) Continue the alternating process until you have used all the digits.

5) The least significant digit of the checksum is the self-checking digit.

Self-checking digits are commonly added to identification numbers on credit cards, inventory tags, luggage, parcels, etc., when they are handled by computerized systems. They may also be used in routing messages, identifying files, and other applications. The purpose of the digits is to minimize entry errors such as transposing digits (69 instead of 96), shifting digits (7260 instead of 3726), missing digits by one (65 instead of 64), etc. You can check the self-checking number automatically for correctness upon entry and can eliminate many errors immediately.

The analysis of self-checking methods is quite complex. For example, a plain checksum will not find transposition errors $(4 + 9 = 9 + 4)$. The Double Add Double algorithm will find simple transposition errors $(2 \times 4 + 9 = 17 \neq 2 \times 9 + 4)$; but will miss some errors, such as transpositions across even numbers of digits (367 instead of 763). However, this method will find many common errors! The value of a method depends on what errors it will detect and on the probability of particular errors in an application.

For example, if the string of digits is

$$549321$$

the result will be:

$$\text{Checksum} = 5 \times 2 + 4 + 9 \times 2 + 3 + 2 \times 2 + 1 = 40$$
$$\text{Self-checking digit} = 0 \text{ (least significant digit of a checksum)}$$

Note that an erroneous entry like 543921 would produce a different self-checking digit (4), but erroneous entries like 049321 or 945321 would not be detected.

Sample Problems:

a.
$$\begin{aligned}
(0041) &= 03 \\
(0042) &= 36 \\
(0043) &= 68 \\
(0044) &= 51
\end{aligned}$$

Result: $\text{Checksum} = 3 \times 2 + 6 + 6 \times 2 + 8 + 5 \times 2 + 1 = 43$
$$(0040) = 03$$

b.
$$\begin{aligned}
(0041) &= 04 \\
(0042) &= 50 \\
(0043) &= 29 \\
(0044) &= 16 \\
(0045) &= 83
\end{aligned}$$

Result: $\text{Checksum} = 5 \times 2 + 0 + 2 \times 2 + 9 + 1 \times 2 + 6 + 8 \times 2 + 3 = 50$
$$(0040) = 00$$

Flowchart:

```
                    ┌──────────────┐
                    │    Start     │
                    └──────┬───────┘
                           │
                           ▼
                 ┌─────────────────────┐
                 │ Checksum  =   0     │
                 │ Count     =  (41)   │
                 │ Pointer   =   42    │
                 └─────────┬───────────┘
                           │
        ┌──────────────────▼──────────┐
        │      ┌─────────────────────┐
        │      │ MSD = (Pointer)/16  │
        │      │ LSD = Pointer AND   │
        │      │      00001111B      │
        │      │ Checksum =Checksum  │
        │      │  + 2 x MSD = LSD    │
        │      └─────────┬───────────┘
        │                │
        │                ▼
        │      ┌─────────────────────┐
        │      │ Pointer = Pointer +1│
        │      │ Count   = Count - 1 │
        │      └─────────┬───────────┘
        │                │
        │                ▼
        │              ╱─────╲
        │   No       ╱   Is    ╲
        └──────────◄   Count 0   ►
                    ╲     ?     ╱
                      ╲───────╱
                           │ Yes
                           ▼
                 ┌─────────────────────┐
                 │ (40) = Checksum     │
                 │  AND 00001111B      │
                 └─────────┬───────────┘
                           │
                           ▼
                    ┌──────────────┐
                    │     End      │
                    └──────────────┘
```

8-18
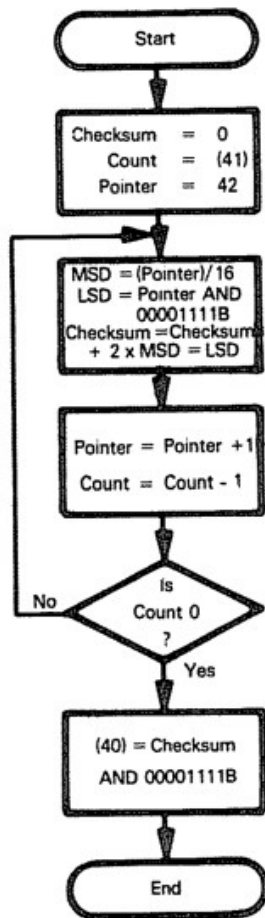
Source Program:

```
            LD      A.(41H)     :COUNT = LENGTH OF STRING IN BYTES
            LD      B,A
            LD      C,0         :CHECKSUM = 0
            LD      HL,42H      :POINT TO START OF STRING OF DIGITS
CHDIG:      LD      A,(HL)      :GET TWO BCD DIGITS FROM STRING
            LD      D,A         :SAVE COPY
            RRA                 :GET MSD BY SHIFTING AND MASKING
            RRA
            RRA
            RRA
            AND     00001111B
            ADD     A,A         :DOUBLE MSD
            DAA                 :MAKE DOUBLED MSD DECIMAL
            ADD     A,C         :ADD DOUBLED MSD TO CHECKSUM
            DAA                 :KEEP CHECKSUM DECIMAL
            LD      C,A
            LD      A,D         :GET LEAST SIGNIFICANT DIGIT
            AND     00001111B   :(MASK OUT MSD)
            ADD     A,C         :ADD LSD TO CHECKSUM
            DAA                 :KEEP CHECKSUM DECIMAL
            LD      C,A
            INC     HL
            DJNZ    CHDIG
            AND     00001111B   :MASK OFF SELF-CHECKING DIGIT
            LD      (40H),A     :SAVE SELF-CHECKING DIGIT
            HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	3A		LD	A,(41H)
0001	41			
0002	00			
0003	47		LD	B,A
0004	0E		LD	C,0
0005	00			
0006	21		LD	HL,42H
0007	42			
0008	00			
0009	7E	CHDIG:	LD	A,(HL)
000A	57		LD	D,A
000B	1F		RRA	
000C	1F		RRA	
000D	1F		RRA	
000E	1F		RRA	
000F	E6		AND	00001111B
0010	0F			
0011	87		ADD	A,A
0012	27		DAA	
0013	81		ADD	A,C
0014	27		DAA	
0015	4F		LD	C,A
0016	7A		LD	A,D
0017	E6		AND	00001111B
0018	0F			
0019	81		ADD	A,C
001A	27		DAA	
001B	4F		LD	C,A
001C	23		INC	HL
001D	10		DJNZ	CHDIG
001E	EA			
001F	E6		AND	00001111B
0020	0F			
0021	32		LD	(40H),A
0022	40			
0023	00			
0024	76		HALT	

The digits are removed by shifting and masking. Four right shifts are needed to separate out the most significant digit.

A decimal adjust (DAA) must follow each addition to produce the proper decimal result. A single DAA after a series of additions will not work (try it!). Remember that DAA works only on the Accumulator.

There is no problem with carries from the decimal sum. since the procedure uses only the least significant digit of the checksum anyway.

An alternative (and superior) approach is to use the Z80 decimal shift instruction RLD. This instruction is a 4-bit shift that moves the contents of the four least significant

bits of the memory location addressed by HL into the four most significant bits of that location, the previous contents of the four most significant bits of that location into the four least significant bits of the Accumulator, and the previous contents of the four least significant bits of the Accumulator into the four least significant bits of the memory location. Thus, RLD not only moves a single digit to the Accumulator, but it also shifts the next digit so that it can be moved to the Accumulator with the next RLD. Figure 8-1 shows an example of how RLD works; RRD is the same instruction except that the shift is right instead of left.

The Double Add Double Mod 10 algorithm can be implemented as follows using RLD:

Source Program:

```
        LD      A,(41H)     ;COUNT =LENGTH OF STRINGS (IN BYTES)
        LD      B,A
        LD      C,0         ;CHECKSUM = 0
        LD      HL,42H      ;POINT TO START OF STRING OF DIGITS
CHDIG:  SUB     A           ;CLEAR MSD
        RLD                 ;GET MSD FROM STRING
        ADD     A,A         ;DOUBLE MSD
        DAA                 ;MAKE DOUBLED MSD DECIMAL
        ADD     A,C         ;ADD DOUBLED MSD TO CHECKSUM
        DAA                 ;KEEP CHECKSUM DECIMAL
        LD      C,A
        SUB     A           ;CLEAR MSD
        RLD                 ;GET LSD FROM STRING
        ADD     A,C         ;ADD LSD TO CHECKSUM
        DAA                 ;KEEP CHECKSUM DECIMAL
        LD      C,A
        INC     HL
        DJNZ    CHDIG
        AND     00001111B   ;MASK OFF SELF-CHECKING DIGIT
        LD      (40H),A     ;SAVE SELF-CHECKING DIGIT
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	3A		LD	A,(41H)
0001	41			
0002	00			
0003	47		LD	B,A
0004	0E		LD	C,0
0005	00			
0006	21		LD	HL,42H
0007	42			
0008	00			
0009	97	CHDIG:	SUB	A
000A	ED		RLD	
000B	6F			
000C	87		ADD	A,A
000D	27		DAA	
000E	81		ADD	A,C
000F	27		DAA	
0010	4F		LD	C,A
0011	97		SUB	A
0012	ED		RLD	
0013	6F			
0014	81		ADD	A,C
0015	27		DAA	
0016	4F		LD	C,A
0017	23		INC	HL
0018	10		DJNZ	CHDIG
0019	EF			
001A	E6		AND	00001111B
001B	0F			
001C	32		LD	(40H),A
001D	40			
001E	00			
001F	76		HALT	

We could improve this program even further (it is already shorter than the previous version). Since we are dropping the most significant digit at the end anyway, there is no reason to clear it out each time with the SUB A instruction.

Initial Conditions

(HL) = 4000
(A) = 7F
(4000) = 12

After RLD

(A) = 71
(4000) = 2F

Before   After

| Accumulator | Memory | | Accumulator | Memory |

| 7 | F | | 1 | 2 | | 7 | 1 | | 2 | F |

High-order bit = 0, set S to 0
4 ones, set P/O to 1

Non-zero result,
set Z to 0

After RRD

(A) = 72
(4000) = F

Before   After

| Accumulator | Memory | | Accumulator | Memory |

| 7 | F | | 1 | 2 | | 7 | 2 | | F | 1 |

High-order bit = 0. set S to 0
4 ones, set P/O to 1

Non-zero result,
set Z to 0
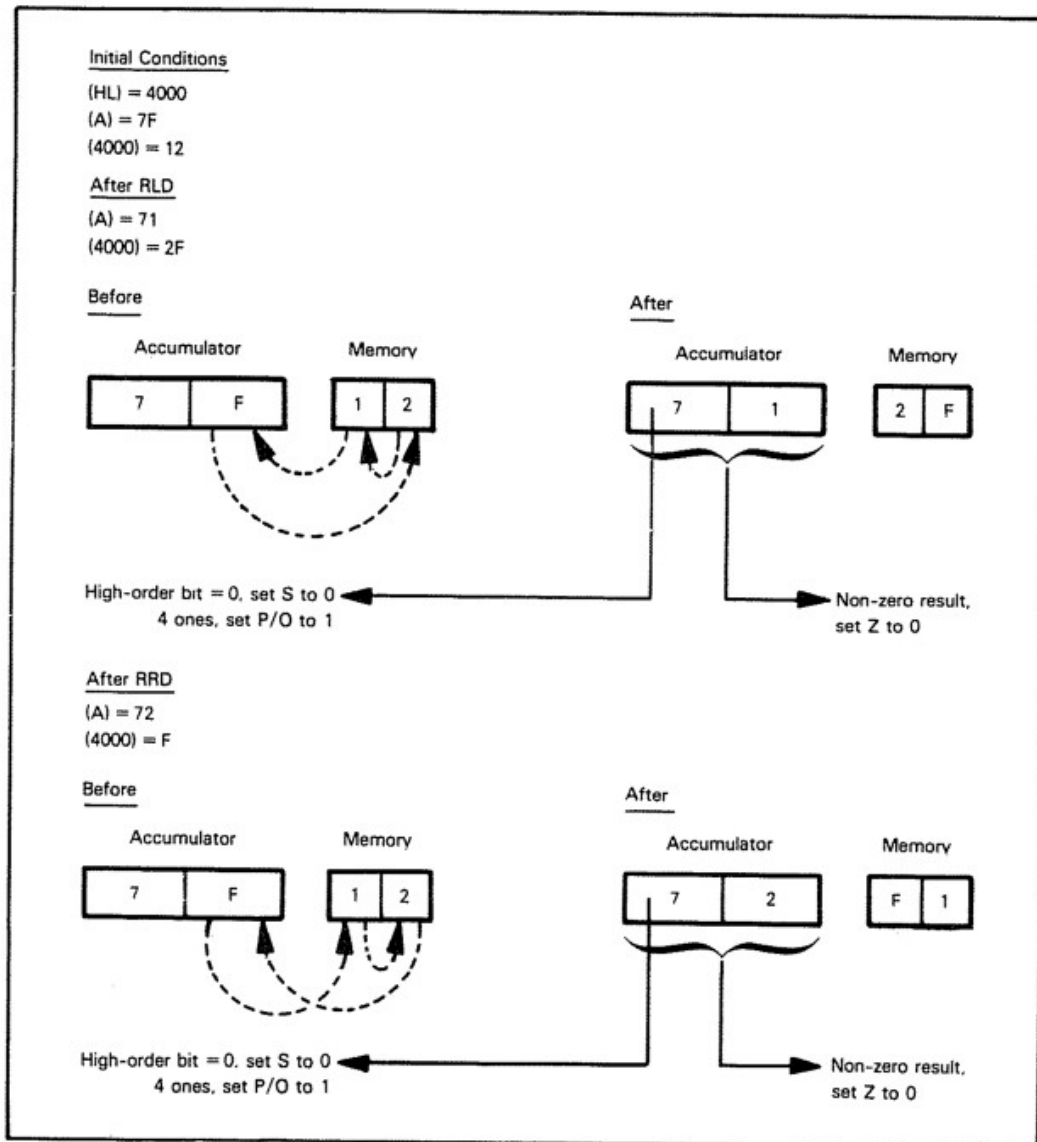
Figure 8-1. Examples of the Z80 Digit Shifts

You can double a decimal number (in the Accumulator) by adding it to itself and then performing a decimal correction. i.e.,

```
ADD    A      ;DOUBLE NUMBER
DAA           ;AND MAKE RESULT DECIMAL
```

**DOUBLING
AND HALVING
BINARY
NUMBERS**

Remember that the Accumulator can hold only valid decimal digits in the range 0-99.

You cannot use SLA A (Shift Left Arithmetic A) because that instruction always clears the Half-Carry (only Add and Subtract instructions set H properly).

You can divide a decimal number by two simply by shifting it right logically and then subtracting three from any digit that is eight or larger (since 10 BCD is 16 binary). The following program divides a decimal number in memory location 0040 by two and places the result into memory location 0041.

```
          LD      A.(40H)     :GET DECIMAL NUMBER
          SRL     A           :DIVIDE BY 2 IN BINARY
          BIT     3.A         :IS LEAST SIGNIFICANT DIGIT 8 OR MORE?
          JR      Z.DONE
          SUB     3           :YES. SUBTRACT 3 FOR DECIMAL CORRECTION
DONE:     LD      (41H).A     :STORE NUMBER DIVIDED BY 2
          HALT
```

Try this program and the method on the decimal numbers 28, 30, and 37. Do you understand why it works?

Rounding is simple whether the numbers are binary or decimal. A binary number can be rounded as follows:

BINARY ROUNDING

> If the most significant bit to be dropped is 1.
> add 1 to the remaining bits. Otherwise, leave
> the remaining bits alone.

This rule works because 1 is halfway between 0 and 10 in binary, much as 5 is halfway in decimal (note that 0.5 decimal = 0.1 binary).

So, the following program will round a 16-bit number in memory locations 0040 and 0041 (MSBs in 0041) to an 8-bit number in memory location 0041.

```
          LD      HL.40H
          BIT     7.(HL)      :IS MSB OF EXTRA BYTE 1?
          JR      Z.DONE
          INC     HL          :NO. ROUND UP
          INC     (HL)
DONE:     HALT
```

If the number is longer than 16 bits. the rounding must ripple through the other bytes as needed.

Decimal rounding is a bit more difficult because the crossover point is now BCD 50 and the rounding must produce a decimal result. The rule is:

DECIMAL ROUNDING

> If the most significant digit is to be dropped
> is 5 or more. add 1 to the remaining digits.

The following program will round a 4-digit BCD number in memory locations 0040 and 0041 (MSBs in 0041) to a 2-digit BCD number in memory location 0041.

```
          LD      HL.40H
          LD      A.(HL)      :IS BYTE TO BE DROPPED 50 OR MORE?
          CP      50H
          JR      C.DONE
          INC     HL          :YES. ROUND MSB'S UP
          LD      A.(HL)
          ADD     A.1
          DAA                 :KEEP DIGITS DECIMAL
          LD      (HL).A
DONE:     HALT
```

Remember that the DAA instruction works only on numbers in the Accumulator. In this case. we could round with the instruction INC A. since we know that the Carry is zero (why? — remember the JR instruction). Normally. we need the sequence ADD A.1 followed by DAA. since INC A does not affect the Carry.

Very often when performing multibyte twos complement signed arithmetic, it is necessary to propagate the sign bit through the high-order bytes. This operation can be performed

in a straightforward manner if, as is usually the case, the sign is in the Carry. The SBC A,A instruction has the effect of propagating the state of the Carry throughout a word. Since A-A always equals 0, SBC A,A is equivalent to subtracting the Carry from 0 and can yield only the values 0 and FFH.

PROBLEMS

1) Multiple-Precision Subtraction

Purpose: Subtract one multiple-word number from another. The length of the numbers is in memory location 0040, the numbers themselves start (least significant bits first) in memory locations 0041 and 0051, respectively, and the difference replaces the number starting in memory location 0041. Subtract the number starting in 0051 from the one starting in 0041.

Sample Problem:

```
        (0040)  =  04

        (0041)  =  C3
        (0042)  =  A7
        (0043)  =  5B
        (0044)  =  2F

        (0051)  =  B8
        (0052)  =  35
        (0053)  =  DF
        (0054)  =  14
Result: (0041)  =  0B
        (0042)  =  72
        (0043)  =  7C
        (0044)  =  1A
```

that is,

```
   2F5BA7C3
 - 14DF35B8
   1A7C720B
```

2) Decimal Subtraction

Purpose: Subtract one multiple-word decimal (BCD) number from another. The length of the numbers is in memory location 0040, the numbers themselves start (least significant bits first) in memory locations 0041 and 0051, respectively, and the difference replaces the number starting in memory location 0041. Subtract the number starting in 0051 from the one starting in 0041.

Sample Problem:

```
        (0040)  =  04

        (0041)  =  85
        (0042)  =  19
        (0043)  =  70
        (0044)  =  36

        (0051)  =  59
        (0052)  =  34
        (0053)  =  66
        (0054)  =  12
```

```
Result:   (0041)  =  26
          (0042)  =  85
          (0043)  =  03
          (0044)  =  24
```

that is.
```
              36701985
              12663459
              _____
              24038526
```

3)   8-Bit by 16-Bit Binary Multiplication

Purpose: Multiply the 16-bit unsigned number in memory locations 0040 and 0041 (most significant bits in 0041) by the 8-bit unsigned number in memory location 0042. Store the result in memory locations 0043 through 0045, with the most significant bits in memory location 0045.

Sample Problems:

a.
```
                (0040)  =  03
                (0041)  =  00
                (0042)  =  05

      Result:   (0043)  =  0F
                (0044)  =  00
                (0045)  =  00
```

that is, $3 \times 5 = 15$

b.
```
                (0040)  =  6F
                (0041)  =  72 (29,295 decimal)
                (0042)  =  61 (97 decimal)

      Result:   (0043)  =  0F
                (0044)  =  5C
                (0045)  =  2B
```

that is. $29,295 \times 97 = 2,841,615$

4)   Signed Binary Division

Purpose: Divide the 16-bit signed number in memory locations 0040 and 0041 (most significant bits in 0041 by the 8-bit signed number in memory location 0042. The numbers are normalized so that the magnitude of memory location 0042 is greater than the magnitude of memory location 0041. Store the quotient (signed) in memory location 0043 and the remainder (always positive) in memory location 0044.

Sample Problems:

a.
```
                (0040)  =  C0
                (0041)  =  FF (-64)
                (0042)  =  08

      Result:   (0043)  =  F8 (-8) quotient
                (0044)  =  00 (0) remainder
```

b.
```
                (0040)  =  93
                (0041)  =  ED (-4717)
                (0042)  =  47 (71 decimal)

      Result:   (0043)  =  BD (-67 decimal)
                (0044)  =  28 (+40 decimal)
```

Hint: Determine the sign of the result, perform an unsigned division, and adjust the quotient and remainder properly.

5) Self-Checking Numbers Aligned 1, 3, 7 Mod 10

Purpose: Calculate a checksum digit from a string of BCD digits. The length of the string of digits (number of words) is in memory location 0041. the string of digits (2 BCD digits to a word) starts in memory location 0042. Calculate the checksum digit by the Aligned 1. 3, 7 Mod 10 method and store it in memory location 0040.

The Aligned 1. 3, 7 Mod 10 technique works as follows:

1) Clear the checksum to start.
2) Add the leading digit to the checksum.
3) Multiply the next digit by 3 and add the result to the checksum.
4) Multiply the next digit by 7 and add the result to the checksum.
5) Continue the process (Steps 2-4) until you have used all the digits.
6) The self-checking digit is the least significant digit of the checksum.

For example. if the string of digits is:

$$549321$$

the result will be:

$$\text{Checksum} = 5 + 3 \times 4 + 7 \times 9 + 3 + 3 \times 2 + 7 \times 1 = 96$$
$$\text{Self-checking digit} = 6$$

Sample Problems:

a.
$$
\begin{aligned}
(0041) &= 03 \\
(0042) &= 36 \\
(0043) &= 68 \\
(0044) &= 51
\end{aligned}
$$

Result: Checksum $= 3 + 3 \times 6 + 7 \times 6 + 8 + 3 \times 5 + 7 \times 1 = 93$
$(0040) = 03$

b.
$$
\begin{aligned}
(0041) &= 04 \\
(0042) &= 50 \\
(0043) &= 29 \\
(0044) &= 16 \\
(0045) &= 83
\end{aligned}
$$

Result: Checksum $= 5 + 3 \times 0 + 7 \times 2 + 9 + 3 \times 1 + 7 \times 6 + 8$
$+ 3 \times 3 = 90$
$(0040) = 00$

Hint: Note that $7 = 2 \times 3 + 1$ and $3 = 2 \times 1 + 1$, so the formula $M_i = 2 \times M_{i-1} + 1$ can be used to calculate the next multiplying factor.

REFERENCES

1. Several multiplication algorithms are described in T. Dollhoff, "Microprocessor Software: How to Optimize Timing and Memory Usage. Part Four. Techniques for the Zilog Z80," Digital Design, February 1977, pp. 44-51.

2. Some microprocessors (such as the 9900, 8086, and Z-8000) have hardware multiplication instructions that are somewhat faster, but maximum speed requires the addition of external hardware.

   Other methods for implementing multiplication, division, and other arithmetic tasks are discussed in:

   Geist, D. J., "MOS Processor Picks up Speed with Bipolar Multipliers," Electronics, July 7, 1977, pp. 113-115.

   Kolodzinski, A. and D. Wainland, "Multiplying with a Microcomputer," Electronic Design, January 18, 1978, pp. 78-83.

   Mick, J. R. and J. Springer, "Single-chip Multiplier Expands Digital Role in Signal Processing," Electronics, May 13, 1976, pp. 103-108.

   Parasuraman, B., "Hardware Multiplication Techniques for Microprocessor Systems," Computer Design, April 1977, pp. 75-82.

   Tao, T. F. et al., "Applications of Microprocessors in Control Problems," 1977 Joint Automatic Control Conference Proceedings, San Francisco, CA., June 22-24, 1977.

   Waser, S., "State-of-the-art in High-Speed Arithmetic Integrated Circuits," Computer Design, July 1978, pp. 67-75.

   Weissberger, A. J. and T. Toal, "Tough Mathematical Tasks Are Child's Play for Number Cruncher," Electronics, February 17, 1977, pp. 102-107.

3. See J. R. Herr, "Self-Checking Number Systems," Computer Design, June 1974, pp. 85-91.

Chapter 9
TABLES AND LISTS

Tables and lists are two of the basic data structures used with all computers. We have already seen tables used to perform code conversions and arithmetic. Tables may also be used to identify or respond to commands and instructions, linearize data, provide access to files or records, define the meaning of keys or switches, and choose among alternate programs. Lists are usually less structured than tables. Lists may record tasks that the processor must perform, messages or data that the processor must record, or conditions that have changed or should be monitored. Tables are a simple way of making decisions or solving problems, since no computations or logical functions are necessary. The task, then, reduces to organizing the table so that the proper entry is easy to find. Lists allow the execution of sequences of tasks, the preparation of sets of results, and the construction of interrelated data files (or data bases). Problems include how to add elements to a list and remove elements from it.

EXAMPLES

Add Entry to List

Purpose: Add the contents of memory location 0040 to a list if it is not already present in the list. The length of the list is in memory location 0041 and the list itself begins in memory location 0042.

Sample Problems:

a.

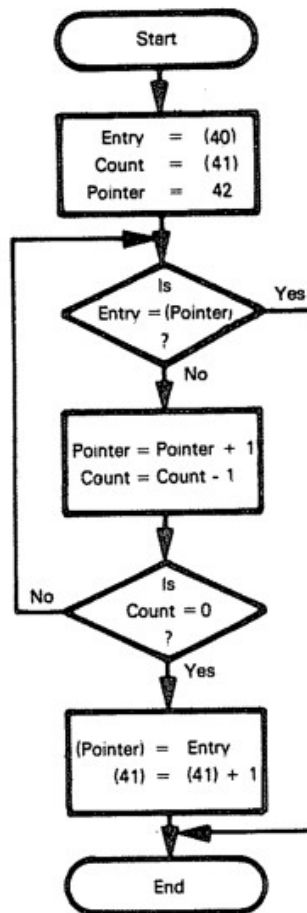|	
(0040)	= 6B
(0041)	= 04
(0042)	= 37
(0043)	= 61
(0044)	= 38
(0045)	= 1D

Result:

(0041)	= 05
(0046)	= 6B

The entry is added to the list, since it is not already present. The length of the list is increased by 1.

b.

(0040)	= 6B
(0041)	= 04
(0042)	= 37
(0043)	= 6B
(0044)	= 38
(0045)	= 1D

Result: No change, since the entry is already in the list.

Flowchart:

Source Program:

```
        LD     HL,40H      :POINT TO ENTRY
        LD     A,(HL)      :GET ENTRY
        INC    HL          :POINT TO COUNT
        LD     B,(HL)      :COUNT = LENGTH OF LIST
        INC    HL          :POINT TO START OF LIST
SRLST:  CP     (HL)        :IS ENTRY = ELEMENT IN LIST?
        JR     Z,DONE      :YES, THROUGH
        INC    HL          :NO, GO ON TO NEXT ELEMENT
        DJNZ   SRLST
        LD     (HL),A      :ADD ENTRY TO LIST
        LD     HL,41H      :ADD 1 TO LIST LENGTH
        INC    (HL)
DONE:   HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	21		LD	HL,40H
0001	40			
0002	00			
0003	7E		LD	A,(HL)
0004	23		INC	HL
0005	46		LD	B,(HL)
0006	23		INC	HL
0007	BE	SRLST:	CP	(HL)
0008	28		JR	Z,DONE
0009	08			
000A	23		INC	HL
000B	10		DJNZ	SRLST
000C	FA			
000D	77	ADELM:	LD	(HL),A
000E	21		LD	HL,41H
000F	41			
0010	00			
0011	34		INC	(HL)
0012	76	DONE:	HALT	

We could also use the block search instruction CPIR in our example, as follows:

Source Program:

```
        LD      HL,40H      ;POINT TO ENTRY
        LD      A,(HL)      ;GET ENTRY
        INC     HL          ;POINT TO COUNT
        LD      B,0         ;COUNT = LENGTH OF LIST (16 BITS)
        LD      C,(HL)
        INC     HL          ;POINT TO START OF LIST
        CPIR                ;LOOK FOR ENTRY IN LIST
        JR      Z,DONE      ;DONE IF ENTRY FOUND
        LD      (HL),A      ;OTHERWISE, ADD ENTRY TO LIST
        LD      HL,41H      ;ADD 1 TO LIST LENGTH
DONE:   HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	23	INC	HL
0005	06	LD	B,0
0006	00		
0007	4E	LD	C,(HL)
0008	23	INC	HL
0009	ED	CPIR	
000A	B1		
000B	28	JR	Z,DONE
000C	05		
000D	77	LD	(HL),A
000E	21	LD	HL,41H
000F	41		
0010	00		
0011	34	INC	(HL)
0012	76	DONE: HALT	

Remember that CPIR automatically repeats the basic Search instruction until either BC is decremented to zero or a true comparison occurs (i.e., A = (HL)).

Be careful of the following slight differences from the previous version:

1) BC is a 16-bit counter. Thus, CPIR can handle strings longer than 256 bytes.

2) The Parity/Overflow bit (P/O) is cleared if BC is decremented to zero, and set otherwise.

Clearly, this method of adding elements is very inefficient if the list

HASHING

is long. We could improve the procedure by limiting the search to part of the list or by ordering the list. We could limit the search by using the entry to get a starting point in the list. This method is called "hashing", and is much like selecting a starting page in a dictionary or directory on the basis of the first letter in an entry. We could order the list by numerical value. The search could then end when the list values went beyond the entry (larger or smaller, depending on the ordering technique used). A new entry would have to be inserted properly, and all the other entries would have to be moved down in the list.

The program could be restructured to use two tables. One table could provide a starting point in the other table; for example, the search point could be based on the most or least significant 4-bit digit in the entry.

The program does not work if the length of the list could be zero (what happens?). We could avoid this problem by checking the length initially. The initialization procedure for the first program would then be:

```
        LD      HL,40H      :POINT TO ENTRY
        LD      A,(HL)      :GET ENTRY
        INC     HL          :POINT TO LENGTH
        LD      B,(HL)      :COUNT = LENGTH OF LIST
        INC     HL          :POINT TO START OF LIST
        INC     B           :IS COUNT ZERO?
        DEC     B
        JR      Z,ADELM     ;YES. GO ADD ENTRY TO LIST
        .
        .
        .
ADELM:  LD      (HL),A      ;ADD ENTRY TO LIST
```

Note that the sequence INC, DEC is an easy way to check for a zero value in a register without using the Accumulator or changing the value in the register.

The procedure:

```
        LD      HL,ADDR
        INC     (HL)
```

is a quick way to add 1 to a counter in memory location ADDR without using the Accumulator. You can use DEC (HL) in a similar manner to subtract 1 from the counter. LD (HL),CONST can place a starting value (such as zero) in the counter. Memory locations should, of course, be used for counters only when no readily accessible registers are available.

If each entry were longer than one word, a pattern-matching program would be necessary. The program would have to proceed to the next entry if a match failed; that is, skip over the last part of the current entry once a mis-match was found.
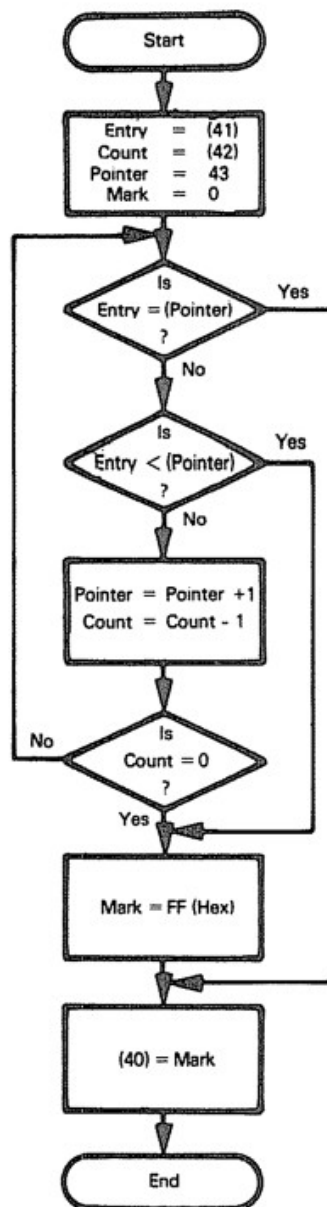
Check an Ordered List

Purpose: Check the contents of memory location 0041 to see if it is in an ordered list. The length of the list is in memory location 0042; the list itself begins in memory location 0043 and consists of unsigned binary numbers in increasing order. If the contents of location 0041 is in the list, clear memory location 0040; otherwise, set memory location 0040 to FF (hex).

Sample Problems:

a.
$$(0041) = 6B$$
$$(0042) = 04$$
$$(0043) = 37$$
$$(0044) = 55$$
$$(0045) = 7D$$
$$(0046) = A1$$

Result: $(0040) = FF$, since 6B is not in the list.

b.
$$(0041) = 6B$$
$$(0042) = 04$$
$$(0043) = 37$$
$$(0044) = 55$$
$$(0045) = 6B$$
$$(0046) = A1$$

Result: $(0040) = 00$, since 6B is in the list.

Flowchart:

```
                    ┌──────────┐
                    │   Start  │
                    └────┬─────┘
                         ▼
              ┌─────────────────────┐
              │  Entry   =   (41)   │
              │  Count   =   (42)   │
              │  Pointer =    43    │
              │  Mark    =     0    │
              └──────────┬──────────┘
                         ▼
        ┌──────►    ◇ Is           ◇ ── Yes ──┐
        │          Entry = (Pointer)          │
        │               ?                     │
        │               │ No                  │
        │               ▼                     │
        │          ◇ Is           ◇ ── Yes ──┤
        │          Entry < (Pointer)          │
        │               ?                     │
        │               │ No                  │
        │               ▼                     │
        │     ┌─────────────────────┐         │
        │     │ Pointer = Pointer +1│         │
        │     │ Count = Count - 1   │         │
        │     └──────────┬──────────┘         │
        │                ▼                    │
        │           ◇ Is     ◇                │
        └── No ──   Count = 0                 │
                        ?                     │
                        │ Yes                 │
                        ▼                     │
              ┌─────────────────────┐         │
              │   Mark = FF (Hex)   │         │
              └──────────┬──────────┘         │
                         ▼◄───────────────────┘
              ┌─────────────────────┐
              │    (40) = Mark      │
              └──────────┬──────────┘
                         ▼
                    ┌──────────┐
                    │   End    │
                    └──────────┘
```

The searching process is a bit different here since the elements are ordered. Once we find an element larger than the entry, the search is over, since subsequent elements will be even larger. You may want to try an example to convince yourself that the procedure works.

As in the previous problem, a table or other method that could choose a good starting point would speed up the search. One method would be to start in the middle and determine which half of the list the entry was in, then divide the half into halves, etc. This method is called a binary search, since it divides the remaining part of the list in half each time.[1]

SEARCHING METHODS

Source Program:

```
        LD      HL,41H      ;POINT TO ENTRY
        LD      A,(HL)      ;GET ENTRY
        INC     HL          ;POINT TO LENGTH
        LD      B,(HL)      ;COUNT = LENGTH OF LIST
        LD      C,0         ;MARK = ZERO FOR IN LIST
        INC     HL          ;POINT TO START OF LIST
SRLST:  CP      (HL)        ;IS ENTRY = ELEMENT IN LIST?
        JR      Z,DONE      ;YES  SEARCH COMPLETED
        JR      C,NOTIN     ;ENTRY NOT IN LIST IF LESS THAN ELEMENT
        INC     HL
        DJNZ    SRLST
NOTIN:  LD      C,OFFH      ;MARK = FF FOR NOT IN LIST
DONE:   LD      A,C         ;SAVE MARK
        LD      (40H),A
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	7E		LD	A,(HL)
0004	23		INC	HL
0005	46		LD	B,(HL)
0006	0E		LD	C,0
0007	00			
0008	23		INC	HL
0009	BE	SRLST:	CP	(HL)
000A	28		JR	Z,DONE
000B	07			
000C	38		JR	C,NOTIN
000D	03			
000E	23		INC	HL
000F	10		DJNZ	SRLST
0010	F8			
0011	0E	NOTIN:	LD	C,OFFH
0012	FF			
0013	79		LD	A,C
0014	32		LD	(40H),A
0015	40			
0016	00			
0017	76		HALT	

The Z80 block search instructions are not as useful here as in the previous example because we want to do more than a simple search. Now we also want to check to see if we have examined the relevant part of the list (i.e., the part where the elements are less than or equal to the entry). Try rewriting the program to use CPI. Remember that you must use the Parity/Overflow flag to determine if the byte counter has been decremented to zero.

Remove Element from Queue

Purpose: Memory locations 0042 and 0043 contain the address of the head of the queue (MSBs in 0043). Place the address of the first element (head) of a queue into memory locations 0040 and 0041 (MSBs in 0041) and update the queue to remove the element. Each element in the queue is two bytes long and contains the address of the next two-byte element in the queue. The last element in the queue contains zero to indicate that there is no next element.

Queues are used to store data in the order in which it will be used, or tasks in the order in which they will be executed. The queue is a first-in, first-out data structure; i.e., elements are removed from the queue in the same order in which they were entered. Operating systems place tasks in queues so that they will be executed in the proper order. I/O drivers transfer data to or from queues so that it will be transmitted or handled in the proper order. Buffers may be queued so that the next available one can easily be found and those that are released can easily be added to the available storage. Queues may also be used to link requests for storage, timing, or I/O so that they can be satisfied in the correct order.

In real applications each element in the queue will typically contain a large amount of information or storage space besides the address required to link the element to the next one.

Sample Problems:

a.
$$\left.\begin{array}{ll}(0042) & = 46\\(0043) & = 00\end{array}\right\} \text{ address of first element in queue}$$

$$\left.\begin{array}{ll}(0046) & = 4D\\(0047) & = 00\end{array}\right\} \text{ address of second element in queue}$$

$$\left.\begin{array}{ll}(004D) & = 00\\(004E) & = 00\end{array}\right\} \text{ end of queue}$$

Result:
$$\left.\begin{array}{ll}(0040) & = 46\\(0041) & = 00\end{array}\right\} \text{ address of element removed from queue}$$

$$\left.\begin{array}{ll}(0042) & = 4D\\(0043) & = 00\end{array}\right\} \text{ address of new first element in queue}$$

b.
$$\left.\begin{array}{ll}(0042) & = 00\\(0043) & = 00\end{array}\right\} \text{ empty queue}$$

Result:
$$\left.\begin{array}{ll}(0040) & = 00\\(0041) & = 00\end{array}\right\} \text{ no element available from queue}$$

Flowchart:

```
                    ┌──────────┐
                    │  Start   │
                    └────┬─────┘
                         │
                         ▼
              ┌────────────────────────┐
              │ Pointer  = (42 and 43) │
              │ (40 and 41) = Pointer  │
              └───────────┬────────────┘
                          │
                          ▼
                       ╱──────╲
          Yes         ╱   Is   ╲
      ┌──────────────╱  Pointer 0 ╲
      │              ╲     ?     ╱
      │               ╲─────────╱
      │                    │ No
      │                    ▼
      │          ┌────────────────────┐
      │          │ (42) = (Pointer)   │
      │          │ (43) = (Pointer + 1) │
      │          └─────────┬──────────┘
      │                    │
      │                    ▼
      │              ┌──────────┐
      └─────────────▶│   End    │
                     └──────────┘
```
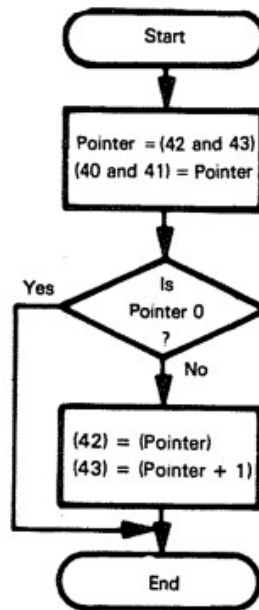
Source Program:

```
        LD      HL,(42H)      :GET ADDRESS OF HEAD OF QUEUE
        LD      (40H),HL      :REMOVE HEAD OF QUEUE
        LD      A,H           :IS QUEUE EMPTY?
        OR      L
        JR      Z,DONE        :YES, DONE
        LD      E,(HL)        :NO, GET ADDRESS OF NEXT ELEMENT
        INC     HL
        LD      D,(HL)
        LD      (42H),DE      :MOVE NEXT ELEMENT TO HEAD OF QUEUE
DONE:   HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	2A	LD	HL,(42H)
0001	42		
0002	00		
0003	22	LD	(40H),HL
0004	40		
0005	00		
0006	7C	LD	A,H
0007	B5	OR	L
0008	28	JR	Z,DONE
0009	07		
000A	5E	LD	E,(HL)
000B	23	INC	HL
000C	56	LD	D,(HL)
000D	ED	LD	(42H),DE
000E	53		
000F	42		
0010	00		
0011	76	DONE: HALT	

Queuing can handle lists that are not in sequential memory locations. Each element must contain the address of the next element. Such lists allow you to handle data or tasks in the proper order, change variables, or fill in definitions in a program. Extra storage is required, but elements can easily be added to the queue or deleted from it.

Note the use of the sequence:

```
LD      A,H
OR      L
```

to determine if the contents of a 16-bit register pair is zero. Remember that INC and DEC do not affect any flags when applied to a register pair. Try to devise some other sequences that could handle this problem — it obviously occurs whenever you use a 16-bit counter rather than the 8-bit counter that we have used in most of the examples.

One problem is that there is no instruction that loads a register pair using the address in a register pair. A sequence of instructions is necessary whenever a register pair must be loaded directly.

It may be useful to maintain pointers to both ends of the queue rather than just to its head. The data structure may then be used in either a first-in, first-out manner or in a last-in, first-out manner, depending on whether new elements are added to the head or the tail. How would you change the program example so that memory locations 0044 and 0045 contain the address of the last element (tail) of the queue?

If there are no elements in the queue, the program clears memory locations 0040 and 0041. A program that requested an element from the queue would then have to check those memory locations to see if its request had been satisfied. Can you suggest other ways to provide this information?

8-Bit Sort

Purpose: Sort an array of unsigned binary numbers into descending order. The length of the array is in memory location 0040 and the array itself begins in memory location 0041.

Sample Problem:

	(0040)	=	06
	(0041)	=	2A
	(0042)	=	B5
	(0043)	=	60
	(0044)	=	3F
	(0045)	=	D1
	(0046)	=	19
Result:	(0041)	=	D1
	(0042)	=	B5
	(0043)	=	60
	(0044)	=	3F
	(0045)	=	2A
	(0046)	=	19

A simple sorting technique works as follows:

SIMPLE SORTING ALGORITHM

Step 1) Clear a flag INTER.

Step 2) Examine each consecutive pair of numbers in the array. If any are out of order, exchange them and set INTER.

Step 3) If INTER = 1 after the entire array has been examined, return to Step 1.

INTER will be set if any consecutive pair of numbers is out of order. Therefore, if IN-TER = 0 at the end of a pass through the entire array, the array is in proper order.

This sorting method is referred to as a "bubble sort". It is an easy algorithm to implement. However, other sorting techniques should be considered when sorting long lists where speed is important.[2]

The technique operates as follows in a simple case. Let us assume that we want to sort an array into descending order; the array has four elements — 12, 03, 15, 08.

1st Iteration:

Step 1) INTER = 0

Step 2) Final order of the array is:
       12
       15
       08
       03
       since the second pair (03.15) is exchanged and so is the third pair (03,08).
       INTER = 1.

2nd Iteration:

Step 1) INTER = 0

Step 2) Final order of the array is:
       15
       12
       08
       03
       since the first pair (12.15) is exchanged. INTER = 1.

3rd Iteration:

Step 1) INTER = 0

Step 2) The elements are already in order, so no exchanges are necessary and INTER remains zero.

Flowchart:

```
                    ┌─────────────┐
                    │    Start    │
                    └──────┬──────┘
                           │
          ┌────────────────┤
          │         ┌──────▼──────────┐
          │         │ Inter   =   0   │
          │         │ Count   = (40)-1│
          │         │ Pointer =   41  │
          │         └──────┬──────────┘
          │                │
          │     ┌──────────┤
          │     │      ╱───▼────╲
          │     │    ╱     Is     ╲
          │  Yes│   ╱  (Pointer) ≥  ╲
          │     │  ╱  (Pointer + 1)  ╲
          │     │  ╲        ?        ╱
          │     │   ╲─────────┬─────╱
          │     │             │ No
          │     │    ┌────────▼──────────┐
          │     │    │ Temp    = (Pointer)│
          │     │    │(Pointer)=(Pointer+1)│
          │     │    │(Pointer+1) = Temp  │
          │     │    │ Inter   = 1        │
          │     │    └────────┬──────────┘
          │     │             │
          │     └─────────────┤
          │          ┌────────▼──────────┐
          │          │ Pointer = Pointer +1│
          │          │ Count = Count - 1  │
          │          └────────┬──────────┘
          │                   │
          │         ╱─────────▼─────╲
          │    No  ╱       Is        ╲
          ├───────╱      Count 0       ╲
          │       ╲         ?          ╱
          │        ╲─────────┬────────╱
          │                  │ Yes
          │        ╱─────────▼─────╲
          │    No  ╱       Is        ╲
          └───────╱      Inter 0      ╲
                  ╲         ?          ╱
                   ╲─────────┬────────╱
                             │ Yes
                    ┌────────▼──────┐
                    │     End       │
                    └───────────────┘
```
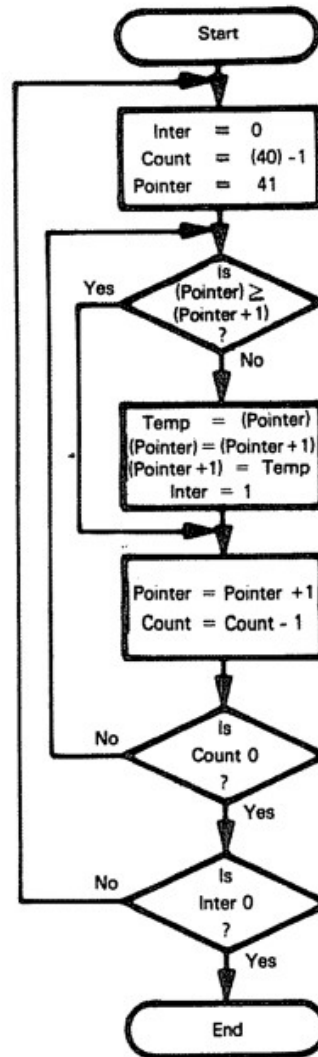
Source Program:

```
SORT:   LD      C,0           ;CLEAR INTERCHANGE FLAG
        LD      HL,40H        :COUNT = LENGTH OF ARRAY
        LD      B,(HL)
        DEC     B             ;NUMBER OF PAIRS = COUNT-1
        INC     HL            ;POINT TO START OF ARRAY
PASS1:  LD      A,(HL)        ;GET ELEMENT FROM ARRAY
        INC     HL
        CP      (HL)          ;IS IT LESS THAN NEXT ELEMENT?
        JR      NC,CNT        ;NO, NO INTERCHANGE NECESSARY
        LD      D,(HL)        ;YES, INTERCHANGE ELEMENTS
        LD      (HL),A
        DEC     HL
        LD      (HL),D
        INC     HL
        LD      C,1           ;SET INTERCHANGE FLAG
CNT:    DJNZ    PASS1
        DEC     C             :WAS INTERCHANGE FLAG SET?
        JR      Z,SORT        ;YES, DO ANOTHER PASS
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	0E	SORT:	LD	C,0
0001	00			
0002	21		LD	HL,40H
0003	40			
0004	00			
0005	46		LD	B,(HL)
0006	05		DEC	B
0007	23		INC	HL
0008	7E	PASS1:	LD	A,(HL)
0009	23		INC	HL
000A	BE		CP	(HL)
000B	30		JR	NC,CNT
000C	07			
000D	56		LD	D,(HL)
000E	77		LD	(HL),A
000F	2B		DEC	HL
0010	72		LD	(HL),D
0011	23		INC	HL
0012	0E		LD	C,1
0013	01			
0014	10	CNT:	DJNZ	PASS1
0015	F2			
0016	0D		DEC	C
0017	28		JR	Z,SORT
0018	E7			
0019	76		HALT	

The case where two elements in the array are equal is very important here. The program should not perform an interchange in that case, since that interchange would occur in every pass. The result would be that every pass would set the interchange flag, thus producing an endless loop.

The program must reduce the counter by 1, since the number of consecutive pairs i' one less than the number of elements (the last element has no successor). Before start ing each sorting pass, we must be careful to reinitialize the counter, pointer, and in terchange flag.

There are many possible minor variations on this program. For example, we could use RES 0,C and SET 0,C to clear and set the interchange flag instead of LD C,0 and LD C,1. We could also use the sequence MOV B,C followed by DJNZ SORT to check the in terchange flag.

Note that Register B should be used for the inner counter, since that counter is decre mented most frequently. This allows us to take maximum advantage of the DJNZ in struction.

Indexing would be a convenient way to perform the interchange if the Z80's index registers were more accessible. Try rewriting the program so as to use one of the index registers and compare the execution time and memory usage of the rewritten program to those of the original program.

Using an Ordered Jump Table

Purpose: Use the contents of memory location 0040 as an index to a jump table start ing in memory location 0041. Each entry in the jump table contains a 16-bit address with LSBs in the first word. The program should transfer control to the address with the appropriate index; that is, if the index is 6, the pro gram jumps to address entry #6 in the table. Assume that the table has fewer than 128 entries.

Sample Problem:

$$
\begin{array}{rcl}
(0040) & = & 02 \\
(0041) & = & 48 \\
(0042) & = & 00 \\
(0043) & = & 4C \\
(0044) & = & 00 \\
(0045) & = & 50 \\
(0046) & = & 00 \\
(0047) & = & 54 \\
(0048) & = & 00 \\
\end{array}
$$

Result: (PC) = 0050, since that is entry #2.
(starting from zero) in the jump table.

Flowchart:
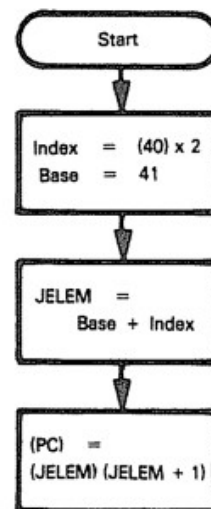
The last box results in a transfer of control to the address obtained from the table.

Source Program:

```
LD      HL.40H      ;POINT TO INDEX
LD      A,(HL)      :GET INDEX
ADD     A.A         :DOUBLE INDEX FOR 2-BYTE TABLE
LD      E.A
LD      D,0         ;EXTEND INDEX TO 16 BITS
INC     HL          :BASE ADDRESS OF JUMP TABLE
ADD     HL.DE       :INDEX INTO JUMP TABLE
LD      E.(HL)      :GET LSB'S OF DESTINATION ADDRESS
INC     HL
LD      D,(HL)      :GET MSB'S OF DESTINATION ADDRESS
EX      DE.HL
JP      (HL)        :TRANSFER CONTROL TO DESTINATION
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	21	LD	HL.40H
0001	40		
0002	00		
0003	7E	LD	A.(HL)
0004	87	ADD	A,A
0005	5F	LD	E,A
0006	16	LD	D,0
0007	00		
0008	23	INC	HL
0009	19	ADD	HL.DE
000A	5E	LD	E,(HL)
000B	23	INC	HL
000C	56	LD	D,(HL)
000D	EB	EX	DE,HL
000E	E9	JP	(HL)

Jump tables are very useful in situations where one of several routines must be selected. Such situations arise in decoding commands. selecting test programs, choosing alternate methods, or selecting an I/O configuration.

The jump table replaces a whole series of conditional jump operations. The program that accesses the jump table could be used to access several different tables merely by changing the starting address.[3]

The data must be multiplied by two to give the correct index. since each entry in the jump table is a two-byte address.

The instruction JP (HL), which transfers the contents of Register Pair HL to the Program Counter. is an indirect jump that is very handy in jump tables and monitor programs. Note that JP (HL) is a

INDIRECT JUMPS

Jump instruction. since it places a new value into the Program Counter; however. it allows us to place a variable address directly into the Program Counter. All of the Conditional Jump instructions (and the Call instructions) use fixed addresses. The only Jump instructions with similar flexibility are the two-word instructions JP (IX) and JP (IY).

No ending operation is necessary. since JP (HL) transfers control to the address obtained from the jump table.

PROBLEMS

1) Remove an Entry From a List

Purpose: Remove the contents of memory location 0040 from a list if it is present. The length of the list is in memory location 0041 and the list itself begins in memory location 0042. Move the entries below the one removed up one position and reduce the length of the list by 1.

Sample Problems:

a.
$$\begin{aligned}
(0040) &= 6B \\
(0041) &= 04 \\
(0042) &= 37 \\
(0043) &= 61 \\
(0044) &= 28 \\
(0045) &= 1D
\end{aligned}$$

Result: No change, since the entry is not in the list.

b.
$$\begin{aligned}
(0040) &= 6B \\
(0041) &= 04 \\
(0042) &= 37 \\
(0043) &= 6B \\
(0044) &= 28 \\
(0045) &= 1D
\end{aligned}$$

Result:
$$\begin{aligned}
(0041) &= 03 \\
(0042) &= 37 \\
(0043) &= 28 \\
(0044) &= 1D
\end{aligned}$$

The entry is removed from the list and the ones below it are moved up one position. The length of the list is reduced by 1.

2) Add an Entry to an Ordered List

Purpose: Place the contents of memory location 0040 into an ordered list if it is not already there. The length of the list is in memory location 0041, and the list itself begins in memory location 0042, which consists of unsigned binary numbers in increasing order. Place the new entry in the correct position in the list, adjust the elements below it down, and increase the length of the list by 1.

Sample Problems:

a.
(0040)	=	6B
(0041)	=	04
(0042)	=	37
(0043)	=	55
(0044)	=	7D
(0045)	=	A1

Result:
(0041)	=	05
(0044)	=	6B
(0045)	=	7D
(0046)	=	A1

b.
(0040)	=	6B
(0041)	=	04
(0042)	=	37
(0043)	=	55
(0044)	=	6B
(0045)	=	A1

Result: No change, since the entry is already in the list.

3) Add an Element to a Queue

Purpose: Add the address in memory locations 0040 and 0041 (MSBs in 0041) to a queue. The address of the first element of the queue is in memory locations 0042 and 0043 (MSBs in 0043). Each element in the queue contains either the address of the next element in the queue or zero if there is no next element; all addresses are 16 bits long with the most significant bits in the second word of the element. The new element goes at the end (tail) of the queue; its address will be in the element that was at the end of the queue and it will contain zero to indicate that it is now the end of the queue.

Sample Problem:

(0040)	=	4D	new element to be added to queue
(0041)	=	00	
(0042)	=	46	pointer to head of queue
(0043)	=	00	
(0046)	=	00	last element in queue
(0047)	=	00	

Result:
(0046)	=	4D	old last element points to
(0047)	=	00	new last element
(004D)	=	00	new last element in queue
(004E)	=	00	

How would you add an element to the queue if memory locations 0044 and 0045 contained the address of the tail (last element) of the queue?

4)   16-Bit Sort

Purpose: Sort an array of unsigned 16-bit binary numbers into descending order. The length of the array is in memory location 0040 and the array itself begins in memory location 0041. Each 16-bit number is stored with the least significant bits in the first word.

Sample Problem:

$$
\begin{aligned}
(0040) &= 03 \\
(0041) &= D1 \\
(0042) &= 19 \\
(0043) &= 60 \\
(0044) &= 3F \\
(0045) &= 2A \\
(0046) &= B5
\end{aligned}
$$

Result:
$$
\begin{aligned}
(0041) &= 2A \\
(0042) &= B5 \\
(0043) &= 60 \\
(0044) &= 3F \\
(0045) &= D1 \\
(0046) &= 19
\end{aligned}
$$

The numbers are B52A, 3F60, and 19D1

5)   Using a Jump Table With a Key

Purpose: Use the contents of memory location 0040 as the key to a jump table starting in memory location 0041. Each entry in the jump table contains an 8-bit key value followed by a 16-bit address (MSBs in second word) to which the program should transfer control if the key is equal to that key value.

Sample Problem:

$$
\begin{aligned}
(0040) &= 38 \\
(0041) &= 32 \\
(0042) &= 4B \\
(0043) &= 00 \\
(0044) &= 35 \\
(0045) &= 4D \\
(0046) &= 00 \\
(0047) &= 38 \\
(0048) &= 4F \\
(0049) &= 00
\end{aligned}
$$

Result: (PC) = 004F, since that address corresponds
to key value 38.

Try writing the program with and without the CPIR instruction. Can you think of a way to simplify the version that uses the CPIR instruction? Hint: place all the corresponding 8-bit words into separate tables so that the program only has to add 1 to the table pointer to move from one key value to the next.[4]

REFERENCES

1. Knuth describes other searching techniques in his book <u>The Art of Computer Programming</u>, <u>Volume III: Sorting and Searching</u>, Addison-Wesley, Reading, Mass., 1978. Knuth also has discussed searching and hashing in a more elementary way in an article entitled "Algorithms" (see the April 1977 issue of <u>Scientific American</u>).

2. There are many sorting algorithms that vary widely in efficiency. Knuth describes some in the book mentioned above (<u>The Art of Computer Programming, Volume III: Sorting and Searching</u>). Sorting and searching algorithms are also discussed in K. A. Schember and J. R. Rumsey, "Minimal Storage Sorting and Searching Techniques for RAM Applications, a Tutorial". <u>Computer</u>, June 1977, pp. 92-100.

3. There are additional examples of the use of jump tables in L. A. Leventhal, "Cut Your Processor's Computation Time". <u>Electronic Design</u>, August 16, 1977, pp. 82-89, and in Chapter 7 of J. B. Peatman, <u>Microcomputer-Based Design</u>, McGraw-Hill, New York, 1977.

4. This method is discussed by T. Dollhoff in "Microprocessor Software: How to Optimize Timing and Memory Usage: Part Four: Techniques for the Zilog Z80". <u>Digital Design</u>, February 1977, pp. 48-49.

Chapter 10
SUBROUTINES

None of the examples that we have shown so far is typically a program all by itself. Most real programs perform a series of tasks, many of which may be the same or may be common to several different programs. We need a way to formulate these tasks once and make the formulations conveniently available both in different parts of the current program and in other programs.

The standard method is to write subroutines that perform particular tasks. The resulting sequences of instructions can be written once, tested once, and then used repeatedly. They can form a subroutine library that provides documented solutions to common problems.

SUBROUTINE LIBRARY

Most microprocessors have special instructions for transferring control to subroutines and restoring control to the main program. We often refer to the special instruction that transfers control to a subroutine as Call, Jump-to-Subroutine, Jump and Mark Place, or Jump and Link. The special instruction that restores control to the main program is usually called Return. On the Z80 microprocessor, the Call instruction (CALL) saves the old value of the Program Counter in the RAM Stack before placing the starting address of the subroutine into the Program Counter; the Return instruction (RET) gets the old value from the Stack and puts it back in the Program Counter. The effect is to transfer program control, first to the subroutine and then back to the main program. Clearly the subroutine may itself transfer control to a subroutine, and so on.

SUBROUTINE INSTRUCTIONS

In order to be really useful, a subroutine must be general. A routine that can perform only a specialized task, such as looking for a particular letter in an input string of fixed length, will not be very useful. If, on the other hand, the subroutine can look for any letter in strings of any length, it will be far more helpful. We call the data or addresses that the subroutine allows to vary "parameters". An important part of writing subroutines is deciding which variables should be parameters.

One problem is transferring the parameters to the subroutine: this process is called passing parameters. The simplest method is for the main program to place the parameters into registers. Then the subroutine can simply assume that the parameters are there. Of course, this technique is limited by the number of registers that are available. The parameters may, however, be addresses as well as data. For example, a sorting routine could begin with the starting address of an array in Register Pair HL.

PASSING PARAMETERS

Other methods are necessary when there are more parameters. One possibility is to use the Stack. The main program can place the parameters into the Stack and the subroutine can retrieve them. The advantages of this method are that the Stack is essentially unlimited in size, and that data in the Stack is not lost even if the Stack is used again.

The disadvantages are that few Z80 instructions use the Stack, and the Call instruction also stores the return address in the Stack. Another method is to use an area of memory for parameters. The main program can place the address of the area into Register Pair HL or into one of the index registers and the subroutine can retrieve the data as needed. However, this procedure is awkward if the parameters themselves are addresses.

Sometimes a subroutine must have special characteristics. A ┃**RELOCATION**┃ subroutine is relocatable if it can be placed anywhere in memory. You can use such a subroutine easily, regardless of the placement of other programs or the arrangement of the memory. A strictly relocatable program can use no absolute addresses; all addresses must be relative to the start of the program. A relocating loader is necessary to place the program in memory properly; the loader will start the program after other programs and will add the starting address or relocation constant to all addresses in the program.

A subroutine is reentrant if it can be interrupted and called by the ┃**REENTRANT**┃ interrupting program and still give the correct results for both the ┃**SUBROUTINE**┃ interrupting and interrupted programs. Reentrancy is important for standard subroutines in an interrupt-based system. Otherwise the interrupt service routines cannot use the standard subroutines without causing errors. Microprocessor subroutines are easy to make reentrant, since the Call instruction uses the Stack and that procedure is automatically reentrant. The only remaining requirement is that the subroutine use the registers and Stack rather than fixed memory locations for temporary storage. This is a bit awkward, but usually can be done if necessary.

A subroutine is recursive if it calls itself. Such a subroutine clearly must also be reentrant. However, recursive subroutines are uncommon in microprocessor applications.

Most programs consist of a main program and several subroutines. This is advantageous because you can use proven routines and debug and test the other subroutines separately. You must, however, be careful to use the subroutines properly and remember their exact effects on registers and memory locations.

SUBROUTINE DOCUMENTATION

Subroutine listings must provide enough information so that ┃**DOCUMENTING**┃ users need not examine the subroutine's internal structure. ┃**SUBROUTINES**┃ Among the necessary specifications are:

- A description of the purpose of the subroutine.
- A list of input and output parameters.
- Registers and memory locations used.
- A sample case.

If these guidelines are followed, the subroutine will be easy to use.

EXAMPLES

It is important to note that the following examples all reserve an area of memory for the RAM Stack. If the monitor in your microcomputer establishes such an area, you may use it instead. If you wish to try establishing your own Stack area, remember to save and restore the monitor's Stack Pointer in order to produce a proper return at the end of your main program.

To save the monitor Stack Pointer, use the instruction LD (addr),SP. To restore the monitor Stack Pointer, use the instruction LD SP,(addr). Both of these instructions require a two-byte operation code (ED 7B for loading the Stack Pointer, ED 73 for storing it) in addition to the two bytes of address.

We have used address 0080 (hex) as the starting point for the Stack. You may have to consistently replace that address with one more suitable for your configuration. You should consult your microcomputer's manual to determine the required changes.

Hex to ASCII

Purpose: Convert the contents of the Accumulator to an ASCII character. Place the result in the Accumulator. Assume that the Accumulator contains a single hexadecimal digit.

Sample Problems:

a.   (A) = 0C

    Result: (A) = 43 'C'

b.   (A) = 06

    Result: (A) = 36 '6'

Flowchart:
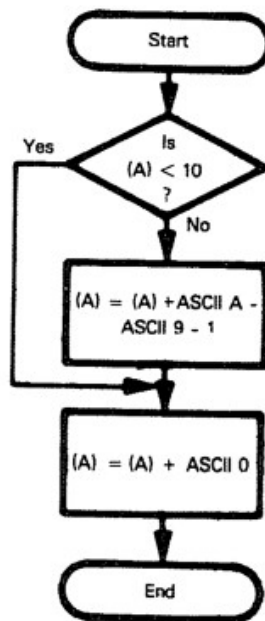
Source Program:

The calling program starts the Stack at memory location 0080, gets the data from memory location 0040, calls the conversion subroutine, and stores the result in memory location 0041.

```
        ORG     0
        LD      SP,80H      ;START STACK AT LOCATION 0080
        LD      A,(40H)     ;GET DATA
        CALL    ASDEC       ;CONVERT DATA TO ASCII
        LD      (41H),A     ;STORE RESULT
        HALT
```

The subroutine converts a hexadecimal digit to ASCII.

```
        ORG     20H
ASDEC:  CP      10          ;IS DATA A DECIMAL DIGIT?
        JR      C,ASCZ
        ADD     A,'A'-'9'-1  ;NO, ADD OFFSET FOR LETTERS
ASCZ:   ADD     A,'0'       ;CONVERT DATA TO ASCII
        RET
```

Subroutine Documentation:

```
;
; SUBROUTINE ASDEC
;
; PURPOSE: ASDEC CONVERTS A HEXADECIMAL
;     DIGIT IN THE ACCUMULATOR TO AN
;     ASCII DIGIT IN THE ACCUMULATOR
;
; INITIAL CONDITIONS: HEX DIGIT IN A
;
; FINAL CONDITIONS: ASCII CHARACTER IN A
;
; REGISTERS USED: A
;
; SAMPLE CASE
;     INITIAL CONDITIONS: 6 IN ACCUMULATOR
;     FINAL CONDITIONS: ASCII 6 (HEX 36)
;         IN ACCUMULATOR
;
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
1) Calling program				
0000	31		LD	SP,80H
0001	80			
0002	00			
0003	3A		LD	A,(40H)
0004	40			
0005	00			
0006	CD		CALL	ASDEC
0007	20			
0008	00			
0009	32		LD	(41H),A
000A	41			
000B	00			
000C	76		HALT	
2) Subroutine				
0020	FE	ASDEC:	CP	10
0021	0A			
0022	38		JR	C,ASCZ
0023	02			
0024	C6		ADD	A,'A'-'9'-1
0025	07			
0026	C6	ASCZ:	ADD	A,'0'
0027	30			
0028	C9		RET	

The instruction LD SP,80H starts the Stack at memory location 0080. Remember that the Stack grows downward (to lower addresses). We usually place the Stack at the high

10-4

end of RAM (i.e.. the highest address) so that it will not interfere with other temporary storage.

The Call instruction places the subroutine starting address (0020 hex) into the Program Counter and saves the old Program Counter (0009 hex) in the Stack. The procedure is:

STEP 1 — Decrement Stack Pointer. save MSBs of old Program Counter in Stack.

STEP 2 — Decrement Stack Pointer. save LSBs of old Program Counter in Stack.

Note that the Z80 Stack Pointer always contains the address of the last occupied Stack location.

The result in this case is:

$$(007F) = 00$$
$$(007E) = 09$$
$$(SP) = 007E$$

The value that is saved is the value of the Program Counter after the processor has fetched the entire Call instruction from memory. Note that the address ends up stored just like other Z80 addresses. with the least significant bits in the lower address.

The Return instruction loads the Program Counter with the contents of the bottom two memory locations in the Stack. The procedure is:

STEP 1 — Load eight bits from Stack into LSBs of Program Counter. Increment Stack Pointer.

STEP 2 — Load eight bits from Stack into MSBs of Program Counter. Increment Stack Pointer.

The result in this case is:

$$(PC) = (007F) \text{ and } (007E)$$
$$= 0009$$
$$(SP) = 0080$$

This subroutine has a single input parameter and produces a single result. The Accumulator is the obvious place to put both.

The calling program involves three steps: placing the data into the Accumulator. calling the subroutine. and storing the result. The overall initialization must also place the Stack in the appropriate area of memory.

The subroutine is reentrant. since it uses no data memory; it is relocatable. since the address ASCZ is relative.

Note that the CALL instruction results in the execution of four or five instructions taking 36 or 38 clock cycles. A subroutine call can take a long time even though it appears to be a single instruction in the program.

If you plan to use the Stack for parameters. remember that CALL places the return address at the top of the Stack. You can increment the Stack Pointer twice (INC SP) to get past the return address. but you must also remember to adjust the Stack Pointer properly before returning. You can also move the Stack Pointer to Registers H and L with the sequence:

```
LD      HL.0
ADD     HL.SP         ;MOVE STACK POINTER TO ADDRESS REGISTER
```

Now you can use implied memory addressing with H and L to access data in the Stack. Another alternative is to move the Stack Pointer to an index register (say IX) with the sequence:

```
LD      IX.0
ADD     IX.SP         ;MOVE STACK POINTER TO INDEX REGISTER
```

This alternative has the advantage that you can now access data and addresses in the Stack with indexed offsets. Furthermore, Register Pair HL is immediately available for use in the subroutine. Note that you can use the instructions LD SP,HL or LD SP,IX to return an adjusted value to the Stack Pointer.

Length of a String of Characters

Purpose: Determine the length of a string of ASCII characters. The starting address of the string is in Register Pair HL. The end of the string is marked by a carriage return character (CR, hex 0D). Place the length of the string (excluding the carriage return) into the Accumulator.

Sample Problems:

a.
$$(HL) = 0043$$
$$(0043) = 0D$$

Result: $(A) = 00$

b.

(HL)	=	0043	
(0043)	=	52	'R'
(0044)	=	41	'A'
(0045)	=	54	'T'
(0046)	=	48	'H'
(0047)	=	45	'E'
(0048)	=	52	'R'
(0049)	=	0D	CR

Result: $(A) = 06$

Flowchart:
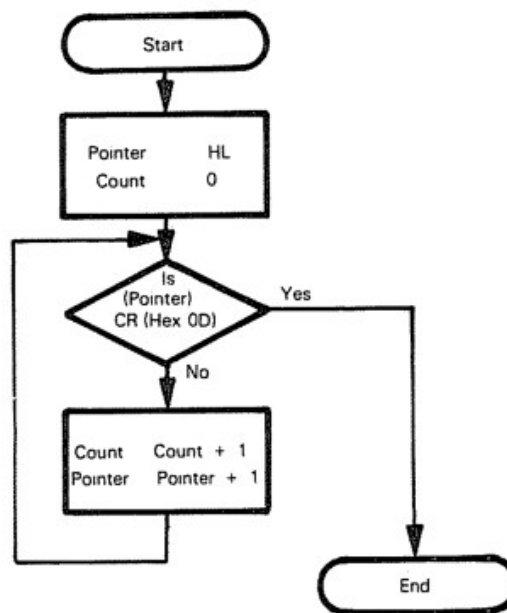
Source Program:

The calling program starts the Stack at memory location 0080, gets the starting address of the string from memory locations 0040 and 0041. calls the string length subroutine, and stores the result in memory location 0042.

```
LD      SP.80H      :START STACK AT LOCATION 0080
LD      HL.(40H)    :GET STARTING ADDRESS OF STRING
CALL    STLEN       :DETERMINE STRING LENGTH
LD      (42H).A     :STORE STRING LENGTH
HALT
```

The subroutine determines the length of a string of ASCII characters and places the length into the Accumulator.

```
        ORG     20H
STLEN:  LD      B.0         :STRING LENGTH = ZERO
        LD      A.0DH       :GET ASCII CARRIAGE RETURN
CHKCR:  CP      (HL)        :IS CHARACTER A CARRIAGE RETURN?
        JR      Z.DONE      :YES. END OF STRING
        INC     B           :NO. ADD 1 TO STRING LENGTH
        INC     HL
        JR      CHKCR
DONE:   LD      A.B
        RET
```

Subroutine Documentation:

```
;

:SUBROUTINE STLEN

;

:PURPOSE: STLEN DETERMINES THE LENGTH OF A
:       STRING (NUMBER OF CHARACTERS BEFORE
:       A CARRIAGE RETURN)

:

:INITIAL CONDITIONS: STARTING ADDRESS OF
,       STRING IN REGISTER PAIR HL

:

:REGISTERS USED: A.B.H.L

.

:SAMPLE CASE:
:       STARTING CONDITIONS: (HL) = 0043
,         (0043) = 35. (0044) = 46. (0045) = 0D
.       FINAL CONDITIONS: (A) = 02

.
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
1) Calling program				
0000	31		LD	SP,80H
0001	80			
0002	00			
0003	2A		LD	HL,(40H)
0004	40			
0005	00			
0006	CD		CALL	STLEN
0007	20			
0008	00			
0009	32		LD	(42H),A
000A	42			
000B	00			
000C	76		HALT	
2) Subroutine				
0020	06	STLEN:	LD	B,0
0021	00			
0022	3E		LD	A,0DH
0023	0D			
0024	BE	CHKCR:	CP	(HL)
0025	28		JR	Z,DONE
0026	04			
0027	04		INC	B
0028	23		INC	HL
0029	18		JR	CHKCR
002A	F9			
002B	78	DONE:	LD	A,B
002C	C9		RET	

The calling program involves four steps: initializing the Stack Pointer, placing the starting address of the string into Register Pair HL, calling the subroutine, and storing the result.

The subroutine is reentrant, since it does not change the contents of any memory locations. It is relocatable, since all the Jump instructions use relative addresses.

The subroutine changes Register B and the address in Register Pair HL as well as the Accumulator. The programmer must be aware that data previously stored in Register B and the address previously loaded into HL will be lost; the subroutine documentation must describe what registers are used.

An alternative to destroying register contents in the subroutine is to save them in the Stack and then restore them before returning. This approach makes the calling routine simpler, but costs extra time and memory (in the program and in the Stack).

This subroutine has a single input parameter, which is an address. The best way to pass this parameter is through a register pair and, since the HL pair is certainly the most flexible as far as addressing options are concerned, it is the obvious choice.

The subroutine contains an unconditional Jump instruction, JR CHKCR. By altering the initial conditions prior to entering the subroutine's loop, can you eliminate this jump?

If the terminating character were not always an ASCII carriage return, we could make that character into another parameter. Now the calling program would have to place

the terminating character into the Accumulator and the starting address of the string into Register Pair HL before calling the subroutine.

One way to pass parameters that do not depend on variable data is to place the values in program memory immediately after the Call instruction. You can use the old Program Counter (saved at the top of the Stack) to access the data, but you must adjust its value properly before returning control to the main program. For example, we could pass the value of the terminating character this way. The main program and subroutine would be:

Calling program:

```
        ORG     0
        LD      SP.80H      ;START STACK AT LOCATION 0080
        LD      (HL),40H    ;GET STARTING ADDRESS OF STRING
        CALL    STLEN       ;DETERMINE STRING LENGTH
        DEFB    '.'         ;TERMINATOR = ASCII PERIOD
        LD      (42H),A     ;STORE STRING LENGTH
        HALT
```

Subroutine:

```
        ORG     20H
STLEN:  POP     DE          ;GET START OF PARAMETER LIST
        LD      A,(DE)      ;GET TERMINATING CHARACTER
        INC     DE          ;ADJUST RETURN ADDRESS
        PUSH    DE
        LD      B,0         ;STRING LENGTH = ZERO
CHKCR:  CP      (HL)        ;IS CHARACTER TERMINATOR?
        JR      Z,DONE      ;YES, END OF STRING
        INC     B           ;NO, ADD 1 TO STRING LENGTH
        INC     HL
        JR      CHKCR
DONE:   LD      A,B
        RET
```

This subroutine is longer and uses Register Pair DE, but the calling program need not load the terminating character into a register. The INC DE instruction is necessary to force a return to the next instruction, rather than to the parameter list.[1]

PUSH and POP transfer the contents of register pairs or index registers to and from the RAM Stack. The eight least significant bits are removed first and stored last to retain consistency with the Z80's upside-down method of storing 16-bit addresses. Remember that the RAM Stack grows downward (to lower addresses).

Add Even Parity to ASCII Characters

Purpose: Add even parity to a string of 7-bit ASCII characters. The length of the string is in the Accumulator and the starting address of the string is in Register Pair HL. Place even parity in the most significant bit of each character, i.e., set the most significant bit to 1 if that makes the total number of 1 bits in the word even.

Sample Problem:

$$
\begin{aligned}
(A) &= 06 \\
(HL) &= 0041 \\
(0041) &= 31 \\
(0042) &= 32 \\
(0043) &= 33 \\
(0044) &= 34 \\
(0045) &= 35 \\
(0046) &= 36
\end{aligned}
$$

Result:
$$
\begin{aligned}
(0041) &= B1 \\
(0042) &= B2 \\
(0043) &= 33 \\
(0044) &= B4 \\
(0045) &= 35 \\
(0046) &= 36
\end{aligned}
$$

Flowchart:

```
              ┌──────────┐
              │  Start   │
              └────┬─────┘
                   │
              ┌────▼─────┐
              │ Pointer = (HL) │
              │ Count   = (A)  │
              └────┬─────┘
                   │
         ┌────────▶│
         │    ◇────▼────◇
   Yes   │   ◇  Does     ◇
   ◀─────┼──◇ (Pointer) have ◇
         │   ◇ even parity ◇
         │    ◇    ?    ◇
         │         │ No
         │    ┌────▼─────┐
         │    │ (Pointer) = (Pointer) │
         │    │ OR 10000000B          │
         │    │ (Set Parity Bit)      │
         │    └────┬─────┘
         │         │
         └───────▶ │
              ┌────▼─────┐
              │ Pointer = Pointer + 1 │
              │ Count   = Count - 1   │
              └────┬─────┘
                   │
              ◇────▼────◇
         No  ◇    Is     ◇
         ◀──◇  Count 0  ◇
              ◇    ?    ◇
                   │ Yes
              ┌────▼─────┐
              │   End    │
              └──────────┘
```
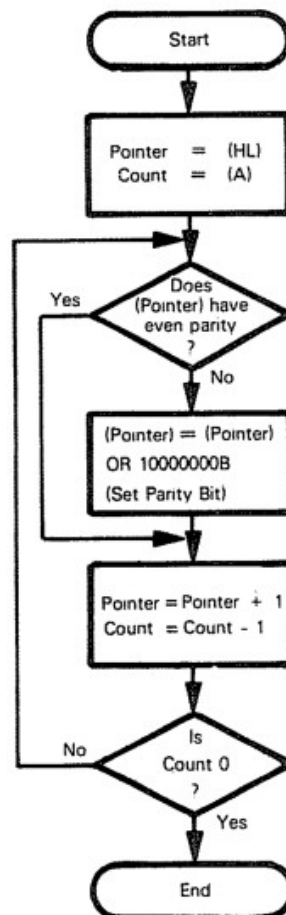
Source Program:

The calling program starts the Stack at memory location 0080, sets the starting address of the string to 0041, gets the string length from memory location 0030, and calls the even parity subroutine.

```
        ORG     0
        LD      SP,80H      ;START STACK AT LOCATION 0080
        LD      HL,41H      ;GET STARTING ADDRESS OF STRING
        LD      A,(30H)     ;GET STRING LENGTH
        CALL    EPAR
        HALT
```

The subroutine adds even parity to a string of ASCII characters.

```
        ORG     20H
EPAR:   LD      B,A
        LD      C,10000000B   ;GET PARITY BIT OF 1
SETPR:  LD      A,(HL)        ;GET A CHARACTER
        OR      C             ;SET PARITY BIT TO 1
        JP      PO,CHCNT      ;IS PARITY NOW EVEN?
        LD      (HL),A        ;YES, SAVE CHARACTER WITH EVEN PARITY
CHCNT:  INC     HL
        DJNZ    SETPR
        HALT
```

Subroutine Documentation:

```
;
;SUBROUTINE EPAR
;
;PURPOSE: EPAR ADDS EVEN PARITY
;   TO A STRING OF 7-BIT ASCII
;   CHARACTERS
;
;INITIAL CONDITIONS: STARTING ADDRESS
;   OF STRING IN HL, LENGTH OF STRING
;   IN A
;
;FINAL CONDITIONS: EVEN PARITY IN
;   MSB OF EACH CHARACTER
;
;REGISTERS USED: A,B,C,H,L
;
;SAMPLE CASE:
;   INITIAL CONDITIONS: (HL) = 0041
;       (A) = 2, (0041) = 32, (0042) = 33
;   FINAL CONDITIONS: (0041) = B2, (0042) = 33
;
```

This subroutine has two parameters, an address and a number. Register Pair HL is used to pass the address and the Accumulator to pass the number. No explicit results are returned, since the subroutine affects only the MSB of each character in the string.

The calling program must place the starting address of the string into Register Pair HL and the length of the string into the Accumulator before transferring control to the subroutine.

The subroutine changes the values in Registers A, H, and L and uses Registers B and C for temporary storage. It is reentrant, since it does not use any fixed memory locations for temporary storage.

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
1) Calling program				
0000	31		LD	SP,80H
0001	80			
0002	00			
0003	21		LD	HL,41H
0004	41			
0005	00			
0006	3A		LD	A,(30H)
0007	30			
0008	00			
0009	CD		CALL	EPAR
000A	20			
000B	00			
000C	76		HALT	
2) Subroutine				
0020	47	EPAR:	LD	B,A
0021	0E		LD	C,10000000B
0022	80			
0023	7E	SETPR:	LD	A,(HL)
0024	B1		OR	C
0025	E2		JP	PO,CHCNT
0026	29			
0027	00			
0028	77		LD	(HL),A
0029	23	CHCNT:	INC	HL
002A	10		DJNZ	SETPR
002B	F7			
002C	C9		RET	

Pattern Match

Purpose: Compare two strings of ASCII characters to see if they are the same. The length of the strings is in the Accumulator. The starting address of one string is in Register Pair HL; the starting address of the other is in Register Pair DE. If the two strings match, clear the Accumulator; otherwise, set the Accumulator to FF (hex).

Sample Problems:

a.

$$(A) = 03$$
$$(DE) = 50$$
$$(HL) = 60$$

$$(0050) = 43 \quad 'C'$$
$$(0051) = 41 \quad 'A'$$
$$(0052) = 54 \quad 'T'$$

$$(0060) = 43 \quad 'C'$$
$$(0061) = 41 \quad 'A'$$
$$(0062) = 54 \quad 'T'$$

Result: (A) = 0, since the strings are the same.

b.

$$(A) = 03$$
$$(DE) = 50$$
$$(HL) = 60$$

$$(0050) = 52 \quad 'R'$$
$$(0051) = 41 \quad 'A'$$
$$(0052) = 54 \quad 'T'$$

$$(0060) = 43 \quad 'C'$$
$$(0061) = 41 \quad 'A'$$
$$(0062) = 54 \quad 'T'$$

Result: (A) = FF (hex), since the first characters differ.

Flowchart:
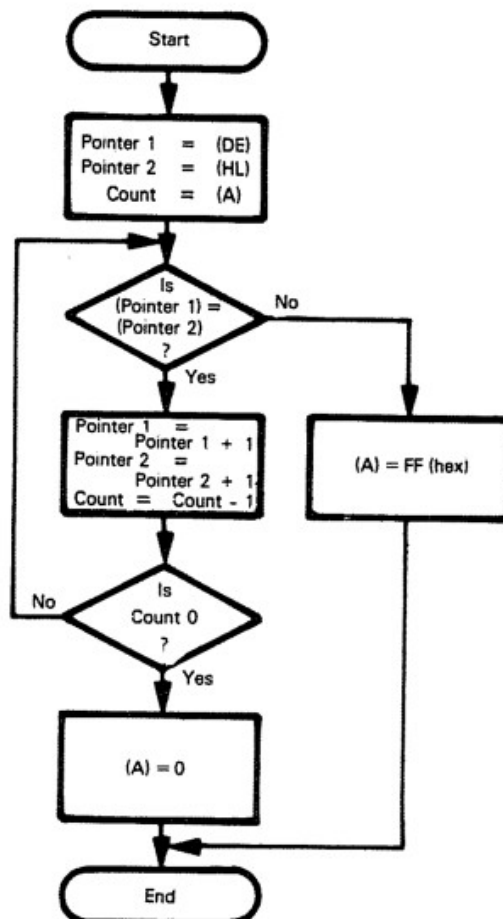
10-13

Source Program:

The calling program starts the Stack at memory location 0080. sets the starting addresses of the strings to 0050 and 0060. respectively. gets the string length from memory location 0040. calls the pattern match subroutine. and places the result into memory location 0041.

```
        ORG     0
        LD      SP.80H      ;START STACK AT LOCATION 0080
        LD      DE.60H      ;GET STARTING ADDRESS OF STRING 1
        LD      HL.50H      ;GET STARTING ADDRESS OF STRING 2
        LD      A.(40H)     ;GET STRING LENGTH
        CALL    PMTCH       ;CHECK FOR MATCH
        LD      (41H).A     ;SAVE MATCH INDICATOR
        HALT
```

The subroutine determines if the two strings are the same.

```
        ORG     20H
PMTCH:  LD      B.A         ;COUNT = STRING LENGTH
        LD      C.0FFH      ;MARK = FF (HEX) FOR NO MATCH
CHCAR:  LD      A.(DE)      ;GET CHARACTER FROM STRING 1
        CP      (HL)        ;IS THERE A MATCH WITH STRING 2?
        JR      NZ.DONE     ;NO. DONE — STRINGS DO NOT MATCH
        INC     DE
        INC     HL
        DJNZ    CHCAR
        LD      C.0         ;MARK = ZERO. STRINGS MATCH
DONE:   LD      A.C
        RET
```

Subroutine Documentation:

;

;SUBROUTINE PMTCH

;

;PURPOSE: PMTCH DETERMINES IF TWO
;   STRINGS ARE EQUIVALENT

;

;INITIAL CONDITIONS: STARTING ADDRESSES
;   OF STRINGS IN DE AND HL.
;   LENGTH OF STRINGS IN ACCUMULATOR

;

;FINAL CONDITIONS: 0 IN A IF
;   STRINGS MATCH. FF IN A OTHERWISE

;

;REGISTERS USED: A.B.D.E.H.L

;

;SAMPLE CASE:
;   STARTING CONDITIONS: (HL) = 0050.
;   (DE) = 0060. (A) = 2
;   (0050) = 36. (0051) = 39
;   (0060) = 36. (0061) = 39
;   FINAL CONDITIONS: (A) = 0 SINCE THE STRINGS MATCH

;

10-14

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
1) Calling program				
0000	31		LD	SP,80H
0001	80			
0002	00			
0003	11		LD	DE,60H
0004	60			
0005	00			
0006	21		LD	HL,50H
0007	50			
0008	00			
0009	3A		LD	A,(40H)
000A	40			
000B	00			
000C	CD		CALL	PMTCH
000D	20			
000E	00			
000F	32		LD	(41H),A
0010	41			
0011	00			
0012	76		HALT	
2) Subroutine				
0020	47	PMTCH:	LD	B,A
0021	0E		LD	C,0FFH
0022	FF			
0023	1A	CHCAR:	LD	A,(DE)
0024	BE		CP	(HL)
0025	20		JR	NZ,DONE
0026	06			
0027	13		INC	DE
0028	23		INC	HL
0029	10		DJNZ	CHCAR
002A	F8			
002B	0E		LD	C,0
002C	00			
002D	79	DONE:	LD	A,C
002E	C9		RET	

This subroutine, like the preceding ones, changes all of the flags. You should generally assume that a subroutine call changes the flags unless it is specifically stated otherwise. If the main program needs the old flag values (for later checking), it must save them in the Stack prior to calling the subroutine. This is accomplished with the PUSH AF instruction.

The subroutine is reentrant and changes all the main registers except C.

This subroutine has three parameters — the two starting addresses and the length of the strings. These parameters use five general-purpose registers.

Multiple-Precision Addition

Purpose: Add two multiple-byte binary numbers. The length of the numbers in bytes is in the Accumulator. The starting addresses of the numbers are in Register Pairs DE and HL. The starting address of the result is in Index Register IX. All the numbers begin with the least significant bits.

Sample Problem:

$$
\begin{aligned}
\text{(A)} &= 04 \\
\text{(DE)} &= 51 \\
\text{(HL)} &= 61 \\
\text{(IX)} &= 71 \\[4pt]
(0051) &= C3 \\
(0052) &= A7 \\
(0053) &= 5B \\
(0054) &= 2F \\
(0061) &= B8 \\
(0062) &= 35 \\
(0063) &= DF \\
(0064) &= 14
\end{aligned}
$$

Result: = (0071) = 7B
(0072) = DD
(0073) = 3A
(0074) = 44

i.e.
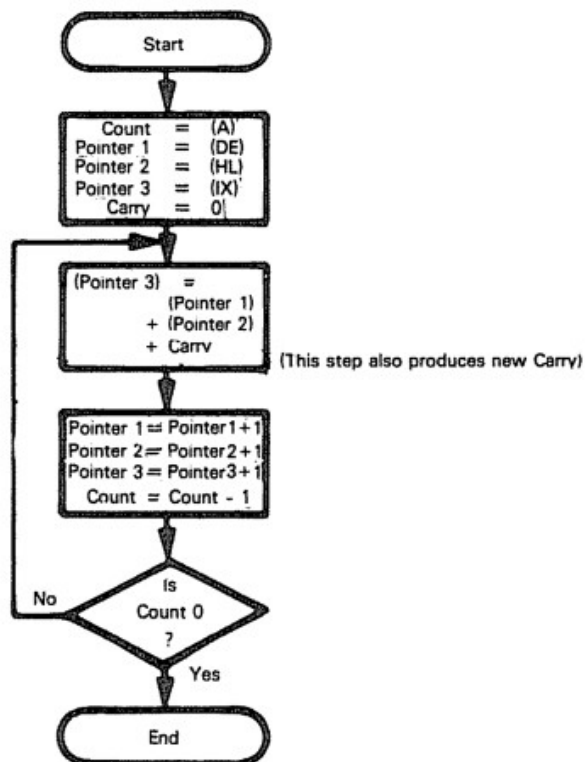
$$
\begin{array}{r}
2F5BA7C3 \\
+\ 14DF35B8 \\
\hline
443ADD7B
\end{array}
$$

Flowchart:

10-16

Source Program:

The calling program starts the Stack at memory location 0080, sets the starting addresses of the various numbers to 0050, 0060, and 0070, respectively, gets the length of the numbers from memory location 0040, and calls the multiple-precision addition subroutine.

```
        ORG     0
        LD      SP,80H      :START STACK AT LOCATION 0080
        LD      HL,50H      :GET STARTING ADDRESS OF FIRST NUMBER
        LD      DE,60H      :GET STARTING ADDRESS OF SECOND NUMBER
        LD      IX,70H      :GET STARTING ADDRESS OF RESULT
        LD      A,(40H)     :GET LENGTH OF NUMBERS IN BYTES
        CALL    MPADD       :MULTIPLE-PRECISION ADDITION
        HALT
```

The subroutine performs multiple-precision binary addition.

```
        ORG     20H
MPADD:  LD      B,A         :COUNT = LENGTH OF NUMBERS IN BYTES
        AND     A           :CLEAR CARRY TO START
ADDW:   LD      A,(DE)      :GET WORD FROM FIRST NUMBER
        ADC     A,(HL)      :ADD WORD FROM SECOND NUMBER
        LD      (IX),A      :STORE ONE WORD OF RESULT
        INC     DE
        INC     HL
        INC     IX
        DJNZ    ADDW
        RET
```

Subroutine Documentation:

:
:SUBROUTINE MPADD
:
:PURPOSE: MPADD ADDS TWO
:   MULTIPLE-BYTE BINARY NUMBERS
:
:INITIAL CONDITIONS: STARTING ADDRESSES
:   OF NUMBERS IN D AND E, H AND L:
:   STARTING ADDRESS OF RESULT IN IX,
:   LENGTH OF NUMBERS IN A
:
:REGISTERS USED: A,B,D,E,H,L,IX
:
:SAMPLE CASE:
:   STARTING CONDITIONS: (HL) = 0050,
:   (DE) = 0060, (IX) = 0070, (A) = 2,
:   (0050) = C3, (0051) = A7, (0060) = B8, (0061) = 35
:   FINAL CONDITIONS: (0070) = 7B, (0071) = DD
:

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
1) Calling program			
0000	31	LD	SP,80H
0001	80		
0002	00		
0003	21	LD	HL,50H
0004	50		
0005	00		
0006	11	LD	DE,60H
0007	60		
0008	00		
0009	DD	LD	IX,70H
000A	21		
000B	70		
000C	00		
000D	3A	LD	A,(40H)
000E	40		
000F	00		
0010	CD	CALL	MPADD
0011	20		
0012	00		
0013	76	HALT	
2) Subroutine			
0020	47	LD	B,A
0021	A7	AND	A
0022	1A	LD	A,(DE)
0023	8E	ADC	A,(HL)
0024	DD	LD	(IX),A
0025	77		
0026	00		
0027	13	INC	DE
0028	23	INC	HL
0029	DD	INC	IX
002A	23		
002B	10	DJNZ	ADDW
002C	F5		
002D	C9	RET	

We use Index Register IX to hold the result address. Try changing the program to use Register Pair BC for this purpose. What happens to the counter?

We could also place the result address at the top of the Stack. The instruction EX (SP),HL exchanges the top of the Stack and Register Pair HL. Change the program so that it uses this instruction, but remember to increment all three pointers after each iteration.

This subroutine has four parameters — three addresses and the length of the numbers. Six 8-bit registers and the 16-bit Index Register IX are used for passing parameters.

PROBLEMS

Note that you are to write both a calling program for the sample problem and a properly documented subroutine.

1) ASCII to Hex

Purpose: Convert the contents of the Accumulator from the ASCII representation of a hexadecimal digit to the 4-bit binary representation of the digit. Place the result into the Accumulator.

Sample Problems:

a.   (A) = 43 'C'

   Result: (A) = 0C

b.   (A) = 36 '6'

   Result: (A) = 06

2) Length of an ASCII Message

Purpose: Determine the length of an ASCII-coded message. The starting address of the string of characters in which the message is located is in Register Pair HL. The message itself starts with an ASCII STX character (hex 02) and ends with ASCII ETX (hex 03). Place the length of the message (the number of characters between the STX and the ETX) into the Accumulator.

Sample Problem:

```
        (HL)  =  0041
      (0041)  =  49
      (0042)  =  02   STX
      (0043)  =  47   'G'
      (0044)  =  4F   'O'
      (0045)  =  03   ETX
```

   Result: (A) = 02

3) Check Even Parity in ASCII Characters

Purpose: Check the even parity of a string of ASCII characters. The length of the string is in the Accumulator and the starting address of the string is in Register Pair HL. If the parity of all the characters in the string is correct, clear the Accumulator; otherwise, set the Accumulator to FF hex (all ones).

Sample Problems:

a.   (A) = 03
                       (HL) = 0042
                     (0042) = B1
                     (0043) = B2
                     (0044) = 33

   Result: (A) = 00, since all the characters have even parity

b.   (A) = 03
                       (HL) = 0042
                     (0042) = B1
                     (0043) = B6
                     (0044) = 33

   Result: (A) = FF, since the character in memory location 0043 does not have even parity

4) String Comparison

Purpose: Compare two strings of ASCII characters to see which is larger (i.e., which would follow the other in 'alphabetical' ordering).

The length of the strings is in the Accumulator; the starting address of string 1 is in Register Pair HL and the starting address of string 2 is in Register Pair DE. If string 1 is larger than or equal to string 2, clear the Accumulator; otherwise, set the Accumulator to FF hex (all ones).

Sample Problems:

a.
$$(A) = 03$$
$$(DE) = 0060$$
$$(HL) = 0050$$
$$(0050) = 43 \quad 'C'$$
$$(0051) = 41 \quad 'A'$$
$$(0052) = 54 \quad 'T'$$

$$(0062) = 42 \quad 'B'$$
$$(0063) = 41 \quad 'A'$$
$$(0064) = 54 \quad 'T'$$

Result = (A) = 00, since CAT is 'larger' than BAT

b.
$$(A) = 03$$
$$(DE) = 0060$$
$$(HL) = 0050$$
$$(0050) = 44 \quad 'D'$$
$$(0051) = 4F \quad 'O'$$
$$(0052) = 47 \quad 'G'$$

$$(0060) = 44 \quad 'D'$$
$$(0061) = 4F \quad 'O'$$
$$(0062) = 47 \quad 'G'$$

Result = (A) = 00, since the two strings are equal

c.
$$(A) = 03$$
$$(DE) = 0060$$
$$(HL) = 0050$$
$$(0050) = 43 \quad 'C'$$
$$(0051) = 41 \quad 'A'$$
$$(0052) = 54 \quad 'T'$$

$$(0060) = 43 \quad 'C'$$
$$(0061) = 55 \quad 'U'$$
$$(0062) = 54 \quad 'T'$$

Result = (A) = FF (hex), since CUT is 'larger' than CAT

5) Decimal Subtraction

Purpose: Subtract one multiple-digit decimal (BCD) number from another. The length of the numbers (in bytes) is in the Accumulator and the starting addresses of the numbers are in Register Pairs DE and HL. Subtract the number with the starting address in HL from the one with the starting address in DE. The starting address of the result is in Index Register IX. All the numbers begin with the least significant digits. The sign of the result is returned in the Accumulator — zero if the result is positive, FF (hex) if it is negative.

Sample Problem:

$$(A) = 04$$
$$(DE) = 0050$$
$$(HL) = 0060$$
$$(IX) = 0070$$

$$(0050) = 85$$
$$(0051) = 19$$
$$(0052) = 70$$
$$(0053) = 36$$
$$(0060) = 59$$
$$(0061) = 34$$
$$(0062) = 66$$
$$(0063) = 12$$

Result:
$$(A) = 00 \text{ (positive)}$$
$$(0070) = 26$$
$$(0071) = 85$$
$$(0072) = 03$$
$$(0073) = 24$$

i.e.,
```
      36701985
  -   12663459
  +   24038526
```

REFERENCES

1. Other examples of this technique (for the 8080 microprocessor) are in S. Mazor and C. Pitchford, "Develop Cooperative Microprocessor Subroutines," _Electronic Design_, June 7, 1978, pp. 116-118.

Chapter 11
INPUT/OUTPUT

There are two problems in the design of input/output sections: one is how to interface peripherals to the computer and transfer data, status, and control signals; the other is how to address I/O devices so that the CPU can select a particular one for a data transfer. Clearly, the first problem is both more complex and more interesting. We will therefore discuss the interfacing of peripherals here and leave addressing to a more hardware-oriented book.

In theory, the transfer of data to or from an I/O device is similar to the transfer of data to or from memory. In fact, we can consider the memory as just another I/O device. The memory is, however, special for the following reasons:

| I/O AND |
| MEMORY |

1) It operates at almost the same speed as the processor.

2) It uses the same type of signals as the CPU. The only circuits usually needed to interface the memory to the CPU are drivers, receivers, and level translators.

3) It requires no special formats or any control signals besides a Read/Write pulse.

4) It automatically latches data sent to it.

5) Its word length is the same as the computer's.

Most I/O devices do not have such convenient features. They may operate at speeds much slower than the processor; for example, a teletypewriter can transfer only 10 characters per second, while a slow processor can transfer 10,000 characters per second. The range of speeds is also very wide — sensors may provide one reading per minute, while video displays or floppy disks may transfer 250,000 bits per second. Furthermore, I/O devices may require continuous signals (motors or thermometers), currents rather than voltages (teletypewriters), or voltages at far different levels than the signals used by the processor (gas-discharge displays). I/O devices may also require special formats, protocols, or control signals. Their word lengths may be much shorter or much longer than the word length of the computer. These variations make the design of I/O sections difficult and mean that each peripheral presents its own special interfacing problem.

We may, however, provide a general description of devices and interfacing methods. We may roughly separate devices into three categories, based on their data rates:

| I/O |
| CATEGORIES |

1) Slow devices that change state no more than once per second. Changing their states typically requires milliseconds or longer. Such devices include lighted displays, switches, relays, and many mechanical sensors and actuators.

2) Medium-speed devices that transfer data at rates of 1 to 10,000 bits per second. Such devices include keyboards, printers, card readers, paper tape readers and punches, cassettes, ordinary communications lines, and many analog data acquisition systems.

3) High-speed devices that transfer data at rates of over 10,000 bits per second. Such devices include magnetic tapes, magnetic disks, high-speed line printers, high-speed communications lines, and video displays.

The interfacing of slow devices is simple. Few control signals are necessary unless the devices are multiplexed, i.e., several are handled from one port, as shown in Figures 11-1 to 11-4.

Input data from slow devices need not be latched, since it remains stable for a long time interval. Output data must, of course, be latched. The only problems with input are transitions that occur while the computer is reading the data. One-shots, cross-coupled latches, or software delay routines can smooth the transitions.

A single port can handle several slow devices. Figure 11-1 shows a demultiplexer that automatically directs the next output data to the next device by counting output operations. Figure 11-2 shows a control port that provides select inputs to a demultiplexer. The data outputs here can come in any order, but an additional output instruction is necessary to change the state of the control port. Output demultiplexers are commonly used to drive several displays from the same output port. Figures 11-3 and 11-4 show the same alternatives for an input multiplexer.

Note the differences between input and output with slow devices:

1) Input data need not be latched, since the input device holds the data for an enormous length of time by computer standards. Output data must be latched, since the output device will not respond to data that is present for only a few CPU clock cycles.

2) Input transitions cause problems because of their duration; brief output transitions cause no problems because the output devices (or the observers) react slowly.

3) The major constraints on input are reaction time and responsiveness, the major constraints on output are response time and observability.

Medium-speed devices must be synchronized in some way to the processor clock. The CPU cannot simply treat these devices as if they held their data forever or could receive data at any time. Instead, the CPU must be able to determine when a

device has new input data or is ready to receive output data. It must also have a way of telling a device that new output data is available or that the previous input data has been accepted. Note that the peripheral may be or contain another processor.

The standard unclocked procedure is the handshake. Here the sender indicates the availability of data to the receiver and

transfers the data; the receiver completes the handshake by acknowledging the reception of the data. The receiver may control the situation by initially requesting the data or by indicating its readiness to accept data; the sender then sends the data and completes the handshake by indicating that data is available. In either case, the sender knows that the transfer has been completed successfully and the receiver knows when new data is available.

The Counter controls where the Demultiplexer sends the data.
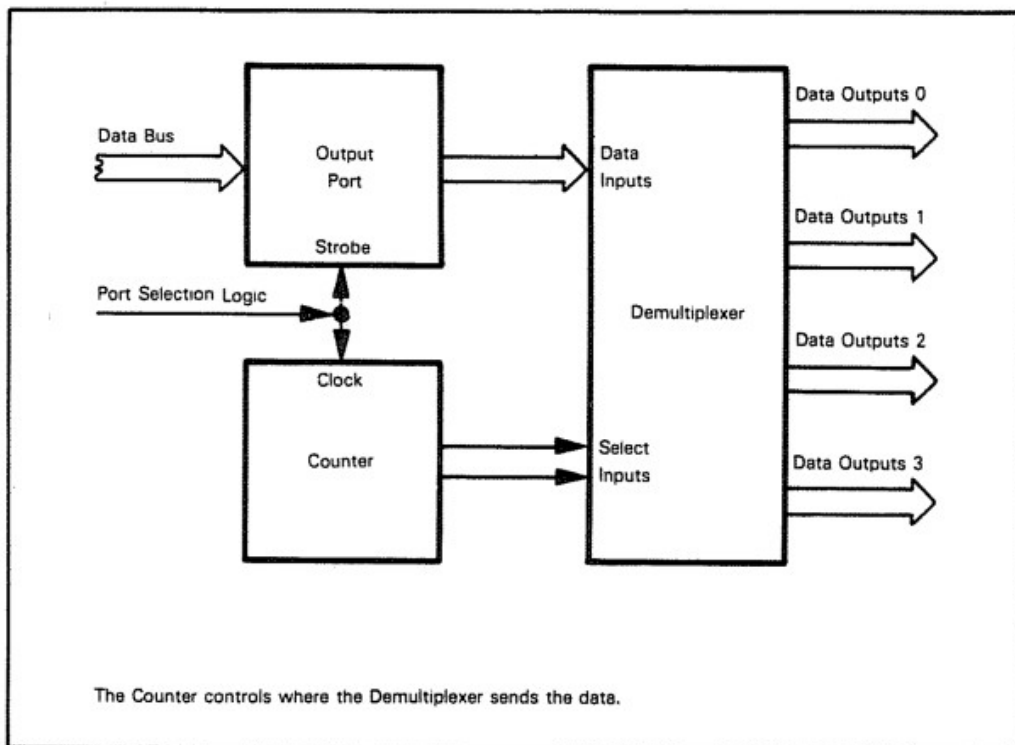
Figure 11-1. An Output Demultiplexer Controlled by a Counter

The CPU sends control information to the Control Port; that port determines where the Demultiplexer sends the data.
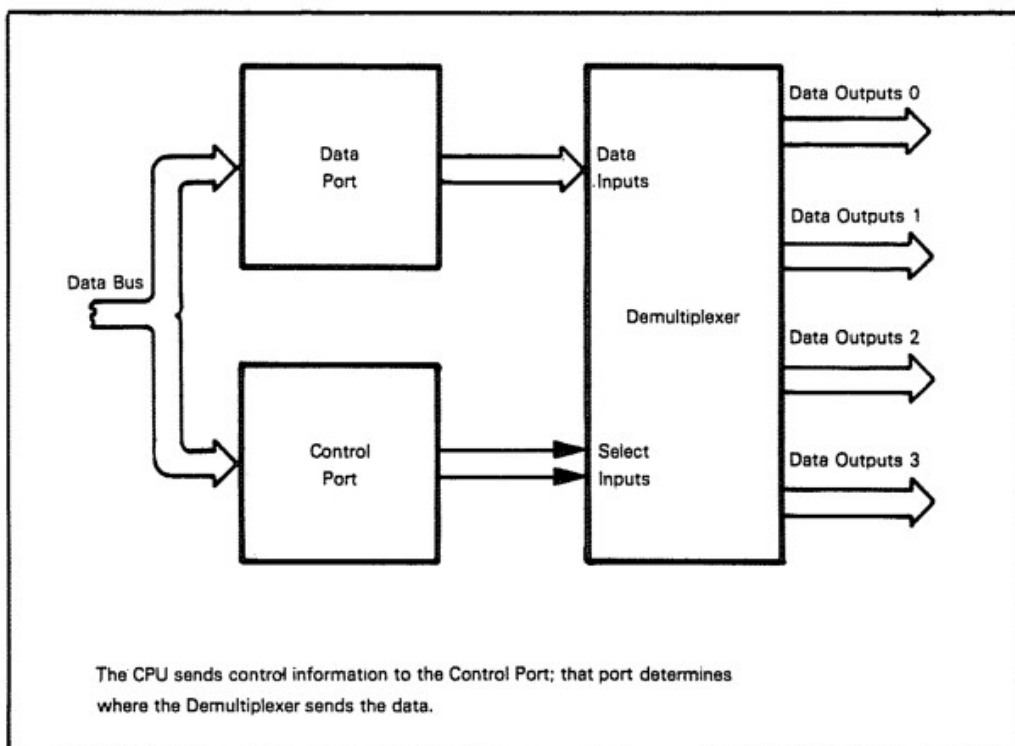
Figure 11-2. An Output Demultiplexer Controlled by a Port

The Counter controls which input the Multiplexer gates to the Input Port.
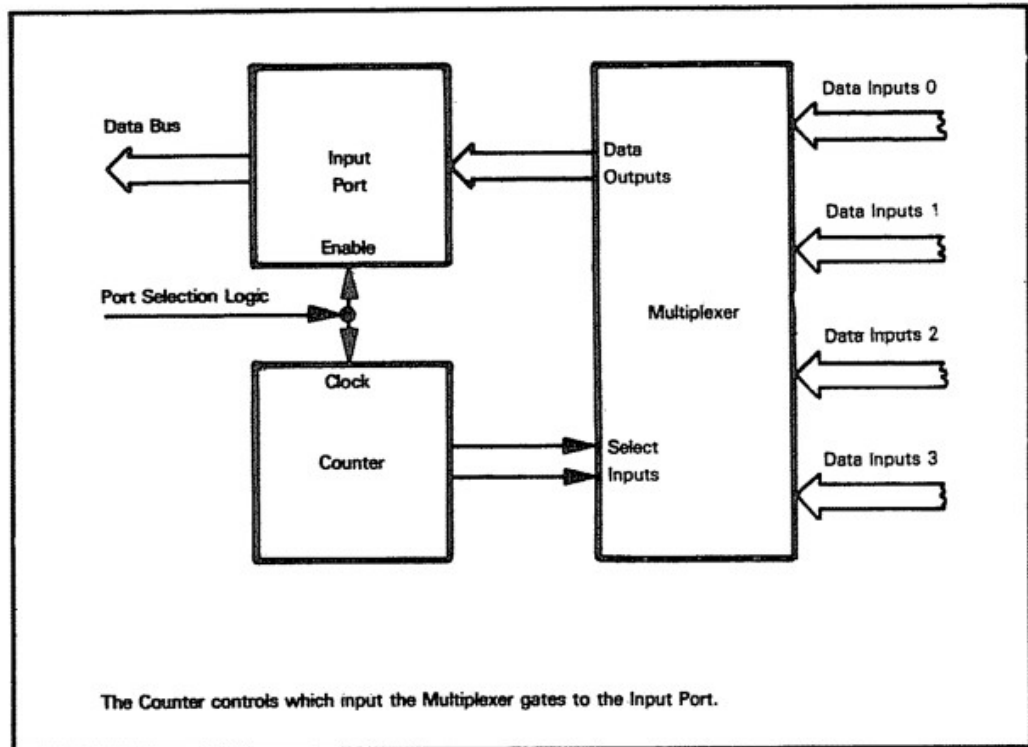
Figure 11-3. An Input Multiplexer Controlled by a Counter

The control information which the CPU sends to the Control Port (with an output operation) determines which input the Multiplexer routes to the Data Port.
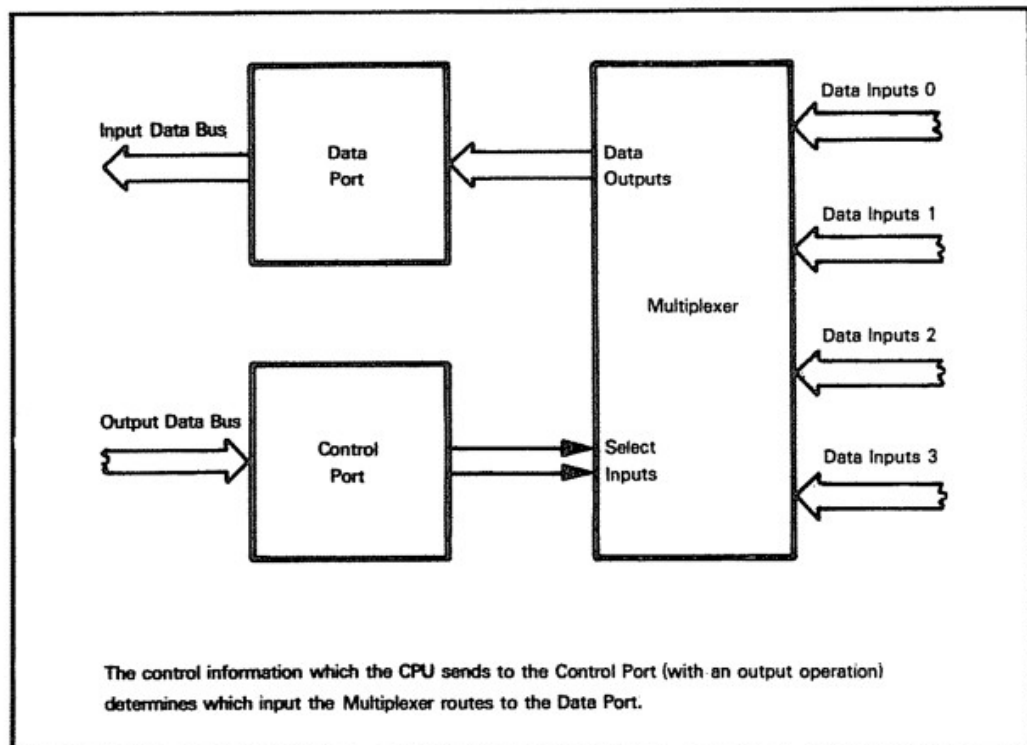
Figure 11-4. An Input Multiplexer Controlled by a Port

Figures 11-5 and 11-6 show typical input and output operations using the handshake method. The procedure whereby the CPU checks the readiness of the peripheral before transferring data is called "polling" Clearly, polling can occupy a large amount of processor time if there are many I/O devices. There are several ways of providing the handshake signals. Among these are:

- Separate dedicated I/O lines. The processor may handle these as additional I/O ports or through special lines or interrupts. The Z80 processor does not have serial I/O lines, but the Z80 Parallel Input/Output device (or PIO) does.

- Special patterns on the I/O lines. These may be single start and stop bits or entire characters or groups of characters. The patterns must be easy to distinguish from background noise or inactive states.

We often call a separate I/O line that indicates the availability of data or the occurrence of a transfer a "strobe". A strobe may, for example, clock data into a latch or fetch data from a buffer.

> **STROBE**

Many peripherals transfer data at regular intervals; i.e., synchronously. Here the only problem is starting the process by lining up to the first input or marking the first output. In some cases, the peripheral provides a clock input from which the processor can obtain timing information.

Transmission errors are a problem with medium-speed devices. Several methods can lessen the likelihood of such errors; they include:

> **REDUCING TRANSMISSION ERRORS**

- Sampling input data at the center of the transmission interval in order to avoid edge effects; that is, keep away from the edges where the data is changing.

- Sampling each input several times and using majority logic such as best three out of five.[1]

- Generating and checking parity; an extra bit is used that makes the number of 1 bits in the correct data even or odd.

- Using other error detecting and correcting codes such as checksums, LRC (longitudinal redundancy check), and CRC (cyclic redundancy check).[2]

High-speed devices that transfer more than 10,000 bits per second require special methods. The usual technique is to construct a special-purpose controller that transfers data directly between the memory and the I/O device. This process is called direct memory access (DMA). The DMA controller must force the CPU off the busses, provide addresses and control signals to the memory, and transfer the data. Such a controller will be fairly complex, typically consisting of 50 to 100 chips, although LSI devices are now available.[3] The CPU must initially load the Address and Data Counters in the controller so that the controller will know where to start and how much to transfer.

> **INTERFACING HIGH-SPEED DEVICES**
>
> **DIRECT MEMORY ACCESS**

a) Peripheral provides data and Data Ready signal to computer I/O section.

b) CPU reads Data Ready signal from I/O section (this may be a hardware interrupt connection).

c) CPU reads data from I/O section.

d) CPU sends Input Acknowledge signal to I/O section, which then provides Input Acknowledge signal to Peripheral (this may be a hardware connection).
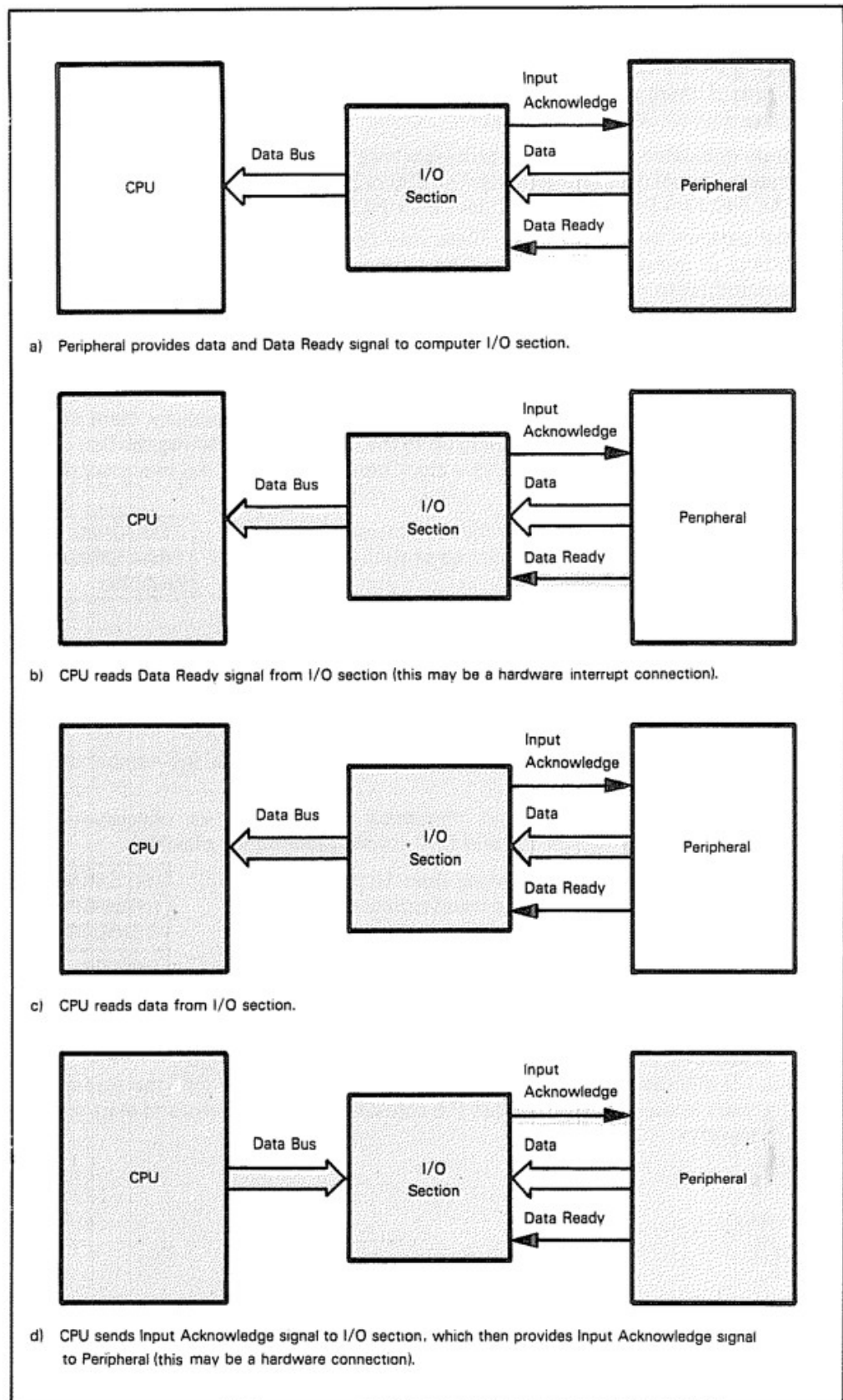
Figure 11-5. An Input Handshake
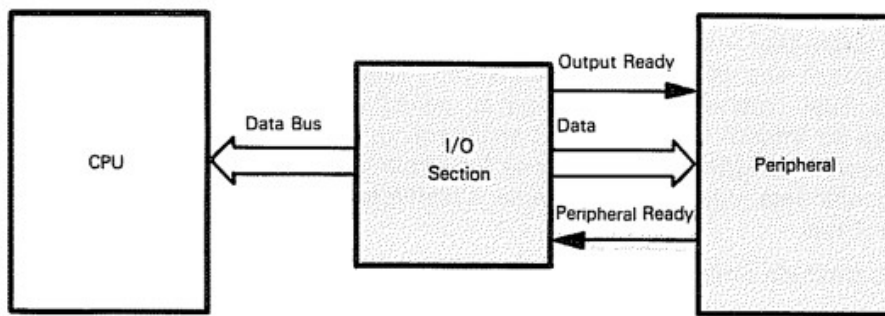
a) Peripheral provides Peripheral Ready signal to computer I/O section.
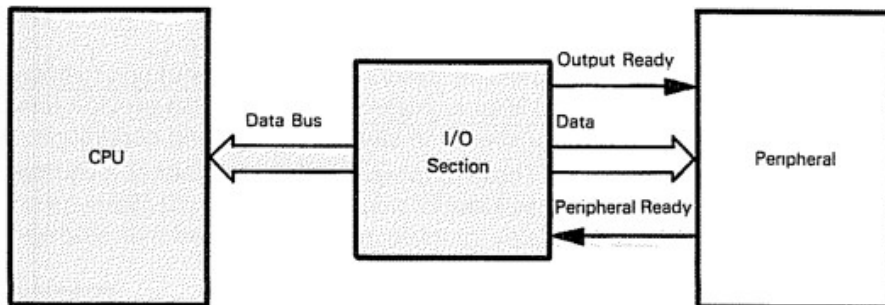
b) CPU reads Peripheral Ready signal from I/O section (this may be a hardware interrupt connection).
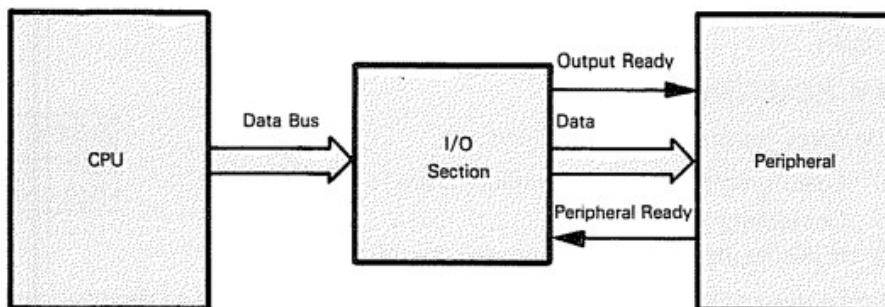
c) CPU sends data to Peripheral.
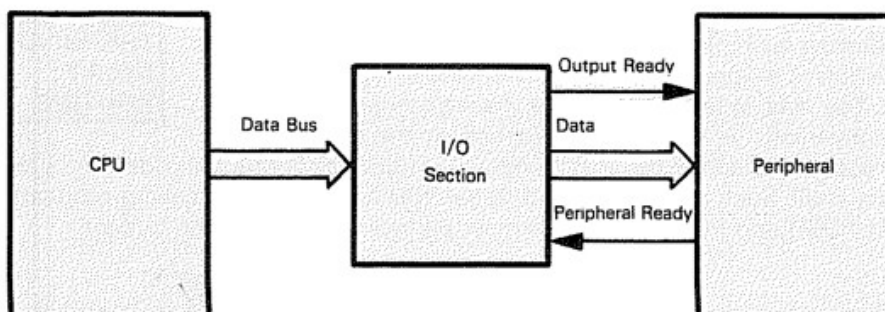
d) CPU sends Output Ready signal to Peripheral (this may be a hardware connection).

Figure 11-6. An Output Handshake

TIMING INTERVALS (DELAYS)

One problem that we will face throughout the discussion of input/output is the generation of timing intervals with specific lengths. Such intervals are necessary to debounce mechanical switches (to smooth their irregular transitions), to provide pulses

USES OF TIMING INTERVALS

with specified lengths and frequencies for displays, and to provide timing for devices that transfer data regularly (for example, a teletypewriter that sends or receives one bit every 9.1 ms).

We can produce timing intervals in several ways:

METHODS FOR PRODUCING TIMING INTERVALS

1) In hardware with one-shots or monostable multivibrators. These devices produce a single pulse of fixed duration in response to a pulse input.

2) In a combination of hardware and software with a flexible programmable timer such as the Z80 Counter-Timer Circuit (or CTC) for Z80 based microcomputers, as described in <u>An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors</u>. The CTC can provide timing intervals of various lengths with a variety of starting and ending conditions.

3) In software with delay routines. These routines use the processor as a counter. This is possible since the processor has a stable clock reference, but it clearly under-utilizes the processor. However, delay routines require no additional hardware and often use processor time that would otherwise be wasted.

The choice among these three methods depends on your application. The software method is inexpensive but may overburden the processor. The programmable timers are relatively expensive, but are easy to interface and may be able to handle many complex timing tasks.

CHOOSING A TIMING METHOD

DELAY ROUTINES

A simple delay routine works as follows:

BASIC SOFTWARE DELAY

Step 1)   Load a register with a specified value.

Step 2)   Decrement the register.

Step 3)   If the result of Step 2 is not zero, repeat Step 2.

This routine does nothing except use time. The amount of time used depends upon the execution time of the various instructions. The maximum length of the delay is limited by the size of the register; however, the entire routine can be placed inside a similar routine that uses another register, and so on.

The following example uses Register C and the Accumulator to provide delays as long as 255 ms. The choice of registers is arbitrary. You may, in fact, find the use of a register pair (e.g., BC) more convenient. A PUSH BC instruction at the start of the

TRANSPARENT DELAY ROUTINE

delay routine and a POP BC at the end will result in a routine that does not affect any registers at all. Such a routine is said to be "transparent" to the calling program. Note that the PUSH and POP instructions must be included in the time budget.

EXAMPLE

Delay Program Using Accumulator

Purpose: The program provides a delay of 1 ms times the contents of Accumulator.

Flowchart:

```
                    ┌─────────┐
              ┌────►│  Start  │
              │     └────┬────┘
              │          │
              │     ┌────▼────────┐
              │     │Count = MSCNT│
              │     └────┬────────┘
              │          │
              │   ┌──────▼───────────┐
              │   │Count = Count - 1 │
              │   └──────┬───────────┘
              │          │
              │        ┌─▼─┐
         No   │       ╱ Is  ╲
          ────┤      ◄Count 0►
              │       ╲  ?  ╱
              │        └─┬─┘
              │          │ Yes
              │   ┌──────▼───────┐
              │   │ (A) = (A) - 1│
              │   └──────┬───────┘
              │          │
              │        ┌─▼─┐
         No   │       ╱ Is  ╲
          ────┘      ◄ (A) 0 ►
                      ╲  ?  ╱
                       └─┬─┘
                         │ Yes
                    ┌────▼────┐
                    │   End   │
                    └─────────┘
```
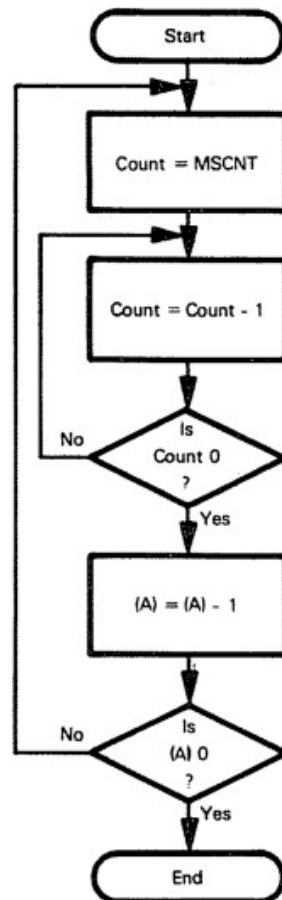
The value of MSCNT depends on the speed of the CPU and the memory cycle.

Source Program:

```
DELAY:  LD    C.MSCNT      :GET COUNT FOR 1 MS DELAY
DLY1:   DEC   C            :COUNT = COUNT -1
        JR    NZ,DLY1      :CONTINUE UNTIL COUNT = ZERO
        DEC   A            :DECREMENT NUMBER OF REMAINING MS
        JR    NZ,DELAY     :CONTINUE UNTIL NUMBER OF MS = ZERO
        RET
```

Object Program:

Memory Location (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0030	0E	DELAY:	LD	C,MSCNT
0031	MSCNT			
0032	0D	DLY1:	DEC	C
0033	20		JR	NZ,DLY1
0034	FD			
0035	3D		DEC	A
0036	20		JR	NZ,DELAY
0037	F8			
0038	C9		RET	

Time Budget:

Instruction		Number of Times Executed
LD	C.MSCNT	(A)
DEC	C	(A) x MSCNT
JR	NZ,DLY1	(A) x MSCNT
DEC	A	(A)
JR	NZ,DELAY	(A)
RET		

The total time used should be (A) x 1 ms. If the memory is operating at full speed, the instructions require the following numbers of clock cycles.

LD	C,MSCNT	7
DEC	C or DEC A	4
JR	NZ	7 or 12
RET		10

The alternative times for JR are for the condition being met (12) or not met (7).

Ignoring the CALL and RET instructions (which occur only once), the program takes:

$$(A) \times (7 + 16 \times MSCNT - 5 + 16) - 5$$

clock cycles. The -5's are caused by the fact that JR takes less time during the final iteration when the condition is not met.

So, to make the delay 1 ms.

$$13 + 16 \times MSCNT = N_C$$

where N_C is the number of clock cycles per millisecond. At the standard 4 MHz Z80 clock rate, $N_C = 4000$, so:

Z80 DELAY LOOP CONSTANT

$$16 \times MSCNT = 3987$$

MSCNT = 249 (hex F9) at a Z80 clock rate of 4 MHz

SIMPLE I/O DEVICES

THE Z80 PARALLEL INPUT/OUTPUT CIRCUIT (PIO)

The key element in most Z80 input/output sections is the Z80 Parallel Input/Output Circuit or PIO. This device combines latches, buffers, flip-flops, and other logic circuits needed for handshaking and other simple interfacing techniques. The PIO contains many logic connections, certain sets of which can be selected according to the contents of programmable registers. Thus, the designer has the equivalent of a Circuit Designer's Casebook under his control. The initialization phase of the program places the appropriate values into registers to select the required logic connections. An input/output section based on PIOs can handle many different applications, and changes or corrections can be made in software rather than by rewiring.

Figure 11-7 is the block diagram of a PIO. The device contains two nearly identical 8-bit ports — A, which is usually an input port, and B, which is usually an output port. Each port (see Figure 11-8) contains:

- An 8-bit Data Output register
- An 8-bit Data Input register
- A 2-bit Mode Control register, which indicates whether the port is in an output, input, bidirectional, or control mode

| PIO REGISTERS AND CONTROL LINES |

- An 8-bit Input/Output Control register, which determines whether the corresponding data pins are inputs (1) or outputs (0) in the control mode
- Two control lines (STB and RDY) that are configured by the Mode Control register. These lines can be used for the handshaking signals shown in Figures 11-5 and 11-6.
- A 2-bit Mask Control register (used only in the control mode) that determines the active polarity of the inputs and whether they will be logically ORed or ANDed to form an interrupt signal
- An 8-bit Mask register (used only in the control mode) that determines which port lines will be monitored to form the interrupt signal
- An 8-bit Vector Address register used with the interrupt system

For now, we will be concerned only with the Mode Control registers, the Input/Output Control registers, and the control lines. We will discuss the interrupt-related features of the PIO in Chapter 12.

The meanings of the bits in the various control and mask registers are related to the underlying hardware and are entirely arbitrary as far as the assembly language programmer is concerned. You must either memorize them or look them up in this chapter and in Chapter 12.

Each PIO occupies four input port addresses and four output port addresses. The B/A SEL (Port B or A select) and C/D SEL (Control or Data Select) lines choose one of the four ports as described in

| PIO ADDRESSES |

Table 11-1. Most often, designers attach address bit A_0 to the B/A SEL input and address bit A_1 to the C/D SEL input. The PIO then occupies four consecutive port addresses as described in the last column of Table 11-1.

Clearly there are far more internal control registers than there are port addresses for them. In fact, all the control registers for each port occupy one address according to the C/D SEL connection. So some of the data bits sent to a control register are actually used for addressing purposes. Note the following situations (see Table 11-2):

$D_0 = 0$ means that the remaining data bits are loaded into the Interrupt Vector register.
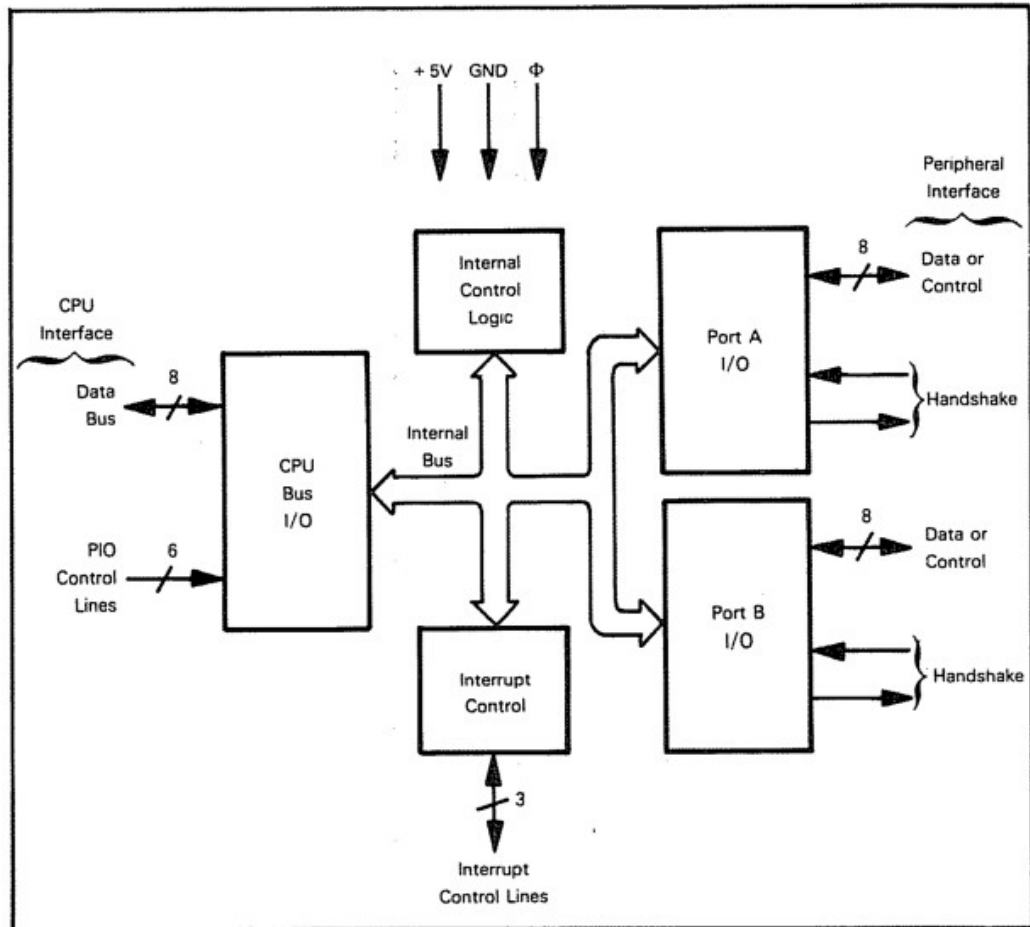
Figure 11-7. PIO Block Diagram
(Courtesy of Zilog)
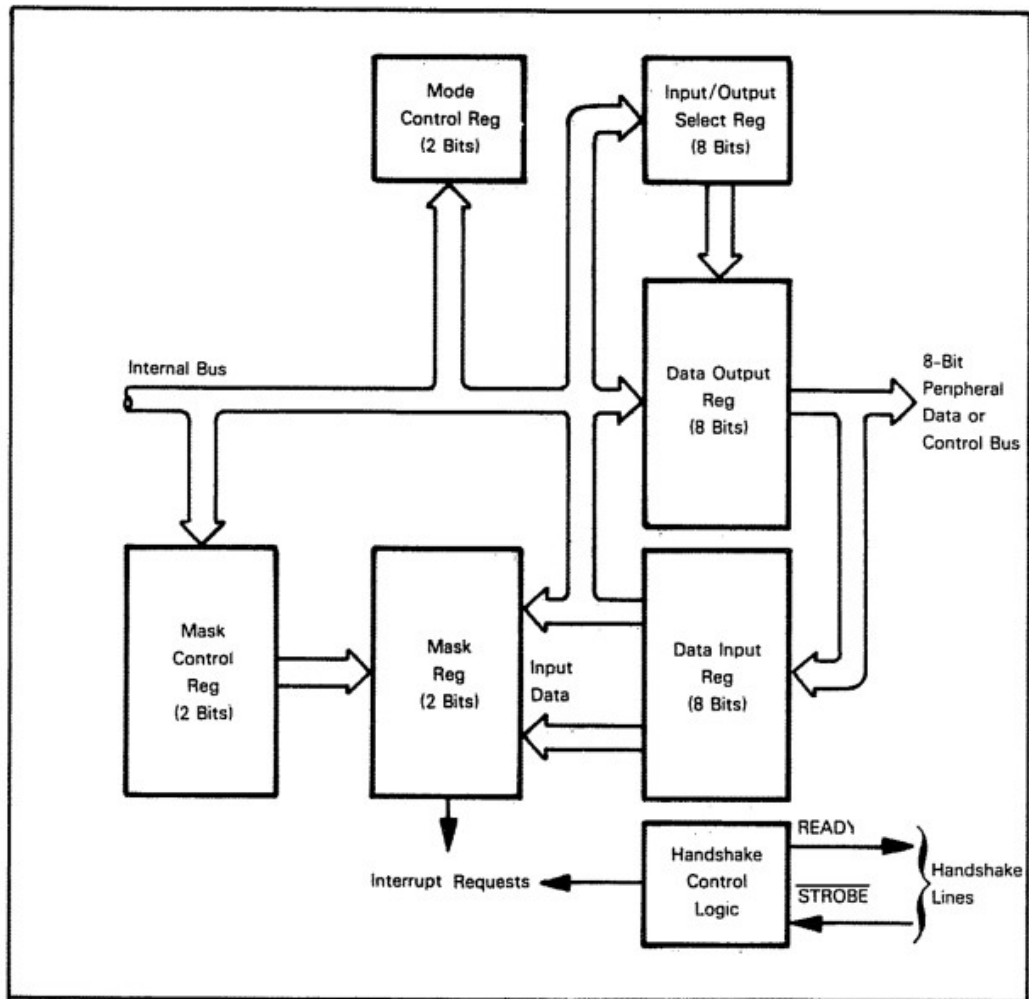
Figure 11-8. Block Diagram of PIO Port
(Courtesy of Zilog)

Table 11-1. PIO Addresses

CONTROL OR DATA SELECT	PORT B OR A SELECT	REGISTER ADDRESSED	PORT ADDRESS (STARTING WITH PIOADD)
0	0	Data Register A	PIOADD
0	1	Data Register B	PIOADD+1
1	0	Control A	PIOADD+2
1	1	Control B	PIOADD+3
The port addresses assume that C/D SEL is tied to A_1 and B/A SEL to A_0.			

Table 11-2. Addressing of PIO Control Registers

REGISTER	ADDRESSING
MODE CONTROL	$D_3 = D_2 = D_1 = D_0 = 1$
INPUT/OUTPUT CONTROL	NEXT WORD AFTER MODE CONTROL SETS MODE 3
MASK CONTROL REGISTER	$D_3 = 0, D_2 = D_1 = D_0 = 1$
INTERRUPT MASK REGISTER	NEXT WORD AFTER MASK CONTROL REGISTER ACCESSED WITH $D_4 = 1$
INTERRUPT ENABLE	$D_3 = D_2 = 0, D_1 = D_0 = 1$
INTERRUPT VECTOR	$D_0 = 1$

$D_3 = 0, D_2 = D_1 = D_0 = 1$ means that the remaining data bits are loaded into the Mask Control register. If $D_4 = 1$, the next control word is loaded into the Interrupt Mask register. Interrupts can be enabled or disabled with $D_3 = D_2 = 0, D_1 = D_0 = 1$.

$D_3 = D_2 = D_1 = D_0 = 1$ means that the remaining data bits are loaded into the Mode Control register. If $D_7 = D_6 = 1$ (control mode), the next control word is loaded into the Input/Output Control register.

This sharing of an external address means that:

1) The programmer must be very careful of the order of operations. The meaning of a particular Output instruction depends on the sequence in which it occurs.

2) The programmer should document the PIO configuration in detail. The device is complex, and a reader is unlikely to be able to make much sense out of the sequence of operations that configures it.

We should note that one usually configures the control registers of the PIO just once in the initialization phase of the program. The rest of the program then uses only the PIO data registers.

PIO MODE CONTROL

The mode of operation of a PIO is established by writing a control word to the PIO in the form shown in Figure 11-1. Table 11-3 describes the meanings of the various modes and the control words required to establish them. Note that bits D_5 and D_4 are not used. When power is turned on, the PIO comes up in mode 1 (input).

PIO MODES

We may summarize the modes as follows:

1) Mode 0 — OUTPUT

PIO OUTPUT MODE

Writing data into the port Output register latches the data and causes it to appear on the port Data Bus. The READY (RDY) line goes high to indicate Data Ready; it remains high until the peripheral sends a rising edge on the STROBE ($\overline{STB}$) line to indicate Data Accepted or Device Ready. The rising edge of $\overline{STB}$ causes an interrupt if the interrupt has been enabled.

2) Mode 1 — INPUT

PIO INPUT MODE

The peripheral latches data into the port Input register using the STROBE signal. The rising edge of $\overline{STB}$ causes an interrupt (if enabled) and deactivates RDY. When the CPU reads the data, RDY goes high to indicate Data Accepted or Input Register Empty. Note that the peripheral can strobe data into the register regardless of the state of RDY. The programmer must thus handle the problem of overrun, i.e., new data being placed into the register before the old data is read.

3) Mode 2 — BIDIRECTIONAL

PIO BIDIRECTIONAL MODE

This mode uses all four handshake lines, so it is allowed only on Port A. The Port A RDY and $\overline{STB}$ signals are used for output control and the Port B RDY and $\overline{STB}$ signals are used for input control. The only difference between this mode and a combination of modes 0 and 1 is that data from the Port A Output register is enabled onto the port Data Bus only when $\overline{A\ STB}$ is active. This allows the Port A bus to be used bidirectionally under the control of $\overline{A\ STB}$ (Output Data Request) and $\overline{B\ STB}$ (Input Data Available). Note that the B side control signals are governed by Input Register A in this mode.

4) Mode 3 — CONTROL

PIO CONTROL MODE
PIO DIRECTIONS IN CONTROL MODE

This mode does not use the RDY and $\overline{STB}$ signals. It is intended for status and control applications in which each bit has an individual meaning. When mode 3 is selected, the next control word sent to the PIO defines the directions of the port data bits (Figure 11-9). A '1' in a bit position makes the corresponding bus line an input, while a '0' makes it an output.

M1	M0	Mode
0	0	Output
0	1	Input
1	0	Bidirectional
1	1	Bit Control

M1	M0	X	X	1	1	1	1

When selecting Mode 3, the next word must
set the I/O Register:

I/O7	I/O6	I/O5	I/O4	I/O3	I/O2	I/O1	I/O0

I/O = 1 Sets bit to Input
I/O = 0 Sets bit to Output

PIO Mode	Meaning	Control Word	
		(Binary)	(Hex)
0	Output	00001111	0F
1	Input	01001111	4F
2	Bidirectional	10001111	8F
3	Control	11001111	CF

Note that bits 4 and 5 are not used and could
have any values.

Figure 11-9. Mode Control for the Z80 PIO

Note the following features of the PIO modes:

FEATURES OF PIO MODES

1) In modes 0,1, and 2 the peripheral indicates Data Ready,
Device Ready, or Data Accepted with a rising edge on the
$\overline{STB}$ line. This edge also causes an interrupt if the interrupt is enabled.

2) In modes 0,1, and 2 the PIO indicates Data Ready, Input Buffer Empty, or Data Accepted by sending RDY high. This signal remains high until the next rising edge on $\overline{STB}$.

3) Only Port A can be used bidirectionally. If Port A is in mode 2 (bidirectional), Port B can only be in mode 3 (control) since no handshake lines are available.

4) The control mode (3) is the only mode in which the Input/Output Control register is used. Otherwise, the entire port is used for either input or output.

5) There is no way for the processor to determine if a pulse has occurred on $\overline{STB}$ if interrupts are not being used. The PIO is designed for use in interrupt-driven rather than polling systems (see Chapter 12). $\overline{STB}$ should be tied low if it is not being used.

6) The processor cannot directly control the RDY lines. The RDY line on a port goes high when data is transferred to or from the port and goes low on the rising edge of $\overline{STB}$.

7) The contents of the data Output register can be read if the port is in the output or bidirectional mode. If the port is in the control mode, the output register data from the lines assigned as outputs can be read. The contents of control registers cannot be read.

8) If the RDY output is tied to the $\overline{STB}$ input on a port in the output mode, RDY will go high for one clock period after each output operation. This brief pulse can be used to multiplex displays as shown in Figure 11-1.

CONFIGURING THE PIO

The program must select the logic connections in the PIO before transferring data to or from it. This selection (or configuration) is usually part of the startup routine. Note that the PIO comes up in the input mode with all interrupts disabled and inhibited and control signals deactivated (low) when power is turned on. However, the PIO does not have a RESET input and does not necessarily return to the reset state when the CPU is reset. The steps in PIO configuration are:

1) Establish the mode of operation by writing the appropriate control words to the Mode Control register. Interrupt control as well as I/O mode information may have to be sent.

> **STEPS IN CONFIGURING A PIO**

2) If in mode 3, establish the directions of the I/O pins by writing a control word to the Input/Output Control register. This word must follow the control word that selected mode 3.

Let us now look at some examples of configuring a PIO without interrupts:

1) OUTPUT PORT

```
        LD      A,00001111B     :MAKE PORT B OUTPUT
        OUT     (PIOCRB),A
```

2) INPUT PORT

```
        LD      A,01001111B     :MAKE PORT A INPUT
        OUT     (PIOCRA),A
```

3) BIDIRECTIONAL PORT

```
        LD      A,10001111B     :MAKE PORT A BIDIRECTIONAL
        OUT     (PIOCRA),A
```

Remember that only Port A can be bidirectional and that Port B must then be a control port.

4) CONTROL PORT, ALL INPUTS

```
        LD      A,11001111B     :MAKE PORT A CONTROL
        OUT     (PIOCRA),A
        LD      A,0FFH          :ALL BITS INPUTS
        OUT     (PIOCRA),A
```

5) CONTROL PORT, ALL OUTPUTS

```
        LD      A,11001111B     :MAKE PORT B CONTROL
        OUT     (PIOCRB),A
        SUB     A               :ALL BITS OUTPUTS
        OUT     (PIOCRB),A
```

6) CONTROL PORT, LINES 1,5,6 INPUTS; LINES 0,2,3,4,7 OUTPUTS

```
        LD      A,11001111B     :MAKE PORT A CONTROL
        OUT     (PIOCRA),A
        LD      A,01100010B     :LINES 1,5,6. INPUTS — 0,2,3,4,7 OUTPUTS
        OUT     (PIOCRA),A
```

Z80 INPUT/OUTPUT INSTRUCTIONS

The Z80 microprocessor has an extensive set of Input/Output instructions. All I/O instructions use 8-bit device addresses, thus allowing up to 256 input ports and 256 output ports. But

Z80 I/O INSTRUCTIONS

remember that each PIO occupies four output port addresses and four input port addresses.

The I/O instructions can be grouped as follows:

1) Instructions that use absolute addressing. IN A,(port) and OUT (port),A transfer eight bits of data between the Accumulator and the port addressed by the second byte of the instruction.

2) Single-byte instructions that use register indirect addressing. IN reg,(C) and OUT (C),reg transfer eight bits of data between the specified register and the port addressed by Register C.

3) Block I/O instructions. INI and OUTI transfer eight bits of data between the memory location addressed by Register Pair HL and the port addressed by Register C. Both instructions then increment Register Pair HL and decrement the byte counter in Register B. The Z flag is set if B is decremented to zero and reset otherwise. IND and OUTD are the same instructions except that they decrement Register Pair HL instead of incrementing it.

4) Repeated Block I/O instructions. INIR and OTIR repeat the effects of INI and OUTI, respectively, until B is decremented to zero. INDR and OTDR have the same relationship to IND and OUTD.

You should note the following features of each group of instructions:

1) Instructions with absolute addressing.

I/O INSTRUCTIONS WITH ABSOLUTE ADDRESSING

   • Data is always transferred to or from the Accumulator.

   • No flags are affected.

   • The port address is part of the program memory and cannot be changed if that memory is read-only.

2) Single-byte instructions with register indirect addressing.

I/O INSTRUCTIONS WITH INDIRECT ADDRESSING

   • Data can be transferred to or from any of the primary 8-bit registers (A,B,C,D,E,H,L). However, remember that Register C contains the port address.

   • IN reg,(C) sets the Sign (S), Zero (Z), and Parity (P/O) flags according to the value of the input data. The Carry flag (C) is not modified, but the Half Carry (H) and Negative (N) flags are reset. OUT (C),reg does not affect any flags.

   • The port address is always in Register C. This address is not part of the program memory and could be a parameter for an I/O subroutine (or I/O driver). One I/O driver could thus be used in several different applications or with several similar I/O devices in the same application.

I/O DRIVER

3) Block I/O instructions.

- Data is always transferred to or from the memory location addressed by Register Pair HL.

- The Z (Zero) flag is set if Register B is decremented to zero and cleared otherwise. The S (Sign), P/O (Parity), and H (Half Carry) flags are affected, but their final values are uncertain.

- The port address is always in Register C. Here again, this address could be a parameter for an I/O driver.

- Register B is an 8-bit counter. Thus, the repeated Block I/O instructions can transfer a maximum of 256 bytes. This differs from the Block Move and Block Compare instructions, which use Register Pair BC as a 16-bit counter and can handle up to 65K bytes.

Some examples of the various I/O instructions (without any timing considerations) are:

1) Load the Accumulator from Input Port 2.

   a. Using absolute addressing

```
        IN          A,(2)
```

   b. Using register indirect addressing

```
        LD          C,2
        IN          A,(C)
```

2) Store the contents of the Accumulator in Output Port 5.

   a. Using absolute addressing

```
        OUT         (5),A
```

   b. Using register indirect addressing

```
        LD          C,5
        OUT         (C),A
```

3) Load memory location 0040 from Input Port 2.

   a. Using absolute addressing

```
        IN          A,(2)       ;GET DATA
        LD          (40H),A     ;STORE DATA
```

   b. Using register indirect addressing

```
        LD          C,2         ;GET PORT NUMBER
        IN          A,(C)       ;GET DATA
        LD          (40H),A     ;STORE DATA
```

   c. Using block I/O

```
        LD          C,2         ;GET PORT NUMBER
        LD          HL,40H      ;GET MEMORY DESTINATION
        INI                     ;GET DATA
```

4) Store the contents of memory location 0040 in Output Port 5.

   a. Using absolute addressing

```
            LD      A,(40H)   ;GET DATA
            OUT     (5),A     ;SEND DATA
```

   b. Using register indirect addressing

```
            LD      C,5       ;GET PORT NUMBER
            LD      A,(40H)   ;GET DATA
            OUT     (C),A
```

   c. Using block I/O

```
            LD      C,5       ;GET PORT NUMBER
            LD      HL,40H    ;GET MEMORY SOURCE
            OUTI              ;SEND DATA
```

5) Load memory locations 0040 through 0047 from Input Port 2.

   a. Using absolute addressing

```
            LD      HL,40H    ;GET STARTING ADDRESS OF DATA
            LD      B,8       ;BYTE COUNTER = 8
INBYTE:     IN      A,(2)     ;FETCH DATA BYTE
            LD      (HL),A    ;STORE BYTE IN MEMORY
            INC     HL
            DJNZ    INBYTE
```

   b. Using block I/O

```
            LD      HL,40H    ;GET STARTING ADDRESS OF DATA
            LD      B,8       ;BYTE COUNTER = 8
            LD      C,2       ;GET PORT NUMBER
INBYTE:     INI
            JR      NZ,INBYTE
```

   c. Using repeated block I/O

```
            LD      HL,40H    ;GET STARTING ADDRESS OF DATA
            LD      B,8       ;BYTE COUNTER = 8
            LD      C,2       ;GET PORT NUMBER
            INIR              ;MOVE INPUT BYTES TO MEMORY
```

6) Send the contents of memory locations 0040 through 0047 to Output Port 5.

   a. Using absolute addressing

```
            LD    HL,40H      :GET STARTING ADDRESS OF DATA
            LD    B,8         :GET BYTE COUNTER
OTBYTE:     LD    A,(HL)      :FETCH BYTE FROM MEMORY
            OUT   (5),A       :OUTPUT BYTE
            INC   HL
            DJNZ  OTBYTE
```

   b. Using block I/O

```
            LD    HL,40H      :GET STARTING ADDRESS OF DATA
            LD    B,8         :GET BYTE COUNTER
            LD    C,5         :GET PORT NUMBER
OTBYTE:     OUTI              :OUTPUT BYTE FROM MEMORY
            JR    NZ,OTBYTE
```

   c. Using repeated block I/O

```
            LD    HL,40H      :GET STARTING ADDRESS OF DATA
            LD    B,8         :GET BYTE COUNTER
            LD    C,5         :GET PORT NUMBER
            OTIR              :OUTPUT BYTES FROM MEMORY
```

Note that the repeated Block I/O instructions operate con-
tinuously. You cannot provide any timing between
transfers. Thus, these instructions cannot be used unless

**USING BLOCK
I/O INSTRUCTIONS**

the peripheral operates at the same speed as the processor or timing is handled sepa-
rately in hardware. Ways to handle timing in hardware include forcing the processor
into Wait states or buffering the data. Note that the Block I/O instructions all place the
contents of the byte counter (Register B) on the top half of the Address Bus during the
actual I/O transfer. In output operations, Register B is decremented first. The byte
counter value is then available to external circuitry.

An obvious application for Block I/O instructions is the configuration of PIOs. Several
words must often be sent to a control register to determine operating mode, select pin
directions, and establish the interrupt system. No timing problems occur, since PIOs
operate at the same speed as the CPU. We will discuss the configuration of Z80 PIOs
and serial interfaces (SIOs) with Block I/O instructions later in this chapter and in
Chapter 12.

In subsequent I/O examples, we will use mainly the instructions with absolute address-
ing. You can easily substitute the instructions with register indirect addressing as long
as you remember to initialize Register C. We will occasionally indicate applications for
the Block I/O instructions.
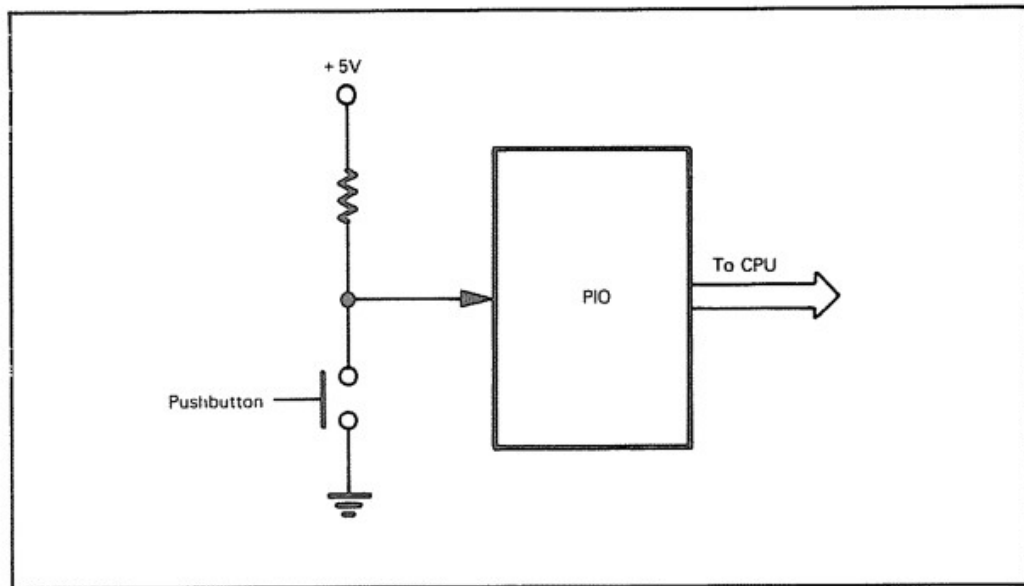
Figure 11-10. A Pushbutton Circuit

EXAMPLES

A Pushbutton Switch

Purpose: To interface a single pushbutton switch (or a single-pole, single-throw (SPST) switch) to a Z80 microprocessor. The pushbutton is a mechanical switch that provides a single contact closure (i.e., a logic zero) while pressed.

Circuit Diagram:

Figure 11-10 shows the circuitry required to interface the pushbutton. It uses one bit of a Z80 PIO that acts as a buffer; no latch is needed, since the pushbutton remains closed for many CPU clock cycles. Pressing the button grounds the PIO input bit. The pullup resistor ensures that the input bit is one if the button is not being pressed.

Programming Examples:

We will perform two tasks with this circuit. They are:

a) Set a memory location based on the state of the button.

b) Count the number of times that the button is pressed.

Task 1: Determine switch closure.

Purpose: Set memory location 0040 to one if the button is not being pressed, and to zero if it is being pressed.

Sample Cases:

1) Button open (i.e., not pressed)

Result = (0040) = 01

2) Button closed (i.e., pressed)

Result = (0040) = 00

11-22

Flowchart:
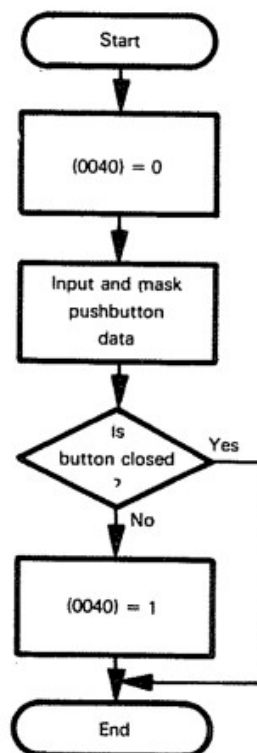
Source Program:

```
        LD      A,01001111B     ;MAKE PORT A INPUT
        OUT     (PIOCRA),A
        LD      HL,40H          ;MARKER = 0
        LD      (HL),0
        IN      A,(PIODRA)      ;READ BUTTON POSITION
        AND     MASK            ;IS BUTTON CLOSED (0)?
        JR      Z,DONE          ;YES, DONE
        INC     (HL)            ;NO, MARKER = 1
DONE:   HALT
```

Object Program:

Memory Location (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3E	LD	A,01001111B
0001	4F		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	21	LD	HL,40H
0005	40		
0006	00		
0007	36	LD	(HL),0
0008	00		
0009	DB	IN	A,(PIODRA)
000A	PIODRA		
000B	E6	AND	MASK
000C	MASK		
000D	28	JR	Z,DONE
000E	01		
000F	34	INC	(HL)
0010	76	HALT	

The port addresses PIOCRA and PIODRA depend on how the PIO is connected in your microcomputer. The PIO control lines are not used in this example. In fact, we could place the A side of the PIO in the control mode with the starting sequence:

```
LD    A,11001111B   ;MAKE PORT A CONTROL
OUT   (PIOCRA),A
LD    A,0FFH        ;ALL BITS INPUTS
OUT   (PIOCRA),A
```

MASK depends on the bit to which the pushbutton is connected; it has a one in the button position and zeros elsewhere.

Button Position (Bit Number)	Mask	
	Binary	Hex
0	00000001	01
1	00000010	02
2	00000100	04
3	00001000	08
4	00010000	10
5	00100000	20
6	01000000	40
7	10000000	80

If the button is attached to bit 0 or bit 7 of the input port, the program can use a Shift instruction to set the Carry and thereby determine the button's state. For example,

```
Bit 7
IN    A,(PIODRA)   ;READ BUTTON POSITION
RLA                ;IS BUTTON CLOSED (ZERO)?
JR    NC,DONE      ;YES, DONE

Bit 0
IN    A,(PIODRA)   ;READ BUTTON POSITION
RRA                ;IS BUTTON CLOSED (ZERO)?
JR    NC,DONE      ;YES, DONE
```

The procedure for bit 7 is even simpler if we have the address of the PIO data register in Register C. This is because the input instructions using register indirect addressing (e.g., IN A,(C)) affect the Sign flag. The required sequence is:

Bit 7 (PIODRA in Register C)

```
IN      A,(C)        ;READ BUTTON POSITION
JP      P,DONE       ;DONE IF BUTTON CLOSED (ZERO)
```

If the button is attached to bits 6 or 7 of the input port, the program can use the Sign bit to determine the button's state. For example.

Bit 7

```
IN      A,(PIODRA)   ;READ BUTTON POSITION
AND     A            ;IS BUTTON CLOSED (ZERO)?
JP      P,DONE       ;YES, DONE
```

IN A,(port) does not affect the flags; therefore, we must use the AND A instruction to set the flags without changing the Accumulator.

Bit 6

```
IN      A,(PIODRA)   ;READ BUTTON POSITION
ADD     A,A          ;IS BUTTON CLOSED (ZERO)?
JP      P,DONE       ;YES, DONE
```

RLA cannot be used because it does not affect the Sign bit.

Task 2: Count switch closures.

Purpose: Count the number of button closures by incrementing memory location 0040 after each closure.

Sample Case:

Pressing the button ten times after the start of the program should give

(0040) = 0A

Note: In order to count the number of times that the button has been pressed, we must be sure that each closure causes a single transition. However, a mechanical pushbutton does not produce a single transition for each closure, because the mechanical contacts bounce back and forth before settling into their final positions. We can use a one-shot to eliminate the bounce or we can handle it in software.

```
SWITCH
BOUNCE
```

The program can debounce the pushbutton by waiting after it finds a closure. The required delay is called the debouncing time and is part of the specifications of the pushbutton. It is typically a few milliseconds long. The program should not examine the pushbutton during this period because it might mistake the bounces for new closures. The program may either enter a delay routine like the one described previously or may simply perform other tasks for the specified amount of time.

```
DEBOUNCING
IN SOFTWARE
```

Even after debouncing, the program must still wait for the present closure to end before looking for a new closure. This procedure avoids double counting. The following program uses a software delay of 1 ms to debounce the pushbutton. You may want to try varying the delay or eliminating it entirely to see what happens. To run this program, you must also enter the delay subroutine into memory starting at location 0030.

Flowchart:
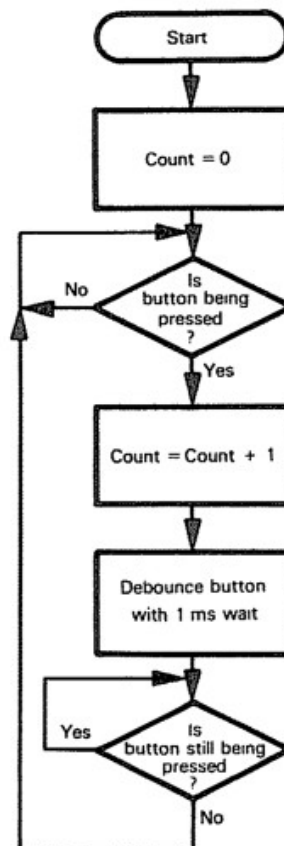
Source Program:

```
          LD      A.01001111B    :MAKE PORT A INPUT
          OUT     (PIOCRA),A
          LD      HL,40H
          LD      (HL),0         :CLOSURE COUNT = ZERO
CHKCL:    IN      A,(PIODRA)     :READ BUTTON POSITION
          AND     MASK           :IS BUTTON BEING PRESSED (0)?
          JR      NZ,CHKCL       :NO, WAIT UNTIL IT IS
          INC     (HL)           :YES. INCREMENT CLOSURE COUNT
          CALL    DELAY          :WAIT 1 MS TO DEBOUNCE
CHKOP:    IN      A,(PIODRA)     :READ BUTTON POSITION
          AND     MASK           :IS BUTTON STILL BEING PRESSED (0)?
          JR      Z,CHKOP        :YES, WAIT FOR RELEASE
          JR      CHKCL          ;NO, LOOK FOR NEXT CLOSURE
```

Object Program:

Memory Location (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	3E		LD	A.01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	21		LD	HL,40H
0005	40			
0006	00			
0007	36		LD	(HL),0
0008	00			
0009	DB	CHKCL:	IN	A,(PIODRA)
000A	PIODRA			
000B	E6		AND	MASK
000C	MASK			
000D	20		JR	NZ,CHKCL
000E	FA			
000F	34		INC	(HL)
0010	CD		CALL	DELAY
0011	30			
0012	00			
0013	DB	CHKOP:	IN	A,(PIODRA)
0014	PIODRA			
0015	E6		AND	MASK
0016	MASK			
0017	28		JR	Z,CHKOP
0018	FA			
0019	18		JR	CHKCL
001A	EE			

The three instructions beginning with the label CHKOP are used to determine when the switch reopens.

Clearly we do not really need a PIO for this simple interface. An addressable tri-state buffer would do the job at far lower cost.

A Toggle Switch

Purpose: To interface a single-pole, double-throw (SPDT) toggle switch to a Z80 microprocessor. The toggle is a mechanical device that is either in the normally closed (NC) position or the normally open (NO) position.

Circuit Diagram:

Figure 11-11 shows the circuitry required to interface the switch. Like the pushbutton, the switch uses one bit of a Z80 PIO that serves as an addressable buffer. Unlike the button, the switch may be left in either position. Typical program tasks are to determine the switch position and to see if the position has

> **DEBOUNCING WITH CROSS-COUPLED NAND GATES**

changed. Either a one-shot with a pulse length of a few milliseconds or a pair of cross-coupled NAND gates (see Figure 11-12) can debounce a mechanical switch.

The circuits will produce a single step or pulse in response to a change in switch position even if the switch bounces before settling into its new position.

Programming Examples:

We will perform two tasks involving this circuit. They are:

1) Set a memory location to one when the switch is closed.
2) Set a memory location to one when the state of the switch changes.

Task 1: Wait for switch to close.

Purpose: Memory location 0040 is zero until the switch is closed and then is set to one; that is, the processor clears memory location 0040, waits for the switch to be closed, and then sets memory location 0040 to one.

The switch could be marked Run/Halt, since the processor will not proceed until the switch is closed.

Flowchart:
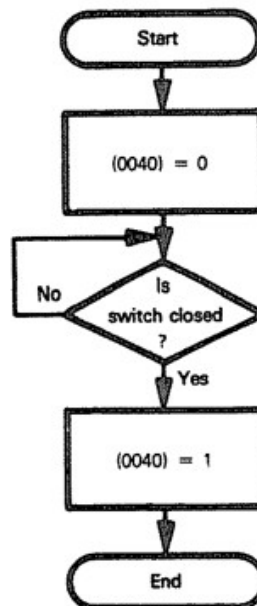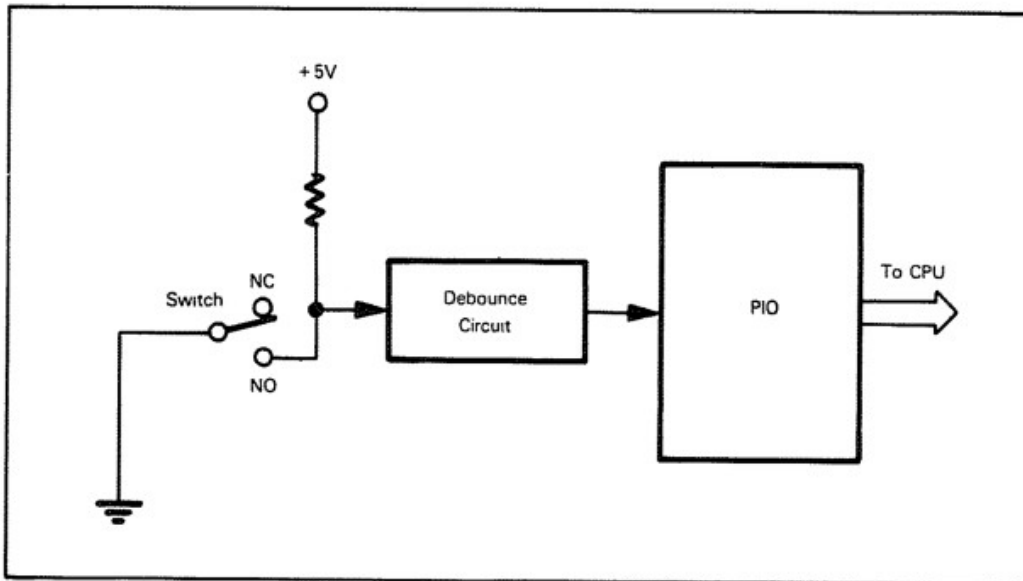
Figure 11-11. A Toggle Switch Circuit
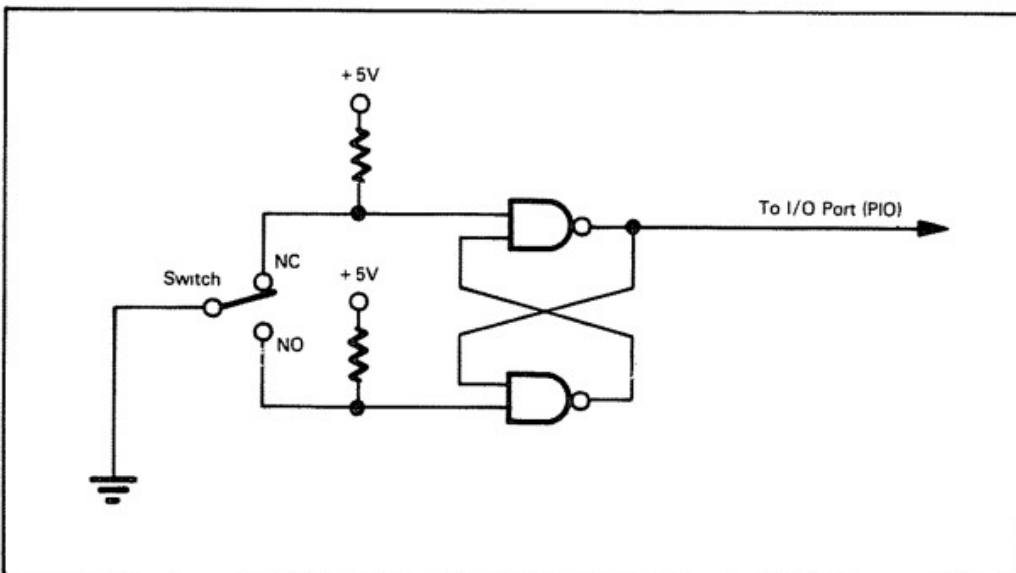
Figure 11-12. A Debounce Circuit Based on Cross-coupled NAND Gates

Source Program:

```
        LD      A,01001111B   ;MAKE PORT A INPUT
        OUT     (PIOCRA),A
        LD      HL,40H
        LD      (HL),0        ;MARKER = ZERO
WAITC:  IN      A,(PIODRA)    ;READ SWITCH POSITION
        AND     MASK          ;IS SWITCH CLOSED (ZERO)?
        JR      NZ,WAITC      ;NO, WAIT FOR SWITCH TO CLOSE
        INC     (HL)          ;YES, MARKER = 1
        HALT
```

Object Program:

Memory Location (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3E	LD	A,01001111B
0001	4F		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	21	LD	HL,40H
0005	40		
0006	00		
0007	36	LD	(HL),0
0008	00		
0009	DB	WAITC: IN	A,(PIODRA)
000A	PIODRA		
000B	E6	AND	MASK
000C	MASK		
000D	20	JR	NZ,WAITC
000E	FA		
000F	34	INC	(HL)
0010	76	HALT	

Task 2: Wait for switch to change.

Purpose: Memory location 0040 remains zero until the switch position changes; i.e., the processor waits until the switch changes, then sets memory location 0040 to 1.

Flowchart:
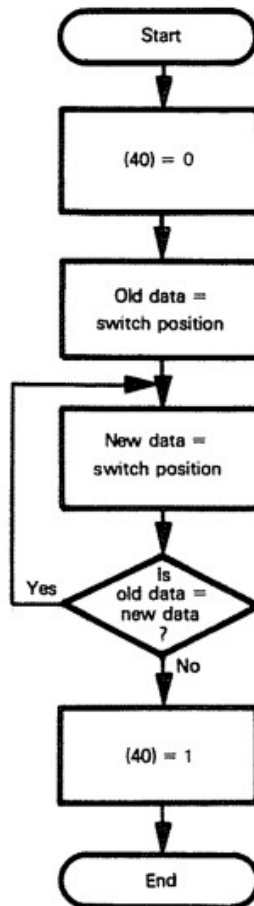
Source Program:

```
          LD      A,01001111B    ;MAKE PORT A INPUT
          OUT     (PIOCRA),A
          LD      HL,40H
          LD      (HL),0         ;MARKER = ZERO
          IN      A,(PIODRA)     ;GET OLD SWITCH POSITION
          AND     MASK
          LD      B,A
SRCH:     IN      A,(PIODRA)     ;GET NEW SWITCH POSITION
          AND     MASK
          CP      B              ;ARE NEW AND OLD POSITIONS THE SAME?
          JR      Z,SRCH         ;YES, WAIT
          INC     (HL)           ;NO, MARKER = ONE
          HALT
```

Object Program:

Memory Location (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	21		LD	HL,40H
0005	40			
0006	00			
0007	DB		IN	A,(PIODRA)
0008	PIODRA			
0009	E6		AND	MASK
000A	MASK			
000B	47		LD	B,A
000C	DB	SRCH:	IN	A,(PIODRA)
000D	PIODRA			
000E	E6		AND	MASK
000F	MASK			
0010	B8		CP	B
0011	28		JR	Z,SRCH
0012	F9			
0013	34		INC	(HL)
0014	76		HALT	

A Subtract or Exclusive OR could replace the Compare in the program. Either of these instructions would, however, change the contents of the Accumulator. The Exclusive OR would be useful if several switches were attached to the same PIO, since it would produce a one bit for each switch that changed state. How would you rewrite this program so as to debounce the switch in software?
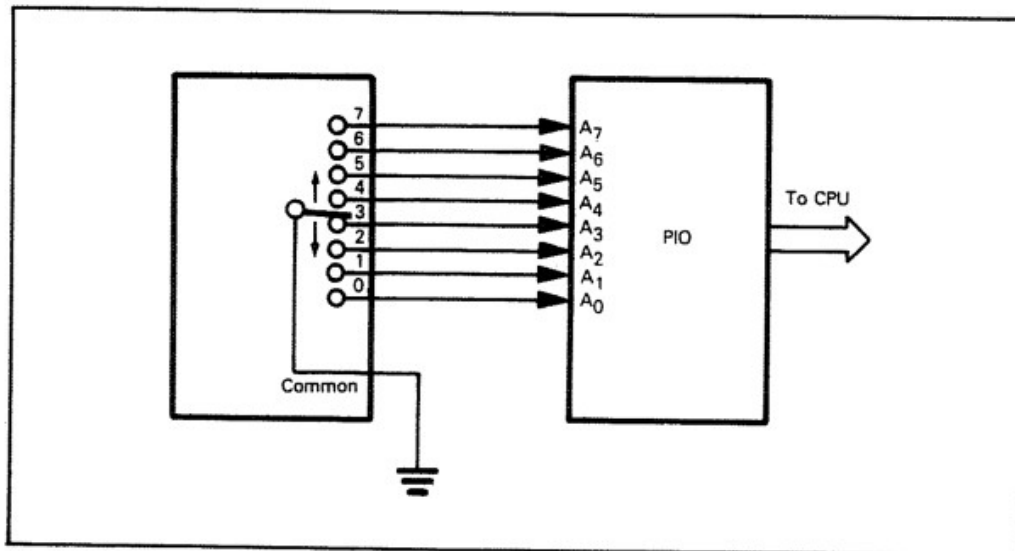
Figure 11-13. A Multiple-Position Switch

A Multiple-Position (Rotary, Selector, or Thumbwheel) Switch

Purpose: To interface a multiple-position switch to a microprocessor. The lead corresponding to the switch position is grounded, while the other leads are high (logic ones).

Circuit Diagram:

Figure 11-13 shows the circuitry required to interface an 8-position switch. The switch uses all eight data bits of one side of a PIO. Typical tasks are to determine the position of the switch and to check whether or not that position has changed. Two special situations must be handled:

1) The switch is temporarily between positions so that no leads are grounded.

2) The switch has not yet reached its final position.

The first of these situations can be handled by waiting until the input is not all ones, i.e., until a switch lead is grounded. We can handle the second situation by examining the switch again after a delay (such as 1 or 2 seconds) and only accepting the input when it remains the same. This delay will not affect the responsiveness of the system to the switch. We can also use another switch (i.e., a Load switch) to tell the processor when the selector switch should be read.

Programming Examples:

We will perform two tasks involving the circuit of Figure 11-13. These are:

a) Monitor the switch until it is in a definite position, then determine the position and store its binary value in a memory location.

b) Wait for the position of the switch to change, then store the new position in a memory location.

If the switch is in a position, the lead from that position is grounded through the common line. Pullup resistors on the input lines avoid problems caused by noise.

Table 11-3. Data Input vs. Switch Position

Switch Position	Data Input	
	Binary	Hex
0	11111110	FE
1	11111101	FD
2	11111011	FB
3	11110111	F7
4	11101111	EF
5	11011111	DF
6	10111111	BF
7	01111111	7F

Task 1: Determine switch position.

Purpose: The program waits for the switch to be in a specific position and then places the number of that position into memory location 0040.

Table 11-3 contains the data inputs corresponding to the various switch positions.

This scheme is inefficient, since it requires eight bits to distinguish among eight different positions.

A TTL or MOS encoder could reduce the number of bits needed. Figure 11-14 shows a circuit using the 74LS148 TTL 8-to-3 encoder.[4] We attach the switch outputs in inverse order, since the 74LS148 device has active-low inputs and outputs. The output of

USING A TTL ENCODER

the encoder circuit is a 3-bit representation of the switch position. Many switches include encoders so that their outputs are coded, usually as a BCD digit (in negative logic).

The encoder produces active-low outputs, so, for example, switch position 5, which is attached to input 2, produces an output of 2 in negative logic (or 5 in positive logic). You may want to verify the double negative for yourself.
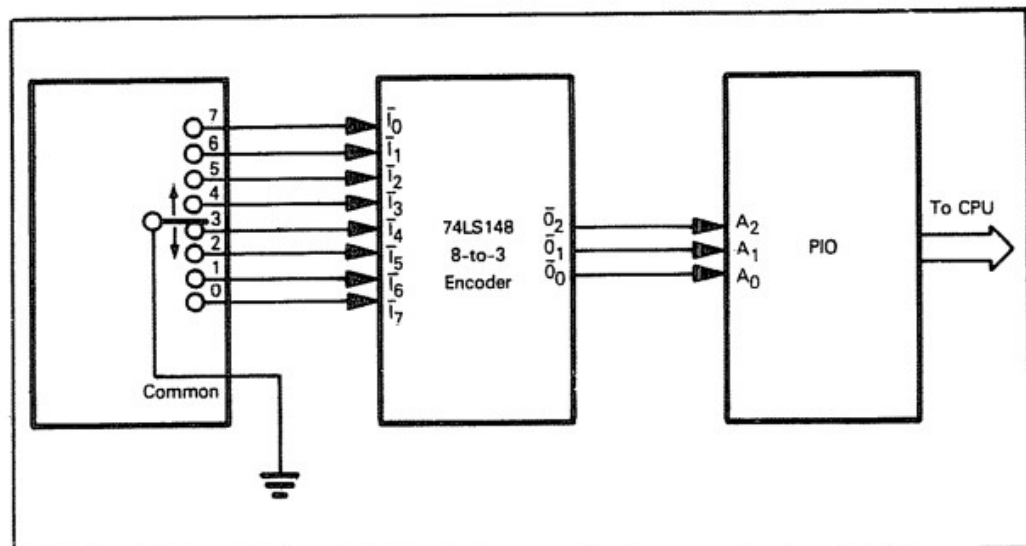
Figure 11-14. A Multiple-Position Switch with an Encoder

11-34

Flowchart:
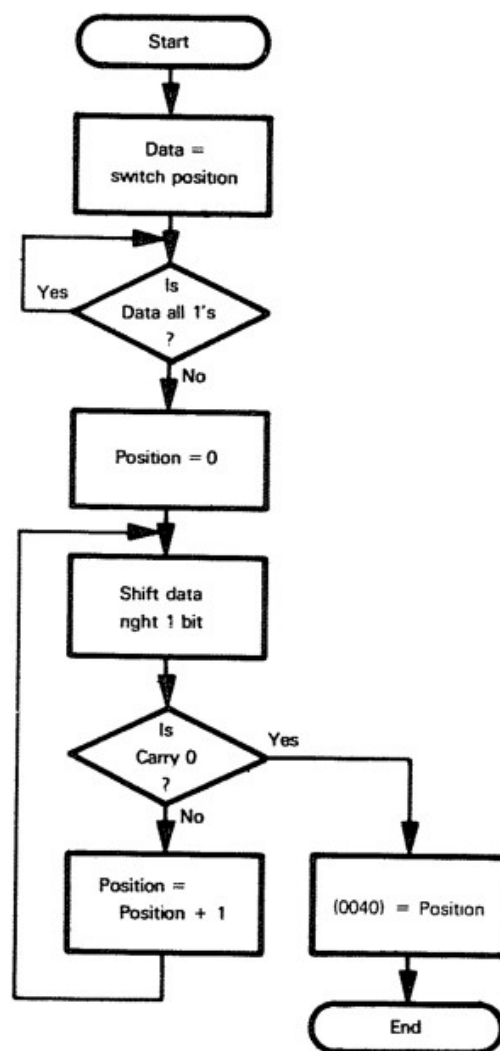
Source Program:

```
        LD      A,01001111B     ;MAKE PORT A INPUT
        OUT     (PIOCRA),A
CHKSW:  IN      A,(PIODRA)      ;GET SWITCH DATA
        CP      0FFH            ;IS SWITCH IN A POSITION?
        JR      Z,CHKSW         ;NO, WAIT FOR A POSITION
        LD      B,0             ;SWITCH POSITION = ZERO
CHPOS:  RRA                     ;IS NEXT BIT GROUNDED POSITION?
        JR      NC,DONE         ;YES, SWITCH POSITION FOUND
        INC     B               ;NO, INCREMENT SWITCH POSITION
        JR      CHPOS
DONE:   LD      HL,40H          ;STORE SWITCH POSITION
        LD      (HL),B
        HALT
```

Object Program:

Memory Location (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	DB	CHKSW:	IN	A,(PIODRA)
0005	PIODRA			
0006	FE		CP	0FFH
0007	FF			
0008	28		JR	Z,CHKSW
0009	FA			
000A	06		LD	B,0
000B	00			
000C	1F	CHPOS:	RRA	
000D	30		JR	NC,DONE
000E	03			
000F	04		INC	B
0010	18		JR	CHPOS
0011	FA			
0012	21	DONE:	LD	HL,40H
0013	40			
0014	00			
0015	70		LD	(HL),B
0016	76		HALT	

Suppose that a faulty switch or defective PIO results in the input always being $0FF_{16}$. How could you change the program so that it would detect this error?

There is an unconditional jump, JR CHPOS, in the source program. Can you change the initial conditions so as to make this instruction unnecessary?

This example assumes that the switch is debounced in hardware. How would you change the program to debounce the switch in software?

Task 2: Wait for switch position to change.

Purpose: The program waits for the switch position to change and places the new position (decoded) into memory location 0040. The program waits until the switch reaches its new position.

Flowchart:
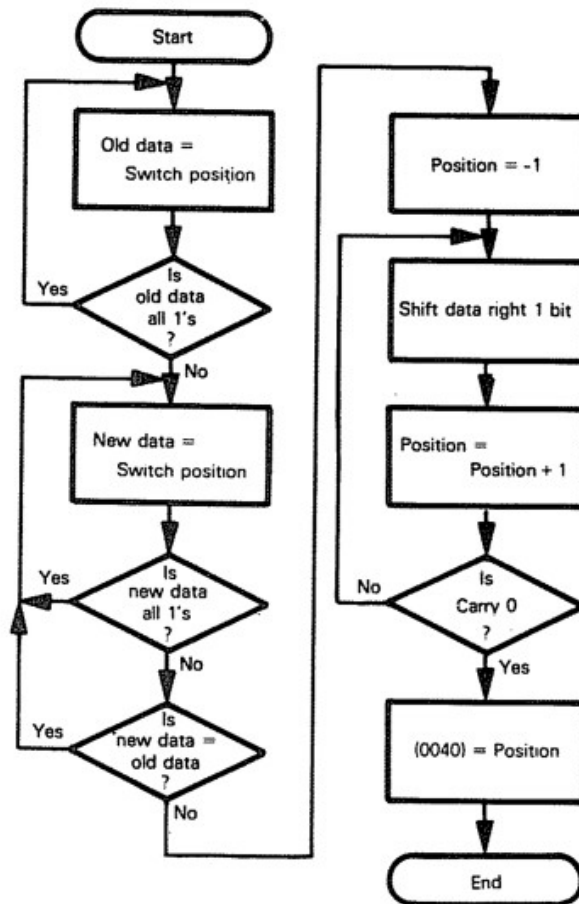
Source Program:

```
        LD      A,01001111B     ;MAKE PORT A INPUT
        OUT     (PIOCRA),A
CHFST:  IN      A,(PIODRA)      ;GET SWITCH DATA
        CP      0FFH            ;IS SWITCH IN A POSITION?
        JR      Z,CHFST         ;NO, WAIT UNTIL IT IS
        LD      B,A
CHSEC:  IN      A,(PIODRA)      ;GET NEW SWITCH DATA
        CP      0FFH            ;IS SWITCH IN A POSITION?
        JR      Z,CHSEC         ;NO, WAIT UNTIL IT IS
        CP      B               ;IS POSITION SAME AS BEFORE?
        JR      Z,CHSEC         ;YES, WAIT FOR IT TO CHANGE
        LD      B,0FFH          ;SWITCH POSITION = -1
CHPOS:  INC     B               ;INCREMENT SWITCH POSITION
        RRA                     ;IS NEXT BIT GROUNDED POSITION?
        JR      C,CHPOS         ;NO, KEEP LOOKING FOR GROUNDED POSITION
        LD      HL,40H          ;STORE SWITCH POSITION
        LD      (HL),B
        HALT
```

Object Program:

Memory Location (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	DB	CHFST:	IN	A,(PIODRA)
0005	PIODRA			
0006	FE		CP	0FFH
0007	FF			
0008	28		JR	Z,CHFST
0009	FA			
000A	47		LD	B,A
000B	DB	CHSEC:	IN	A,(PIODRA)
000C	PIODRA			
000D	FE		CP	0FFH
000E	FF			
000F	28		JR	Z,CHSEC
0010	FA			
0011	B8		CP	B
0012	28		JR	Z,CHSEC
0013	F7			
0014	06		LD	B,0FFH
0015	FF			
0016	04	CHPOS:	INC	B
0017	1F		RRA	
0018	38		JR	C,CHPOS
0019	FC			
001A	21		LD	HL,40H
001B	40			
001C	00			
001D	70		LD	(HL),B
001E	76		HALT	

An alternative method for determining if the switch is in a position is:

```
CHKSW:  IN    A,(PIODRA)
        INC   A
        JR    Z,CHKSW
```

Why does this work? What happens to the input data?

A Single LED

Purpose: To interface a single light-emitting diode to a Z80 microprocessor. The LED can be attached so that either a logic zero or a logic one turns it on.

Circuit Diagram:

Figure 11-15 shows the circuitry required to interface an LED. The LED lights when its anode is positive with respect to its cathode (Figure 11-15a). Therefore, you can either light the LED by grounding the cathode and having the computer supply a one to the anode (Figure 11-15b) or by connecting the anode to +5 volts and having the computer supply a zero to the cathode (Figure 11-15c). Using the cathode is the most common approach. The LED is brightest when it operates from pulsed currents of about 10 or 50 mA applied a few hundred times per second. LEDs have a very short turn-on time (in the microsecond range) so they are well suited to multiplexing (operating several from a single port). LED circuits usually need peripheral or transistor drivers and current-limiting resistors. MOS devices normally cannot drive LEDs directly and make them bright enough for easy viewing.

`LED CONTROL`

Note: The PIO has an output latch on each port. However, the B port is normally used for output, since it has somewhat more drive capability. In particular, the B port outputs are capable of driving Darlington transistors (providing 1.5 mA minimum at 1.5 V). Darlington transistors are high-gain transistors capable of switching large amounts of current at high speed; they are useful in driving solenoids, relays, and other devices.

Task: Turn the light on or off.

Purpose: The program turns a single LED either on or off.

A. Send a Logic One to the LED (turn a positive display on or a negative display off).

Source Program:

(form data initially)

```
        LD      A,00001111B    :MAKE PORT B OUTPUT
        OUT     (PIOCRB),A
        LD      A,MASKP        :GET DATA FOR LED
        OUT     (PIODRB),A     :SEND DATA TO LED
        HALT
```

An alternative using the control mode is:

```
        LD      A,11001111B    :MAKE PORT B CONTROL
        OUT     (PIOCRB),A
        SUB     A              :MAKE ALL B LINES OUTPUTS
        OUT     (PIOCRB),A
        LD      A,MASKP        :GET DATA FOR LED
        OUT     (PIODRB),A     :SEND DATA TO LED
        HALT
```

(update data)

```
        IN      A,(PIODRB)     :GET OLD DATA
        SET     LED,A          :TURN ON LED BIT
        OUT     (PIODRB),A     :SEND DATA TO LED
        HALT
```

MASKP has a one bit in the LED position and zeros elsewhere. Note that we can read the PIO Data Output register when the PIO is in the output mode. We can also read any combination of input data and output register data when the PIO is in the control mode; the combination is defined by the assignment of inputs and outputs.

a) Basic LED circuitry. The resistor R should limit the maximum current to 50 mA and the average current to 10 mA.

b) Interfacing an LED with positive logic. A logic '1' from the CPU turns the LED on.

c) Interfacing an LED with negative logic. A logic '0' from the CPU turns the LED on. The driver or the CPU may invert the logic levels.
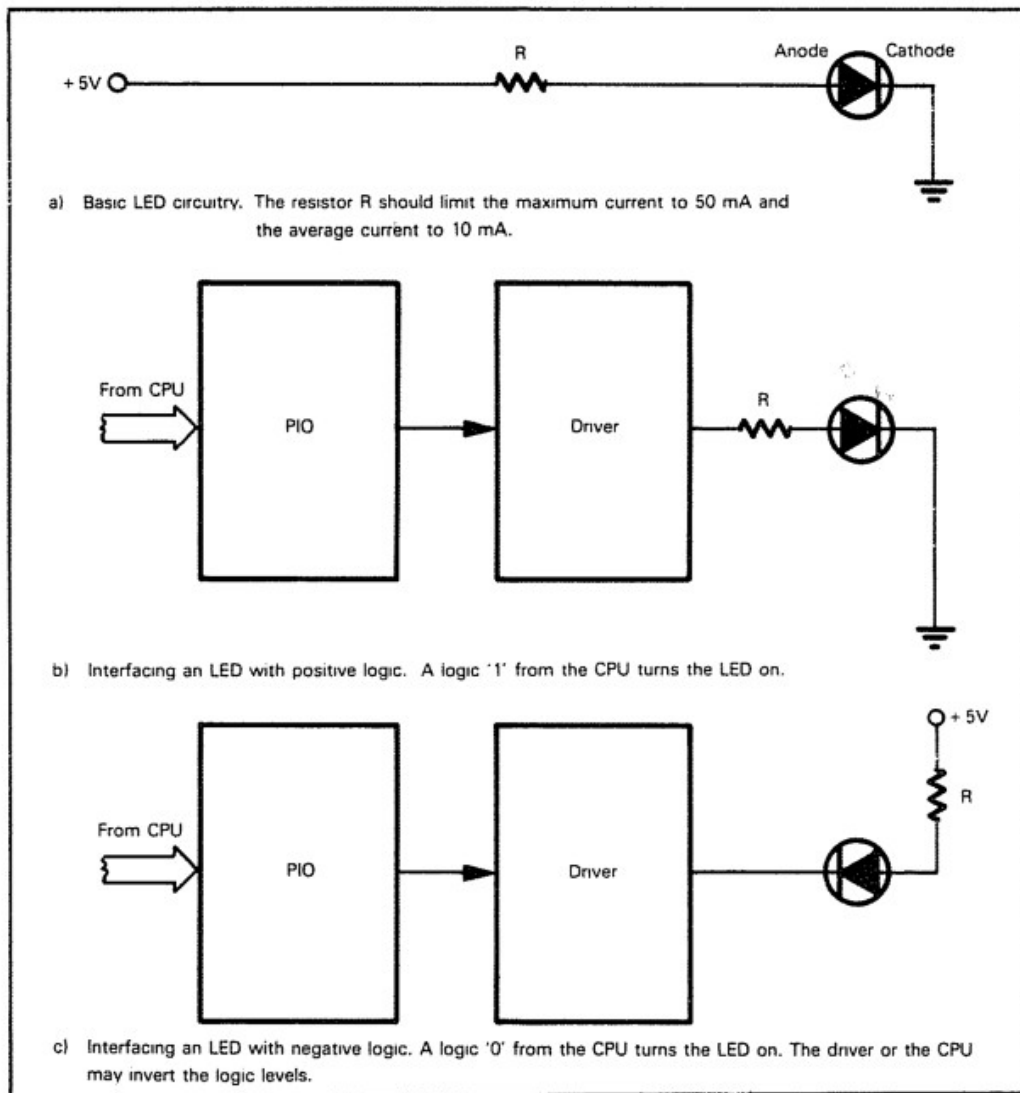
Figure 11-15. Interfacing an LED

Object Program:

Memory Location (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
	(form data initially)		
0000	3E	LD	A,00001111B
0001	0F		
0002	D3	OUT	(PIOCRB),A
0003	PIOCRB		
0004	3E	LD	A,MASKP
0005	MASKP		
0006	D3	OUT	(PIODRB),A
0007	PIODRB		
0008	76	HALT	
	(update data)		
0009	DB	IN	A,(PIODRB)
000A	PIODRB		
000B	CB	SET	LED,A
000C	LED		
000D	D3	OUT	(PIODRB),A
000E	PIODRB		
000F	76	HALT	

B.   Send a Logic Zero to the LED (turn a positive display off or a negative display on).

The differences are that MASKP must be replaced by its logical complement MASKN and SET LED,A must be replaced by RES LED,A. Note that the second byte of the object code for SET LED,A and RES LED,A depends on the actual bit position to which the name LED refers.

MASKN has a zero bit in the LED position and ones elsewhere.
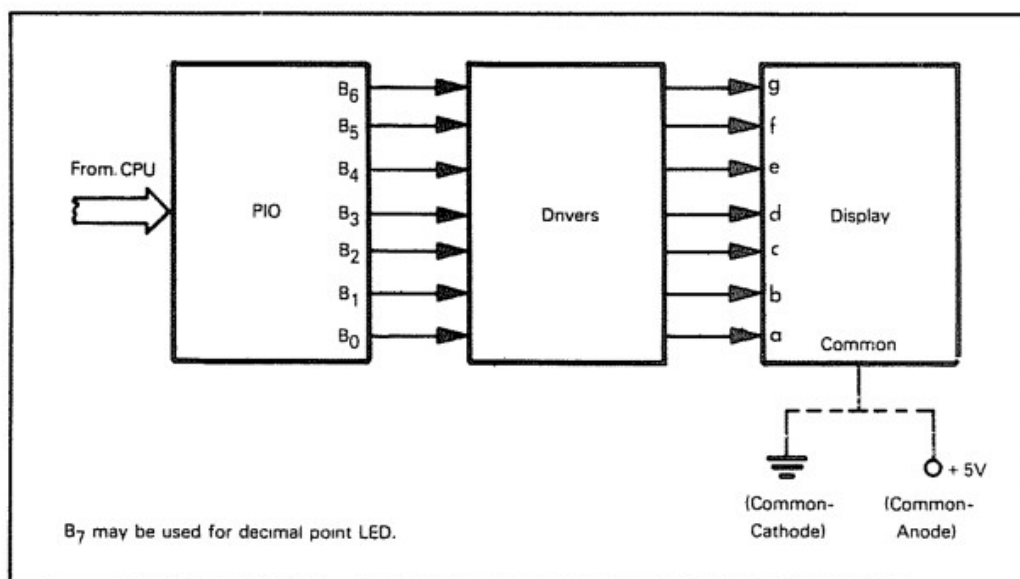
Figure 11-16. Interfacing a Seven-Segment Display

Seven-Segment LED Display

Purpose: To interface a seven-segment LED display to a Z80 microprocessor. The display may be either common-anode (negative logic) or common-cathode (positive logic).

Circuit Diagram:

Figure 11-16 shows the circuitry required to interface a seven-segment display. Each segment may have one, two, or more LEDs attached in the same way. There are two ways of connecting the displays. One is tying all the cathodes together to ground (see Figure 11-17a); this is a

> COMMON-ANODE
> OR
> COMMON-CATHODE
> DISPLAYS

"common-cathode" display. Tying all the anodes together to a positive voltage supply (see Figure 11-17b) is a "common-anode" display, and a logic zero at a cathode lights a segment. So the common-cathode display uses positive logic and the common-anode display negative logic. Either display requires appropriate drivers and resistors.

The Common line from the display is tied either to ground or to +5 volts. The display segments are customarily labelled:

11-43
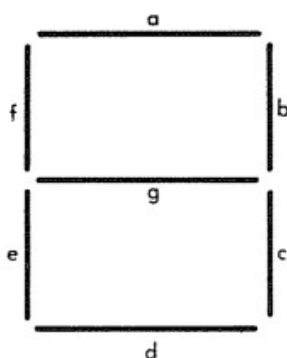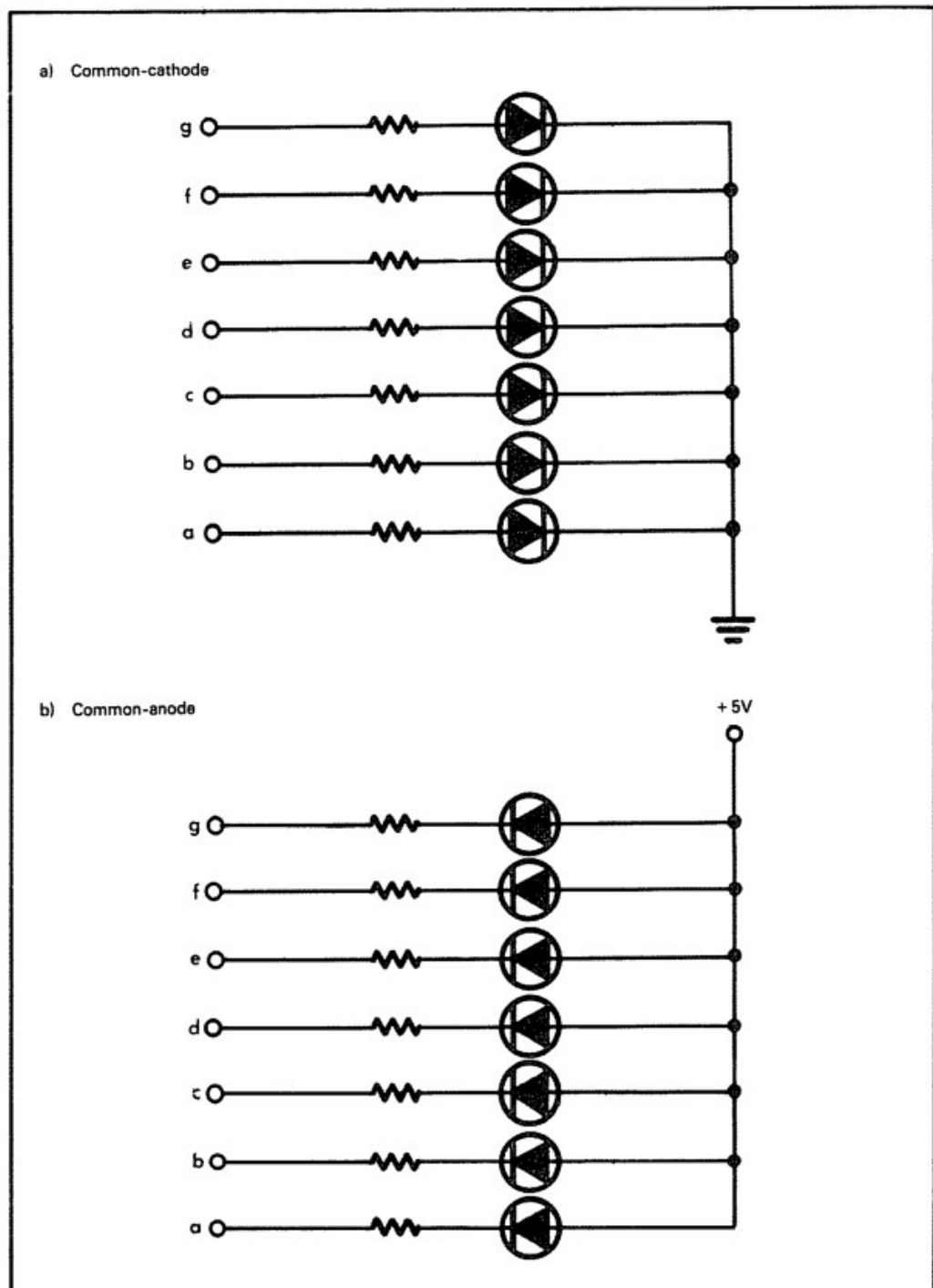
Figure 11-17. Seven-Segment Display Organization

Table 11-4. Seven-Segment Representations of Decimal Numbers

Number	Hexadecimal Representation	
	Common-cathode	Common-anode
0	3F	40
1	06	79
2	5B	24
3	4F	30
4	66	19
5	6D	12
6	7D	02
7	07	78
8	7F	00
9	67	18

Bit 7 is always zero and the others are g, f, e, d, c, b, and a in decreasing order of significance.

SEVEN-SEGMENT REPRESENTATIONS

Note: The seven-segment display is widely used because it contains the smallest number of separately controlled segments that can provide recognizable representations of all the decimal digits (see Figure 11-18 and Table 11-4). Seven-segment displays can also produce some letters and other characters (see Table 11-5). Better representations require a substantially larger number of segments and more circuitry.[5] Since seven-segment displays are so popular, low-cost seven-segment decoder/drivers have become widely available. The most popular devices are the 7447 common-anode driver and the 7448 common-cathode driver[6]: these devices have Lamp Test inputs (that turn all the segments on) and blanking inputs and outputs (for blanking leading or trailing zeros).
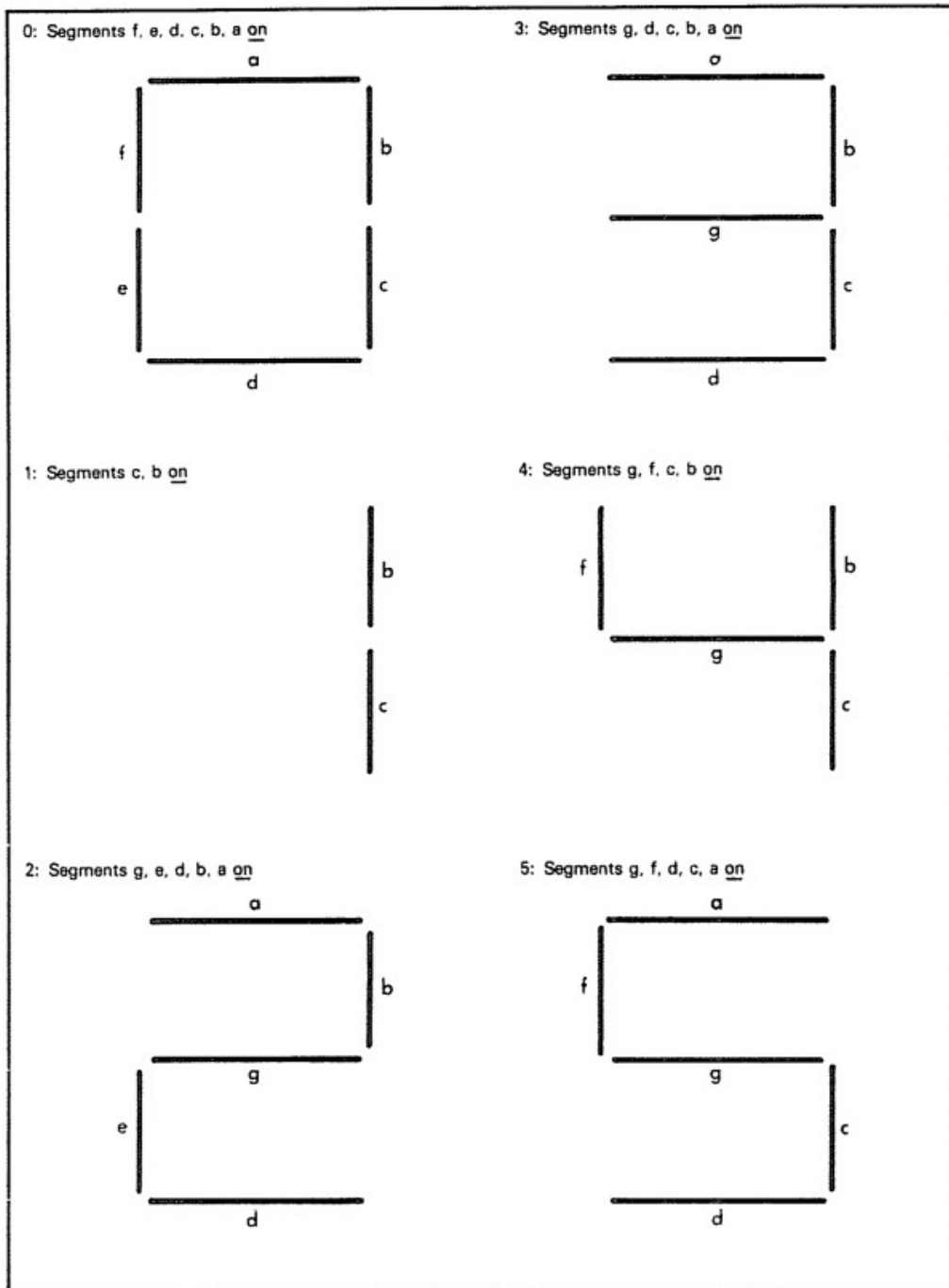
Figure 11-18. Seven-Segment Representations of Decimal Digits

6: Segments g, f, e, d, c, a on

Note that the alternate representation with a off may be reserved for the lower case letter 'b'.

8: Segments g, f, e, d, c, b, a on

This is the same as LAMP TEST.

7: Segments c, b, a on

9: Segments g, f, c, b, a on

An alternate has segment d on also.
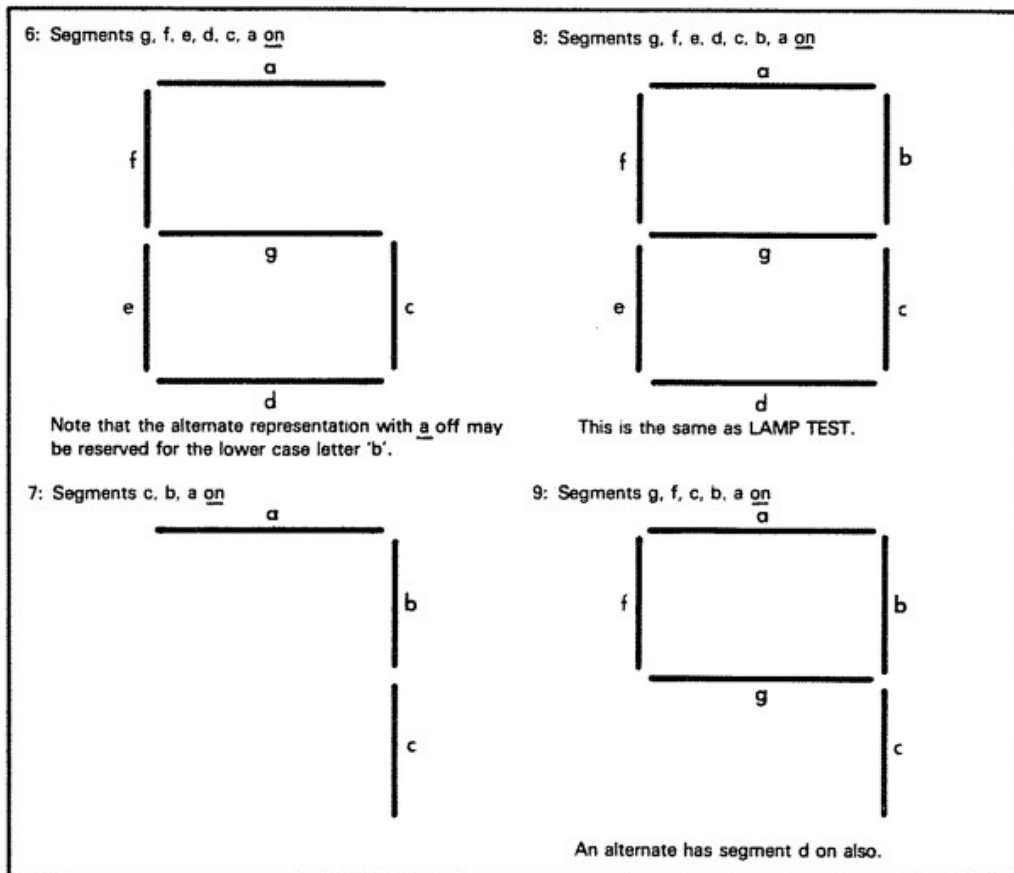
Figure 11-18. Seven-Segment Representations of Decimal Digits
(Continued)

Table 11-5. Seven-Segment Representations of Letters and Symbols

Upper-case Letters

Letter	Hexadecimal Representation	
	Common-cathode	Common-anode
A	77	08
C	39	46
E	79	06
F	71	0E
H	76	09
I	06	79
J	1E	61
L	38	47
O	3F	40
P	73	0C
U	3E	41
Y	66	19

Lower-case Letters and Special Characters

Character	Hexadecimal Representation	
	Common-cathode	Common-anode
b	7C	03
c	58	27
d	5E	21
h	74	0B
n	54	2B
o	5C	23
r	50	2F
u	1C	63
-	40	3F
?	53	2C

11-48

Task 1: Display a decimal digit.

Purpose: Display the contents of memory location 0040 on a seven-segment display if it contains a decimal digit. Otherwise, blank the display.

Sample Problems:

a.　(0040) = 05

　　Result is 5 on display

b.　(0040) = 66

　　Result is a blank display

Flowchart:

```
               ┌──────────────┐
              (    Start       )
               └──────────────┘
                      │
                      ▼
          ┌──────────────────────┐
          │  Code  =   Blank      │
          │  Data  =   (0040)     │
          └──────────────────────┘
                      │
                      ▼
                   ╱─────╲
          Yes     ╱   Is   ╲
        ◄────────◄  Data ≥ 10 ►
                  ╲    ?    ╱
                   ╲─────╱
                      │ No
                      ▼
          ┌──────────────────────┐
          │  Code = (SSEG +       │
          │         Data)         │
          └──────────────────────┘
                      │
                      ▼
          ┌──────────────────────┐
          │   Send code           │
          │   to display          │
          └──────────────────────┘
                      │
                      ▼
               ┌──────────────┐
              (    End         )
               └──────────────┘
```
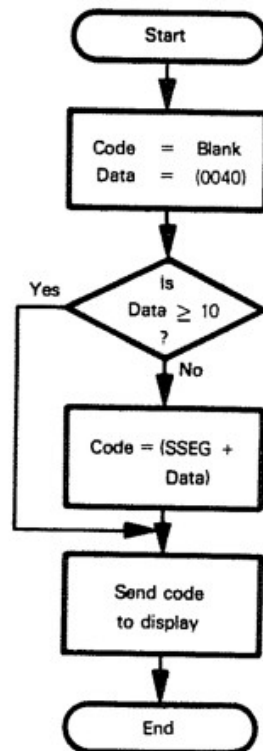
Source Program:

```
        LD      A,00001111B  ;MAKE PORT B OUTPUT
        OUT     (PIOCRB),A
        LD      B,BLANK      ;GET BLANK CODE
        LD      A,(40H)      ;GET DATA
        CP      10           ;IS DATA A DECIMAL DIGIT?
        JR      NC,DSPLY     ;NO, DISPLAY BLANKS
        LD      DE,SSEG      ;GET BASE ADDRESS OF SEVEN-SEGMENT
                             ;  TABLE
        LD      H,0          ;MAKE DATA INTO 16-BIT INDEX
        LD      L,A
        ADD     HL,DE        ;ACCESS ELEMENT IN TABLE
        LD      B,(HL)       ;GET SEVEN-SEGMENT CODE
DSPLY:  LD      A,B
        OUT     (PIODRB),A   ;SEND CODE TO DISPLAY
        HALT
```

BLANK is 00 for a common-cathode display, FF for a common-anode display. An alternative procedure would be to put the blank code at the end of the table and replace all improper data values with 10, i.e.,

```
        LD      A,(40H)      ;GET DATA
        CP      10           ;IS DATA A DECIMAL DIGIT?
        JR      C,CNVRT      ;YES, CONVERT DIRECTLY TO SEVEN-SEGMENT
        LD      A,10         ;NO, GET INDEX FOR BLANK CODE
CNVRT:  LD      DE,SSEG      ;GET BASE ADDRESS OF SEVEN-SEGMENT TABLE
```

Table SSEG is either the common-cathode or common-anode representation of the decimal digits from Table 11-4.

Object Program:

Memory Location (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	3E		LD	A,00001111B
0001	0F			
0002	D3		OUT	(PIOCRB),A
0003	PIOCRB			
0004	06		LD	B,BLANK
0005	BLANK			
0006	3A		LD	A,(40H)
0007	40			
0008	00			
0009	FE		CP	10
000A	0A			
000B	30		JR	NC,DSPLY
000C	08			
000D	11		LD	DE,SSEG
000E	20			
000F	00			
0010	26		LD	H,0
0011	00			
0012	6F		LD	L,A
0013	19		ADD	HL,DE
0014	46		LD	B,(HL)
0015	78	DSPLY:	LD	A,B
0016	D3		OUT	(PIODRB),A
0017	PIODRB			
0018	76		HALT	
0020-0029		SSEG:	(seven-segment code table)	

Several displays may be multiplexed, as shown in Figure 11-19. A brief strobe on the B RDY line clocks the counter and directs data to the next display. Note that B RDY is tied directly back to B STB, i.e., the ready line essentially provides its own acknowledgment. The timing of the PIO is such that this connection results in a strobe with a duration of one clock period. Such a brief strobe is exactly what the counter requires. RESET starts the decimal counter at nine so that the first output operation clears the counter and directs data to the first display.

The following program uses the delay routine to pulse each of ten common-cathode displays for 1 ms.

D. C. B. and A (D most significant, A least significant) are the 4-bit output from the counter. These 4 bits activate the correspondingly numbered output from the decoder, and hence the correspondingly numbered display.
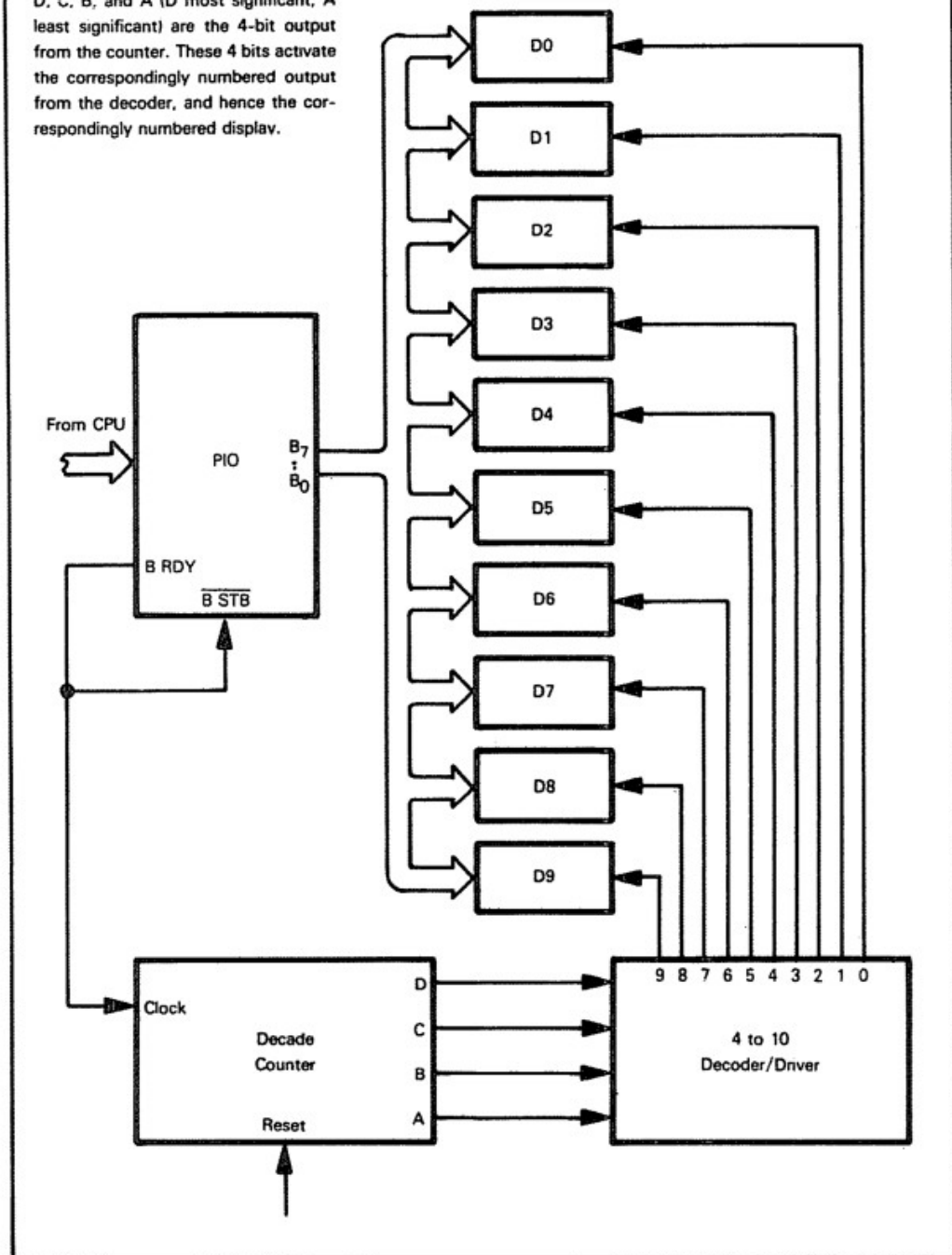
Figure 11-19. Multiplexed Seven-Segment Displays

11-52

Task 2: Display ten decimal digits.

Purpose: Display the contents of memory locations 0040 through 0049 on ten 7-segment displays that are multiplexed with a counter and a decoder.

Sample Problem:

```
(0040) = 66
(0041) = 3F
(0042) = 7F
(0043) = 7F
(0044) = 06
(0045) = 5B
(0046) = 07
(0047) = 4F
(0048) = 6D
(0049) = 7D
```
Display reads 4088127356

Source Program:

```
        LD      A,00001111B     ;MAKE PORT B OUTPUT
        OUT     (PIOCRB),A
DRUN:   LD      HL,40H          ;POINT TO START OF DATA
        LD      B,10            ;NUMBER OF DISPLAYS = 10
        LD      C,PIODRB        ;GET PORT NUMBER
DSPLY:  OUT1                    ;SEND DATA TO DISPLAY
        CALL    DELAY           ;WAIT 1 MS
        JR      NZ,DSPLY        ;COUNT DISPLAYS
        JR      DRUN            ;START ANOTHER SCAN
```

Here we must select the PIO output mode, since the circuit uses the handshake signals.

Note that OUTI sends the data to the output port addressed by Register C, increments the address in Register Pair HL, and decrements the counter in Register B. We have assumed that subroutine DELAY does not affect the Z flag so that it can be used afterwards for a conditional branch.

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	3E		LD	A,00001111B
0001	0F			
0002	D3		OUT	(PIOCRB),A
0003	PIOCRB			
0004	21	DRUN:	LD	HL,40H
0005	40			
0006	00			
0007	06		LD	B,10
0008	0A			
0009	0E		LD	C,PIODRB
000A	PIODRB			
000B	ED	DSPLY:	OUTI	
000C	A3			
000D	CD		CALL	DELAY
000E	30			
000F	00			
0010	20		JR	NZ,DSPLY
0011	F9			
0012	18		JR	DRUN
0013	F0			

PROBLEMS

1) An On-Off Pushbutton

Purpose: Each closure of the pushbutton complements (inverts) all the bits in memory location 0040. The location initially contains zero. The program should continuously examine the pushbutton and complement location 0040 with each closure. You may wish to complement a display output port instead, so as to make the results easier to see.

Sample Case:

Location 0040 initially contains zero.

The first pushbutton closure changes location 0040 to FF (hex), the second changes it back to zero, the third back to FF (hex), etc. Assume that the pushbutton is debounced in hardware. How would you include debouncing in your program?

2) Debouncing a Switch in Software

Purpose: Debounce a mechanical switch by waiting until two readings, taken a debounce time apart, give the same result. Assume that the debounce time (in ms) is in memory location 0040 and place the switch position into memory location 0041.

Sample Problem:

(0040) = 03 causes the program to wait 3 ms between readings.

3) Control for a Rotary Switch

Purpose: Another switch serves as a Load switch for a four-position unencoded rotary switch. The CPU waits for the Load switch to close (be zero), and then reads the position of the rotary switch. This procedure allows the operator to move the rotary switch to its final position before the CPU tries to read it. The program should place the position of the rotary switch into memory location 0040. Debounce the Load switch in software.

Sample Problem:

Place rotary switch in position 2. Close Load switch.

Result: (0040) = 02

4) Record Switch Positions on Lights

Purpose: A set of eight switches should have their positions reflected in eight LEDs. That is to say, if the switch is closed (zero), the LED should be on, otherwise the LED should be off. Assume that the CPU output port is connected to the cathodes of the LEDs.

Sample Problem:

        SWITCH 0 CLOSED
        SWITCH 1 OPEN
        SWITCH 2 CLOSED
        SWITCH 3 OPEN
        SWITCH 4 OPEN
        SWITCH 5 CLOSED
        SWITCH 6 CLOSED
        SWITCH 7 OPEN
    Result:
        LED   0 ON
        LED   1 OFF
        LED   2 ON
        LED   3 OFF
        LED   4 OFF
        LED   5 ON
        LED   6 ON
        LED   7 OFF

How would you change the program so that a switch attached to bit 7 of Port A of PIO #2 determines whether or not the displays are active (i.e., if the control switch is closed, the displays attached to Port B reflect the switches attached to Port A; if the control switch is open, the displays are always off)? A control switch is useful when the displays may distract the operator, as in an airplane.

How would you change the program so as to make the control switch an on-off pushbutton; that is, each closure reverses the previous state of the displays? Assume that the displays start in the active state and that the program examines and debounces the pushbutton before sending data to the displays.

5) Count on a Seven-Segment Display

Purpose: The program should count from 0 to 9 continuously on a seven-segment display, starting with zero.

Hint: Try different timing lengths for the displays and see what happens. When does the count become visible? What happens if the display is blanked part of the time?

MORE COMPLEX I/O DEVICES

More complex I/O devices differ from simple keyboards, switches, and displays in that:

1) They transfer data at higher rates.

2) They may have their own internal clocks and timing.

3) They produce status information and require control information, as well as transferring data.

Because of their high data rates, you cannot handle these I/O devices casually. If the processor does not provide the appropriate service, the system may miss input data or produce erroneous output data. You are therefore working under much more exacting constraints than in dealing with simpler devices. Interrupts are a convenient method for handling complex I/O devices, as we shall see in Chapter 12.

Peripherals such as keyboards, teletypewriters, cassettes, and floppy disks produce their own internal timing. These devices provide streams of data, separated by specific timing intervals. The computer must synchronize the initial input

<table>
<tr><td>SYNCHRONIZING
WITH I/O
DEVICES</td></tr>
</table>

or output operation with the peripheral clock and then provide the proper interval between subsequent operations. A simple delay loop like the one shown previously can produce the timing interval. The synchronization may require one or more of the following procedures:

1) Looking for a transition on a clock or strobe line provided by the peripheral for timing purposes. A simple approach would be to tie the strobe to a PIO $\overline{STB}$ input and look for a change in the interrupt ($\overline{INT}$) output. However, there is no way to directly address the $\overline{INT}$ output (and thus determine its value) and no way to clear it other than through an interrupt service routine. Thus, to use the PIO in a polling system, one must make the strobe available at an input port and latch it if necessary. If the strobe must be latched, a circuit must also be provided to clear the latch as part of the subsequent input or output transfer.

2) Finding the center of the time interval during which the data is stable. We would prefer to determine the value of the data at the center of the pulse rather than at the edges, where the data may be changing. Finding the center requires a delay of one-half of a transmission interval (bit time) after the edge. Sampling the data at the center also means that small timing errors have little effect on the accuracy of the reception.

3) Recognizing a special starting code. This is easy if the code is a single bit or if we have some timing information. The procedure is more complex if the code is long and could start at any time. Shifting will be necessary to determine where the transmitter is starting its bits, characters, or messages (this is often called a search for the correct "framing").

4) Sampling the data several times. This reduces the probability of receiving data incorrectly from noisy lines. Majority logic (such as best 3 out of 5 or 5 out of 8) can be used to decide on the actual data value.

Reception is, of course, much more difficult than transmission, since the peripheral controls the reception and the computer must interpret timing information generated by the peripheral. In transmission, the computer provides the proper timing and formatting for a specific peripheral.

Peripherals may require or provide other information besides data and timing. We refer to other information transmitted by the computer as "control information"; it may select modes of operation, start or stop processes, clock registers, enable

<table>
<tr><td>CONTROL
AND STATUS
INFORMATION</td></tr>
</table>

buffers, choose formats or protocols, provide operator displays, count operations, or

identify the type and priority of the operation. We refer to other information transmitted by the peripheral as "status information"; it may indicate the mode of operation, the readiness of devices, the presence of error conditions, the format of protocol in use, and other states or conditions.

The computer handles control and status information just like data. This information seldom changes, even though actual data may be transferred at a high rate. The control or status information may be single bits, digits, words, or multiple words. Often single bits or short fields are combined and handled by a single input or output port.

Combining status and control information into bytes reduces the total number of I/O port addresses required by the peripherals. However, the combination does mean that individual status input bits must be separately interpreted and control output bits must be separately determined. The procedures for isolating status bits and setting or resetting control bits are as follows:

Separating Out Status Bits

```
┌─────────────────┐
│ SEPARATING      │
│ STATUS          │
│ INFORMATION     │
└─────────────────┘
```

Step 1)   Read status data from the peripheral

Step 2)   Logical AND with a mask (the mask has ones in bit positions that must be examined and zeros elsewhere)

Step 3)   Shift the separated bits to the least significant bit positions

If the field is a single bit, Step 2 is unnecessary since we can test the bit with the BIT instruction. If the single bit is in the most significant, next to most significant, or least significant position, we can use shift logical (AND A or OR A) instructions to determine its value. Remember also that the input instructions with register indirect addressing (e.g., IN A,(C)) affect the Sign flag. These somewhat more accessible bit positions are often reserved for the most frequently used status information. You should try to write the required instruction sequences for the Z80 processor.

Step 3 is unnecessary if the field is a single bit, since the Zero flag will contain the complement of that bit after Step 2 (try it!). A Shift or Load instruction can replace Step 2 if the field is a single bit and occupies the least significant, most significant, or next to most significant bit position. These positions are often reserved for the most frequently used status information. You should try to write the required instruction sequences for the 6800 processor.

Setting and Clearing Control Bits

```
┌─────────────────┐
│ COMBINING       │
│ CONTROL         │
│ INFORMATION     │
└─────────────────┘
```

Step 1)   Read prior control information

Step 2)   Logical AND with mask to clear bits (mask has zeros in bit positions to be cleared, ones elsewhere)

Step 3)   Logically OR with mask to set bits (mask has ones in bit positions to be set, zeros elsewhere)

Step 4)   Send new control information to peripheral

Here again the procedure is simpler if the field is a single bit and occupies a position at the end of the word.

Some examples of separating and combining status bits are:

1) A 3-bit field in bit positions 2 through 4 of a PIO data register is a scaling factor. Place that factor into the Accumulator.

```
;
; READ STATUS DATA FROM INPUT PORT
;
        IN      A,(PIODR)       ;READ STATUS DATA
;
; MASK OFF SCALING FACTOR AND SHIFT
;
        AND     00011100B       ;MASK SCALING FACTOR
        RRCA                    ;SHIFT TWICE TO NORMALIZE
        RRCA
```

2) The Accumulator contains a 2-bit field that must be placed into bit positions 3 and 4 of a PIO data register.

```
;
; MOVE DATA TO FIELD POSITIONS
;
        RLA                     ;SHIFT DATA TO BIT POSITIONS 3 AND 4
        RLA
        RLA
        AND     00011000B       ;CLEAR OTHER BIT POSITIONS
        LD      B,A             ;SAVE NEW FIELD VALUE
;
; COMBINE NEW FIELD VALUE WITH OTHER DATA
;
        IN      A,(PIODR)       ;CLEAR OLD FIELD VALUE
        AND     11100111B
        OR      B               ;INSERT NEW FIELD VALUE
        OUT     (PIODR),A
```

DOCUMENTING STATUS AND CONTROL TRANSFERS

Documentation is a serious problem in handling control and status information. The meanings of status inputs or control outputs are seldom obvious. The programmer should clearly indicate the purposes of input and output operations in the comments, e.g., "CHECK IF READER IS ON," "CHOOSE EVEN PARITY OPTION," or "ACTIVATE BIT RATE COUNTER." The bit manipulation, Logical, and Shift instructions will otherwise be very difficult to remember, understand, or debug.

Table 11-6. Comparison Between Independent Connections
and Matrix Connections for Keyboards

Keyboard Size	Number of Lines with Independent Connections	Number of Lines with Matrix Connections
3 x 3	9	6
4 x 4	16	8
4 x 6	24	10
5 x 5	25	10
6 x 6	36	12
6 x 8	48	14
8 x 8	64	16

EXAMPLES

An Unencoded Keyboard

Purpose: Recognize a key closure from an unencoded 3 x 3 keyboard and place the number of the key that was pressed into the Accumulator.

Keyboards are just collections of switches (see Figure 11-20). Small numbers of keys are easiest to handle if each key is attached separately to a bit of an input port. Interfacing the keyboard is then the same as interfacing a set of switches.

Keyboards with more than eight keys require more than one input port and therefore multibyte operations. This is particularly wasteful if the keys are logically separate, as in a calculator or ter-

MATRIX KEYBOARD

minal keyboard where the user will only strike one at a time. The number of input lines required may be reduced by connecting the keys into a matrix, as shown in Figure 11-21. Now each key represents a potential connection between a row and a column. The keyboard matrix requires n + m external lines, where n is the number of rows and m is the number of columns. This compares to n x m external lines if each key is sepa-rate. Table 11-6 compares the number of keys required by typical configurations.

A program can determine which key has been pressed by using the external lines from the matrix. The usual procedure is a "keyboard scan." We ground Row 0 and examine the column

KEYBOARD SCAN

lines. If any lines are grounded, a key in that row has been pressed, causing a row-to-column connection. We can determine which key was pressed by determining which column line is grounded; that is, which bit of the input port is zero. If no column line is grounded, we proceed to Row 1 and repeat the scan. Note that we can check to see if any keys at all have been pressed by grounding all the rows at once and examining the columns.

The keyboard scan requires that the row lines be tied to an output port and the column lines to an input port. Figure 11-22 shows the arrangement. The CPU can ground a par-ticular row by placing a zero in the appropriate bit of the output port and ones in the other bits.

The CPU can determine the state of a particular column by examining the appropriate bit of the input port.

Each key is a switch just like a pushbutton and grounds an input bit if it is pressed.
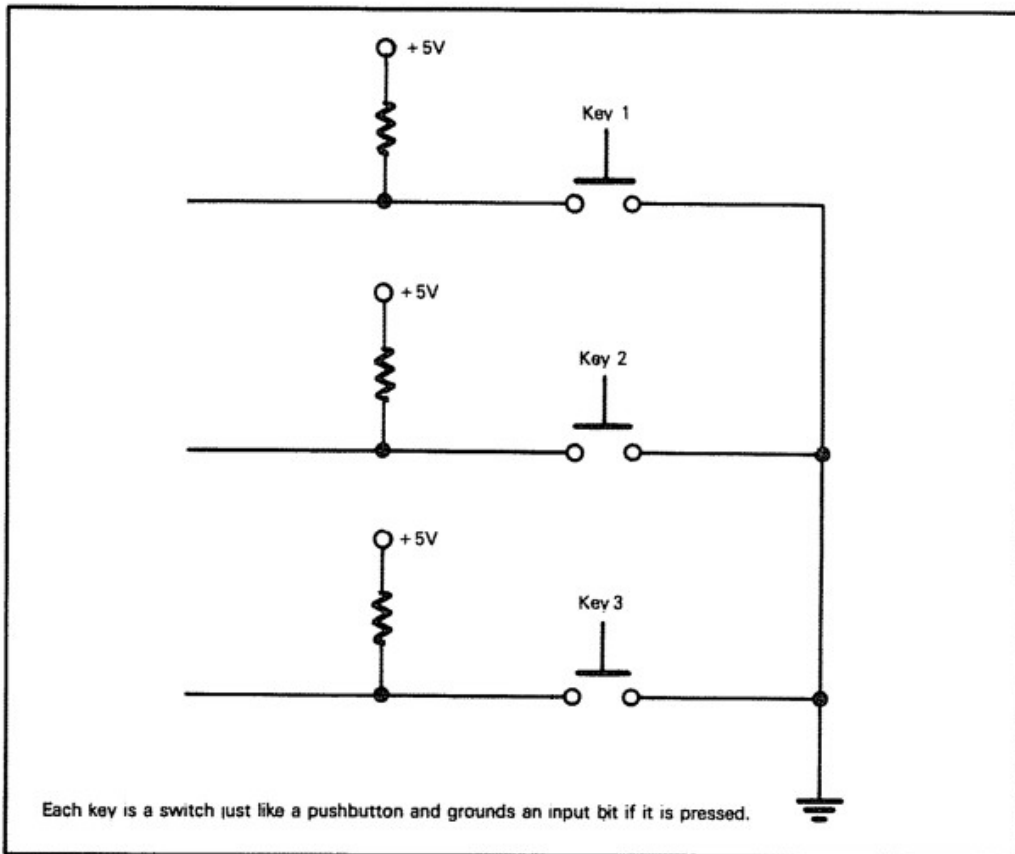
Figure 11-20. A Small Keyboard

Each key now serves to connect a row to a column. For instance, key 4 connects row 1 to column 1.
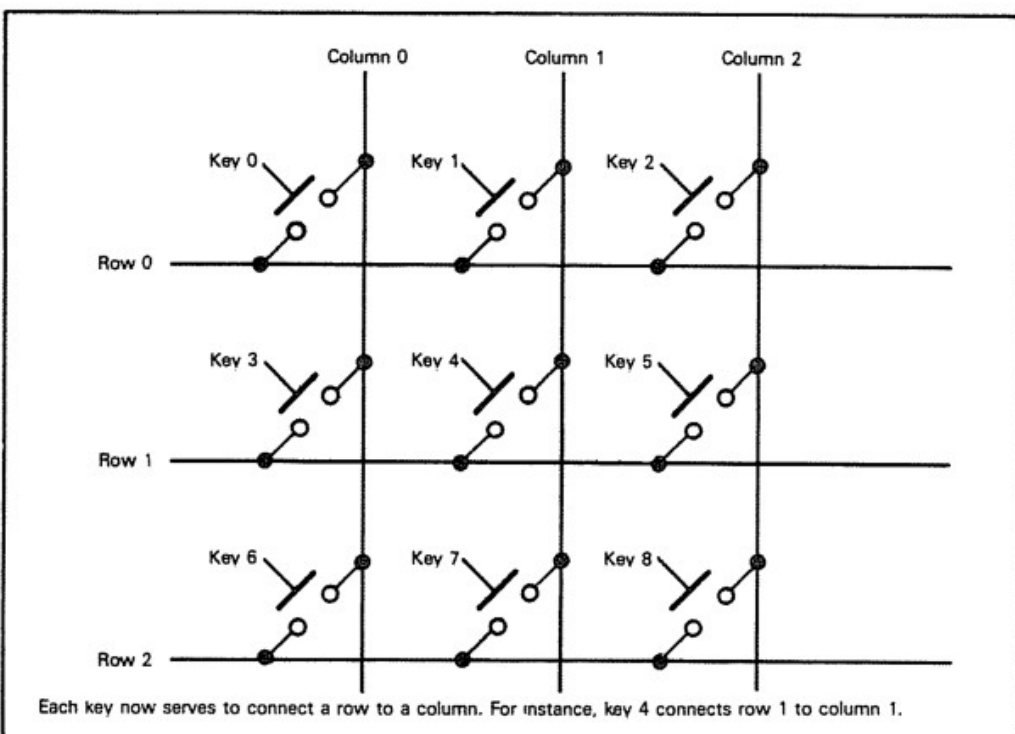
Figure 11-21. A Keyboard Matrix

11-61
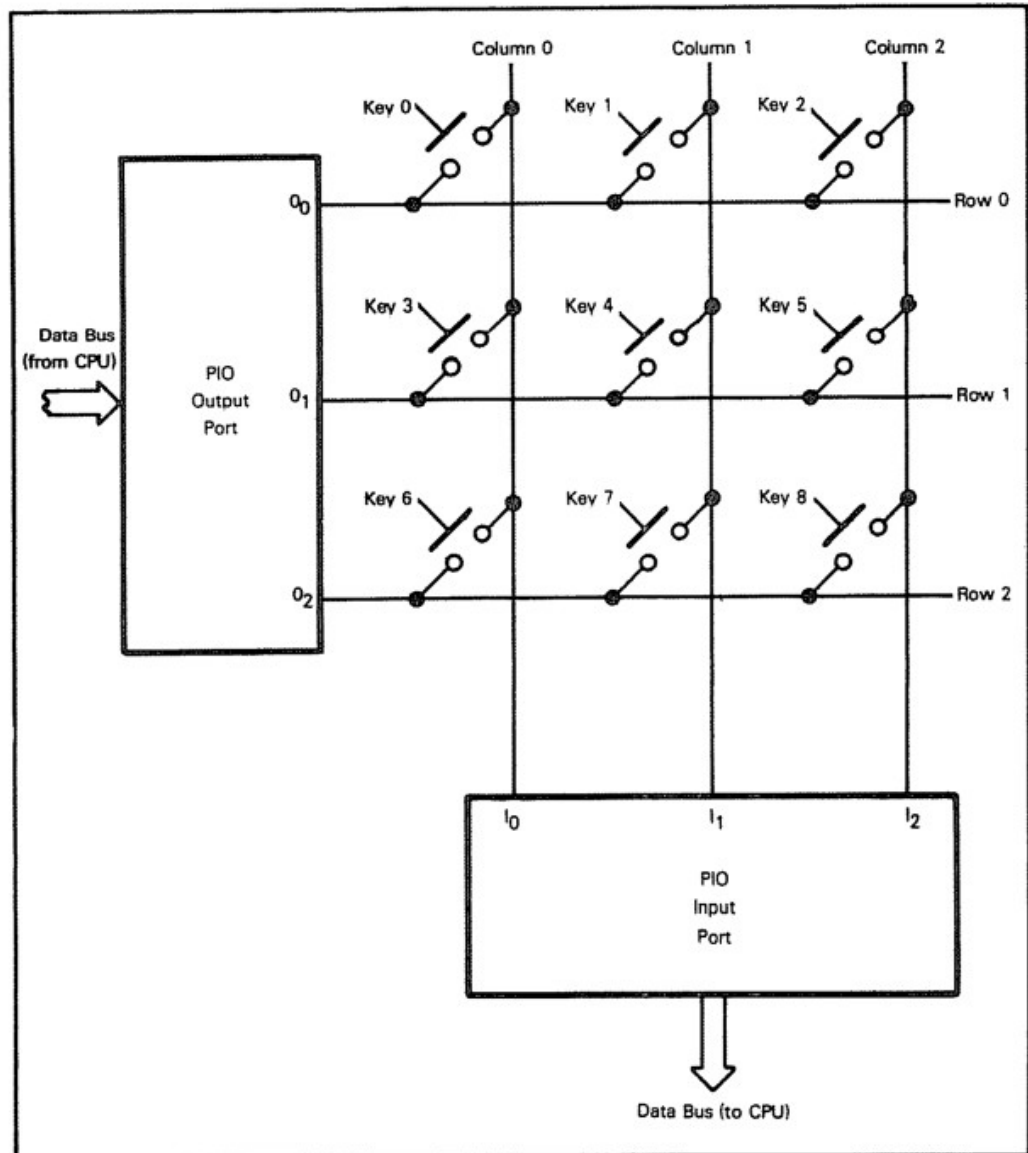
Figure 11-22. I/O Arrangement for a Keyboard Scan

Task 1: Determine key closure.

Purpose: Wait for a key to be pressed.

The procedure is as follows:

1) Ground all the rows by clearing all the output bits.
2) Fetch the column inputs by reading the input port.
3) Return to Step 1 if all the column inputs are ones.

Flowchart:
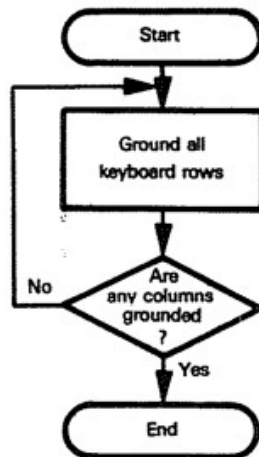
Source Program:

```
        LD      A.01001111B    :MAKE PORT A INPUT
        OUT     (PIOCRA),A
        LD      A,00001111B    :MAKE PORT B OUTPUT
        OUT     (PIOCRB),A
        SUB     A              :GROUND ALL KEYBOARD ROWS
        OUT     (PIODRB),A
WAITK:  IN      A,(PIODRA)     :GET KEYBOARD COLUMN DATA
        AND     00000111B      :MASK COLUMN BITS
        CP      00000111B      :ARE ANY COLUMNS GROUNDED?
        JR      Z.WAITK        :NO, WAIT UNTIL ONE IS
        HALT
```

Object Program:

Memory Location (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3E	LD	A,01001111B
0001	4F		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	3E	LD	A,00001111B
0005	0F		
0006	D3	OUT	(PIOCRB),A
0007	PIOCRB		
0008	97	SUB	A
0009	D3	OUT	(PIODRB),A
000A	PIODRB		
000B	DB	WAITK: IN	A,(PIODRA)
000C	PIODRA		
000D	E6	AND	00000111B
000E	07		
000F	FE	CP	00000111B
0010	07		
0011	28	JR	Z,WAITK
0012	F8		
0013	76	HALT	

PIO Port B is the keyboard output port and Port A is the input port.

Masking off the column bits eliminates any problems that could be caused by the states of the unused input lines.

We could generalize the routine by naming the output and masking patterns:

        ALLG   EQU   11111000B
        OPEN   EQU   00000111B

These names could then be used in the actual program; a different keyboard would require only a change in the definitions and a re-assembly.

Of course, one port of a PIO is all that is really necessary for a 3 x 3 or 4 x 4 keyboard. Try rewriting the program so that it uses only Port A. The PIO must be placed into the control mode so that lines can be individually selected as inputs or outputs.

Task 2: Identify key.

Purpose: Identify a key closure by placing the number of the key into the Accumulator.

The procedure is as follows:

1) Set key number to -1, counter to number of rows, and output pattern to all ones except for a zero in bit 0.
2) Ground a row by sending the output pattern to the keyboard output port.
3) Update the output pattern by shifting the zero bit left one position.
4) Fetch the column inputs by reading the input port.
5) If any column inputs are zero, proceed to Step 8.
6) Add the number of columns to the key number to reach the next row.
7) Decrement counter. Go to Step 2 if any rows have not been scanned, otherwise to Step 10.
8) Add 1 to key number. Shift column inputs right one bit.
9) If Carry = 1, return to Step 8.
10) End of program.

Flowchart:
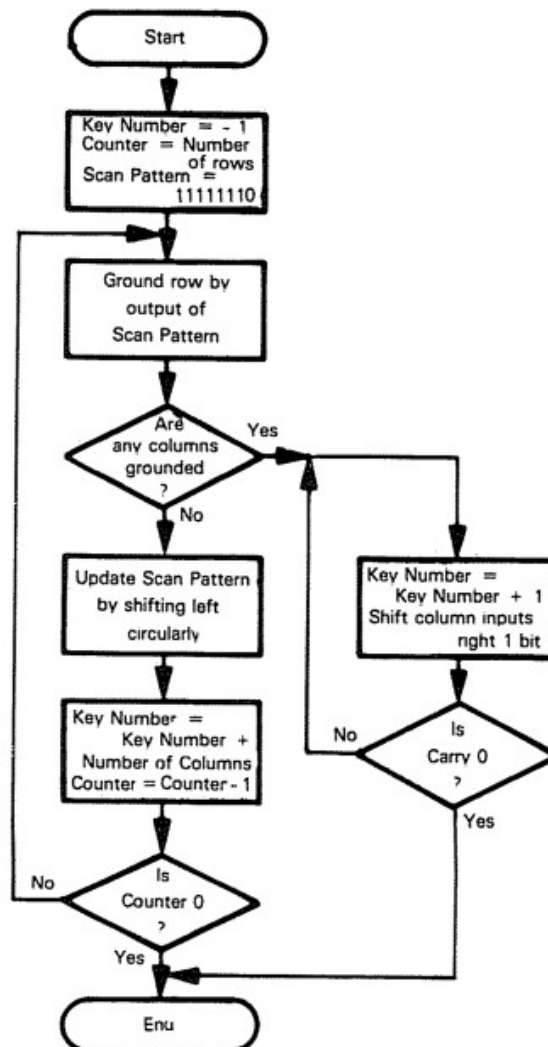
11-65

Source Program:

```
            LD      A,01001111B     ;MAKE PORT A INPUT
            OUT     (PIOCRA),A
            LD      A,00001111B     ;MAKE PORT B OUTPUT
            OUT     (PIOCRB),A
            LD      B,3             ;COUNT = NUMBER OF ROWS
            LD      C,PIODRB        ;GET OUTPUT PORT NUMBER
            LD      D,3             ;GET NUMBER OF COLUMNS
            LD      E,11111110B     ;START SCAN PATTERN TO GROUND ROW
                                    :   ZERO
            LD      H,00000111B     ;GET KEYBOARD MASKING PATTERN
            LD      L,0FFH          ;KEY NUMBER = -1
FROW:       OUT     (C),E           ;SCAN A ROW
            RLC     E               ;UPDATE SCAN PATTERN FOR NEXT ROW
            IN      A,(PIODRA)      ;GET KEYBOARD COLUMN DATA
            AND     H               ;MASK COLUMN BITS
            CP      H               ;ARE ANY COLUMNS GROUNDED?
            JR      NZ,FCOL         ;YES, GO FIND WHICH ONE
            LD      A,L             ;NO, UPDATE KEY NUMBER FOR NEXT ROW
            ADD     A,D
            LD      L,A
            DJNZ    FROW            ;EXAMINE NEXT ROW IF ANY LEFT
            INC     L               ;IDENTIFY CASE IN WHICH KEY NOT FOUND
            JR      DONE
FCOL:       INC     L               ;INCREMENT KEY NUMBER
            RRA                     ;IS THIS COLUMN GROUNDED?
            JR      NC,FCOL         ;NO, EXAMINE NEXT COLUMN
DONE:       HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	3E		LD	A,00001111B
0005	0F			
0006	D3		OUT	(PIOCRB),A
0007	PIOCRB			
0008	06		LD	B,3
0009	03			
000A	0E		LD	C,PIODRB
000B	PIODRB			
000C	16		LD	D,3
000D	03			
000E	1E		LD	E,11111110B
000F	FE			
0010	26		LD	H,00000111B
0011	07			
0012	2E		LD	L, 0FFH
0013	FF			
0014	ED	FROW:	OUT	(C),E
0015	59			
0016	CB		RLC	E
0017	03			
0018	DB		IN	A,(PIODRA)
0019	PIODRA			
001A	A4		AND	H
001B	BC		CP	H
001C	20		JR	NZ,FCOL
001D	08			
001E	7D		LD	A,L
001F	82		ADD	A,D
0020	6F		LD	L,A
0021	10		DJNZ	FROW
0022	F1			
0023	2C		INC	L
0024	18		JR	DONE
0025	04			
0026	2C	FCOL:	INC	L
0027	1F		RRA	
0028	30		JR	NC,FCOL
0029	FC			
002A	76		HALT	

Each time a row scan fails, we must add the number of columns to the key number so as to move past the present row (try it on the keyboard in Figure 11-22).

What is the result of the program if no keys are being pressed? Note the extra INC L instruction so that the program differentiates between no keys pressed and the last key being pressed. What is the final value in the Accumulator for these two cases? Note that the Zero flag could also be used to distinguish the case where no keys were pressed. Can you explain how?

An alternative approach would be to use the PIO in its control mode so that lines could be changed from inputs to outputs. The procedure would be:

1) Ground all the columns and save the row inputs.
2) Ground all the rows and save the column inputs.
3) Use the row and column inputs together to determine the key number from a table.

Try to write a program to implement this procedure.

This program can be generalized by making the number of rows, the number of columns, and the masking pattern into named parameters with EQU pseudo-operations.
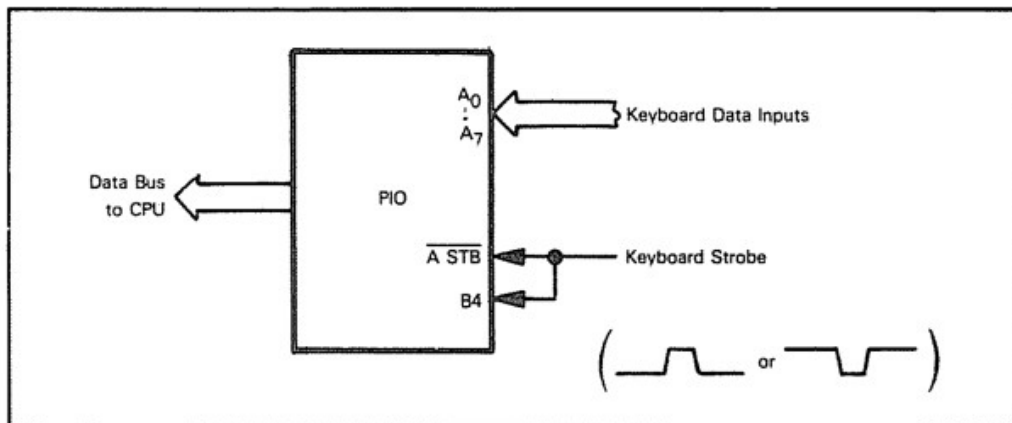
Figure 11-23. I/O Interface for an Encoded Keyboard

An Encoded Keyboard

Purpose: Fetch data, when it is available, from an encoded keyboard that provides a strobe along with each data transfer.

An encoded keyboard provides a unique code for each key. It has internal electronics that perform the scanning and identification procedure of the previous example. The tradeoff is between the simpler software required by the encoded keyboard and the lower cost of the unencoded keyboard.

Encoded keyboards may use diode matrices, TTL encoders, or MOS encoders. The codes may be ASCII, EBCDIC, or a custom code. PROMs are often part of the encoding circuitry.

The encoding circuitry may do more than just encode key **ROLLOVER**
closures. It may also debounce the keys and handle "rollover," the
problem of more than one key being struck at the same time. Common ways of handling rollover are: "2-key rollover," whereby two keys (but not more) struck at the same time are resolved into separate closures, and "n-key rollover," whereby any number of keys struck at the same time are resolved into separate closures.

The encoded keyboard also provides a strobe with each data transfer. The strobe signals that a new closure has occurred. Figure 11-23 shows the interface between an encoded keyboard and the Z80 microprocessor. The rising edge of the strobe latches the data into the input port. We also tie the strobe to the B side of the PIO so that the CPU can determine when a rising edge has occurred. Of course, the B port of one PIO could hold status signals from up to eight ports. The software would then have to determine which ports were active with a shifting and masking operation.

We have assumed in the program that the strobe signal is long enough for the CPU to handle it in software. If it is not, the signal will have to be latched and cleared (with RDY) when the input or output transfer occurs.

You may have to watch the polarity of the strobe, since the PIO always reacts to a rising edge. An inverter gate may be necessary.

Task: Input from keyboard.

Purpose: Wait for the rising edge of a strobe at the B port of a PIO and then place the data from Port A into the Accumulator.

Flowchart:
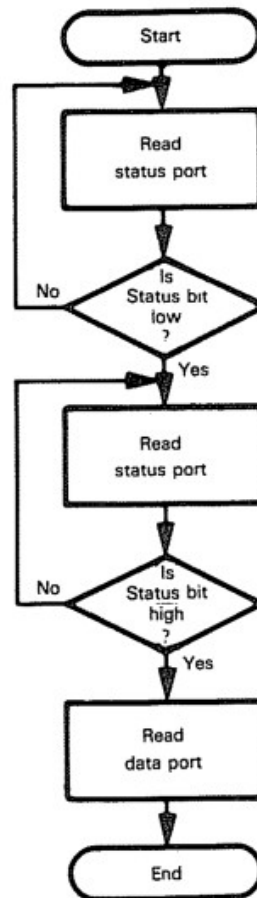
The hardware must hold the control lines in a logic one state during reset to prevent the accidental setting of status flags.

Source Program:

```
        LD      A,01001111B     ;MAKE PORT A INPUT
        OUT     (PIOCRA),A
        LD      A,11001111B     ;MAKE PORT B CONTROL
        OUT     (PIOCRB),A
        LD      A,0FFH          ;ALL PORT B LINES INPUTS
        OUT     (PIOCRB),A
SRCHL:  IN      A,(PIODRB)      ;EXAMINE STATUS PORT
        BIT     STB,A           ;HAS STROBE LINE GONE LOW?
        JR      NZ,SRCHL        ;NO, WAIT UNTIL IT HAS
SRCHH:  IN      A,(PIODRB)      ;EXAMINE STATUS PORT AGAIN
        BIT     STB,A           ;RISING EDGE FOUND?
        JR      Z,SRCHH         ;NO, WAIT UNTIL ONE OCCURS
        IN      A,(PIODRA)      ;YES, FETCH DATA
        HALT
```

11-70

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	3E		LD	A,11001111B
0005	CF			
0006	D3		OUT	(PIOCRB),A
0007	PIOCRB			
0008	3E		LD	A,0FFH
0009	FF			
000A	D3		OUT	(PIOCRB),A
000B	PIOCRB			
000C	DB	SRCHL:	IN	A,(PIODRB)
000D	PIODRB			
000E	CB		BIT	STB,A
000F	STB			
0010	20		JR	NZ,SRCHL
0011	FA			
0012	DB	SRCHH:	IN	A,(PIODRB)
0013	PIODRB			
0014	CB		BIT	STB,A
0015	STB			
0016	28		JR	Z,SRCHH
0017	FA			
0018	DB		IN	A,(PIODRA)
0019	PIODRA			
001A	76		HALT	

If the CPU repeats this routine, it will not fetch another character until the next rising edge occurs on the strobe line. A continuing high level on the strobe line will be ignored.

STB depends on which bit of Port B is used. Figure 11-23 shows bit 4 being used, but bits 0, 6, and 7 are, as usual, the easiest to examine. Try rewriting the program to use the more accessible bit positions.

The second byte of the Bit instructions depends on the value of STB but is not equal to that value. For example, the second byte is $4F_{16}$ if STB = 1, 57_{16} if STB = 2, etc.

A Digital-to-Analog Converter

Purpose: Send data to an 8-bit digital-to-analog converter. which has an active-low latch enable.

Digital-to-analog converters produce the continuous signals required by solenoids, relays, actuators, and other electrical and mechanical output devices. Typical converters consist of switches and resistor ladders with the appropriate resistance values.[7] The user must generally provide a reference voltage and some other digital and analog circuitry, although complete units are becoming available at low cost.

Figure 11-24 describes the 8-bit Signetics NE5018 D/A converter. which contains an on-chip 8-bit parallel data input latch. A low level on the $\overline{\text{LE}}$ (Latch Enable) input gates the input data into the latches. where it remains after $\overline{\text{LE}}$ goes high.

Figure 11-25 illustrates the interfacing of the device to a Z80 microprocessor. Here the A side of the PIO is used to generate the Latch Enable signal. The RDY line from the PIO could be used in the mode where it is tied to the $\overline{\text{STB}}$ line to form a pulse lasting one clock cycle. However, one clock cycle may not be long enough, since the NE5018 requires a 400 ns pulse. Furthermore, the polarity is the opposite of that needed by the NE5018.

Note that the PIO latches the output data. The data therefore remains stable during and after the conversion. The converter typically requires only a few microseconds to produce an analog output. Thus, the converter latch could be left enabled if the port were not used for any other purpose.

In applications where eight bits of resolution are not enough, 10- to 16-bit converters can be used. Additional port logic is required to pass all the data bits; some converters provide part of this logic.

The PIO here serves both as a parallel data port and as a serial control port. Of course, if Port A is used for control, it could actually handle up to eight bits.

Task: Output to converter.

Purpose: Send data from memory location 0040 to the converter.

Flowchart:

Figure 11-24. Signetics NE5018 D/A Converter

All R values equal 5k Ω and are thermally matched

11-73
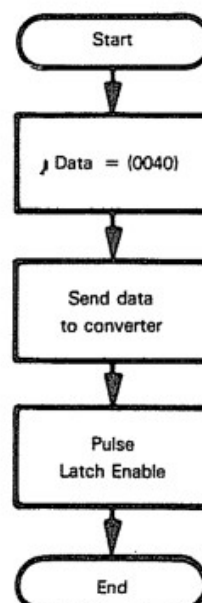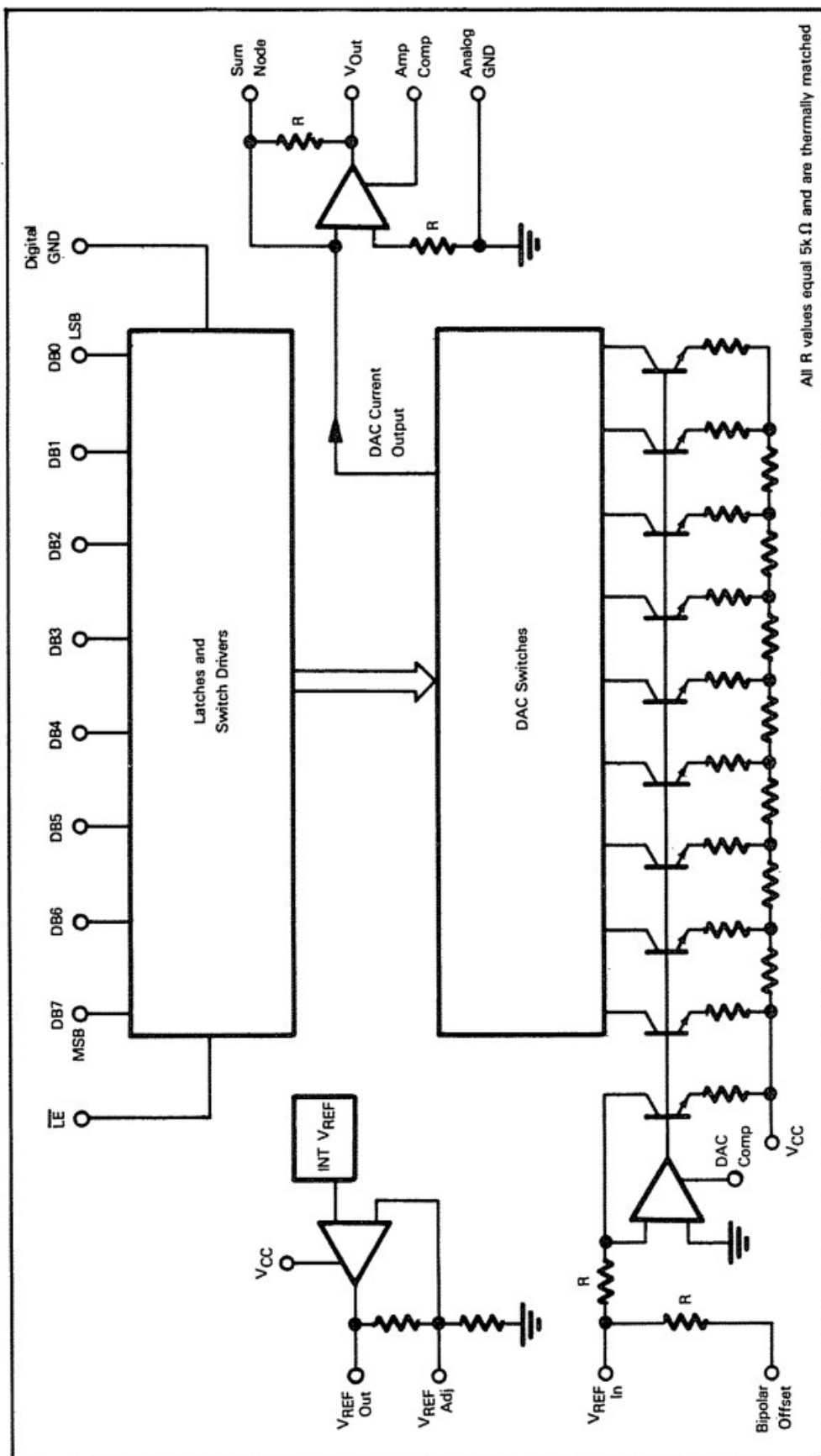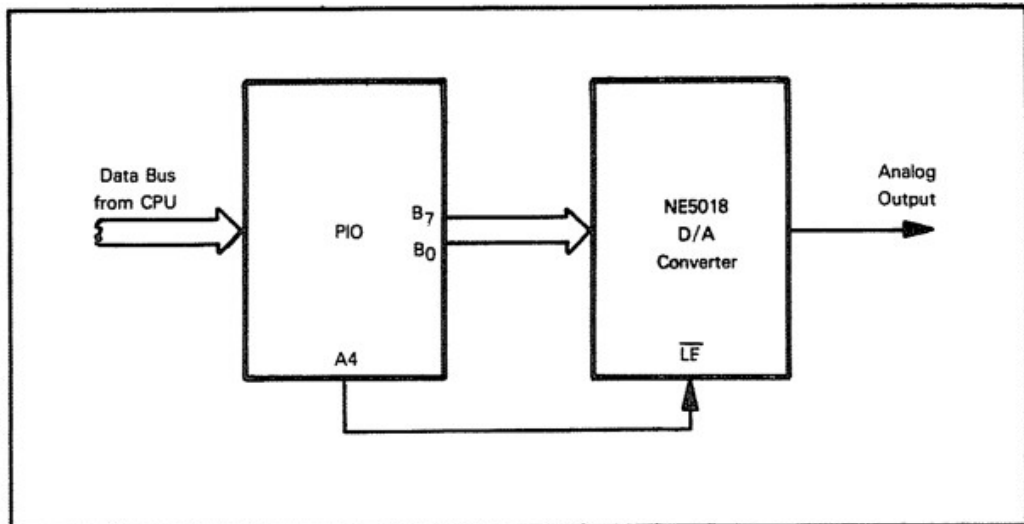
Figure 11-25. Interface for an 8-bit Digital-to-Analog Converter

Source Program:

```
LD      A,11001111B    ;MAKE PORT A CONTROL
OUT     (PIOCRA),A
SUB     A              ;ALL PORT A PINS OUTPUTS
OUT     (PIOCRA),A
LD      A,00001111B    ;MAKE PORT B OUTPUT
OUT     (PIOCRB),A
LD      A,(40H)        ;GET DATA
OUT     (PIODRB),A     ;SEND DATA TO DAC
IN      A,(PIODRA)     ;GET OLD CONTROL DATA
RES     4,A            ;BRING LATCH ENABLE LOW
OUT     (PIODRA),A
SET     4,A            ;BRING LATCH ENABLE HIGH
OUT     (PIODRA),A
HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3E	LD	A,11001111B
0001	CF		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	97	SUB	A
0005	D3	OUT	(PIOCRA),A
0006	PIOCRA		
0007	3E	LD	A,00001111B
0008	0F		
0009	D3	OUT	(PIOCRB),A
000A	PIOCRB		
000B	3A	LD	A,(40H)
000C	40		
000D	00		
000E	D3	OUT	(PIODRB),A
000F	PIODRB		
0010	DB	IN	A,(PIODRA)
0011	PIODRA		
0012	CB	RES	4,A
0013	A7		
0014	D3	OUT	(PIODRA),A
0015	PIODRA		
0016	CB	SET	4,A
0017	E7		
0018	D3	OUT	(PIODRA),A
0019	PIODRA		
001A	76	HALT	

The particular bit that must be set and reset depends, of course, on how the Latch Enable is connected to the control port. Bit 0 is often convenient to use for control purposes since, if that bit is originally cleared, it can be set with an INC instruction and reset with a DEC instruction.

We could use the automatic brief strobe from B ACK if the Latch Enable were active-high (and if this strobe were long enough when B ACK is tied back to $\overline{\text{B STB}}$). The program would then be:

```
LD    A,00001111B    ;MAKE PORT B OUTPUT
OUT   (PIOCRB),A
LD    A,(40H)         ;GET DATA
OUT   (PIODRB),A      ;SEND DATA TO DAC AND ENABLE LATCH
HALT
```

An inverter gate could produce an active-low signal. Note how many fewer instructions are necessary.

Analog-to-Digital Converter

Purpose: Fetch data from an 8-bit analog-to-digital converter that requires an Initiate Conversion pulse to start the conversion process and has a Data Valid line to indicate the completion of the process and the availability of valid data.

Analog-to-digital converters handle the continuous signals produced by various types of sensors and transducers.[8] The converter produces the digital input which the computer requires.

One form of analog-to-digital converter is the successive approximation device, which makes a direct 1-bit comparison during each clock cycle. Such converters are fast but have little noise immunity. Dual slope integrating converters are another form of analog-to-digital converter. These devices take longer but are more resistant to noise. Other techniques, such as the incremental charge balancing technique, are also used.

Analog-to-digital converters usually require some external analog and digital circuitry, although complete units are becoming available at low cost.

Figure 11-26 shows the 8-bit Teledyne Semiconductor 8703 A/D converter. The device contains a result latch and tristate data outputs. A pulse on the Initiate Conversion line starts conversion of the analog input; after about two milliseconds the result will go to the output latches, and the Data Valid output will indicate this by switching first low and then high. Data is read from the latches by applying '0' to the ENABLE input.

Figure 11-27 shows the interface for the Z80 processor and the 8703 converter.[9] Port B is used to provide an Initiate Conversion pulse (active-high) of sufficient length. The Data Valid signal is tied to A STB so that Data Valid going low and then high will latch the converted data into Port A. The Data Valid signal is also tied to a bit of Port B so that the CPU can determine its value. The important edge on the Data Valid line is the low-to-high edge, which indicates the completion of the conversion. As in the case of the encoded keyboard, additional circuitry will be necessary if the pulse on Data Valid is too short to be handled in software. Note that we are using Port B here for both status and control.
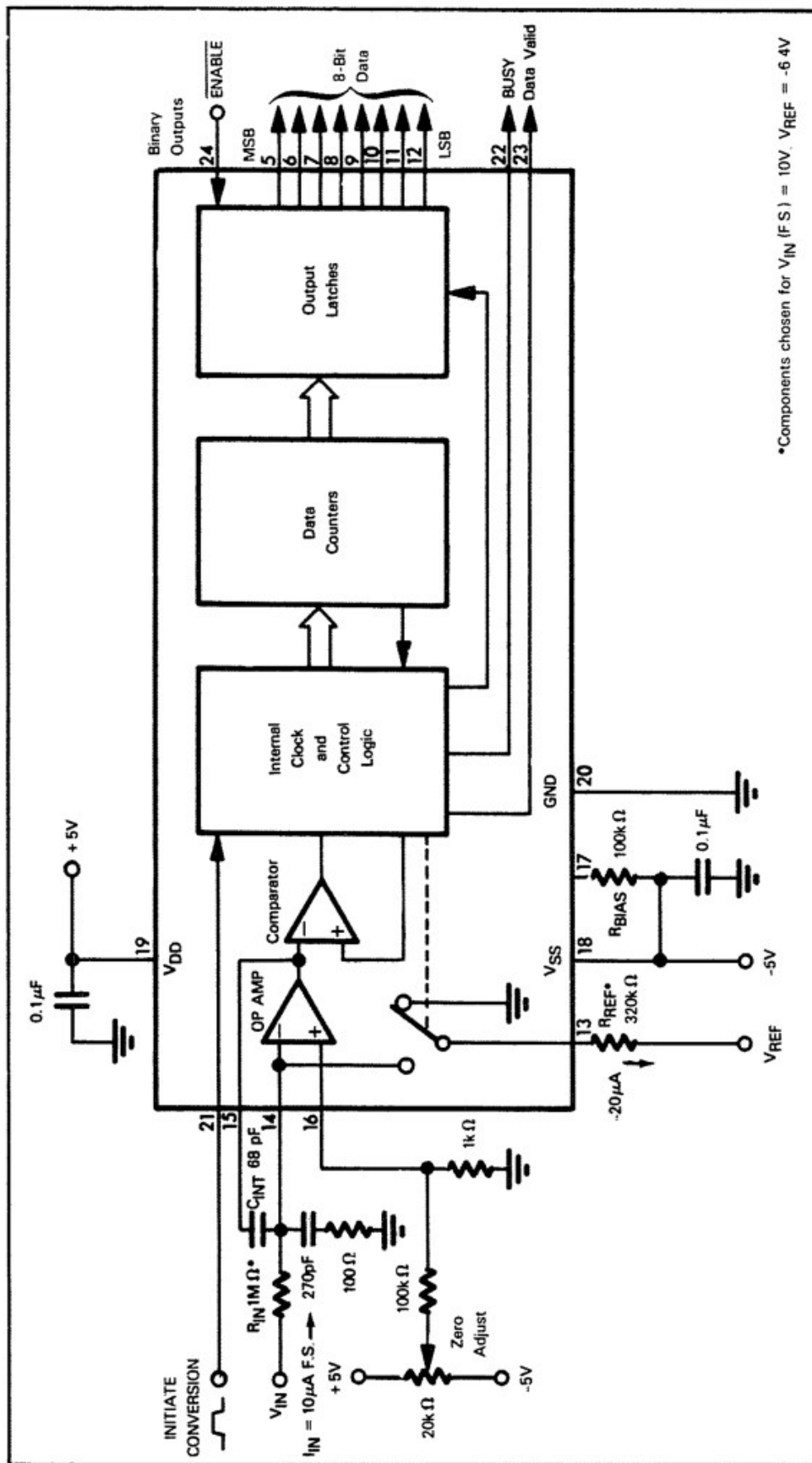
Figure 11-26. Teledyne 8703 A/D Converter
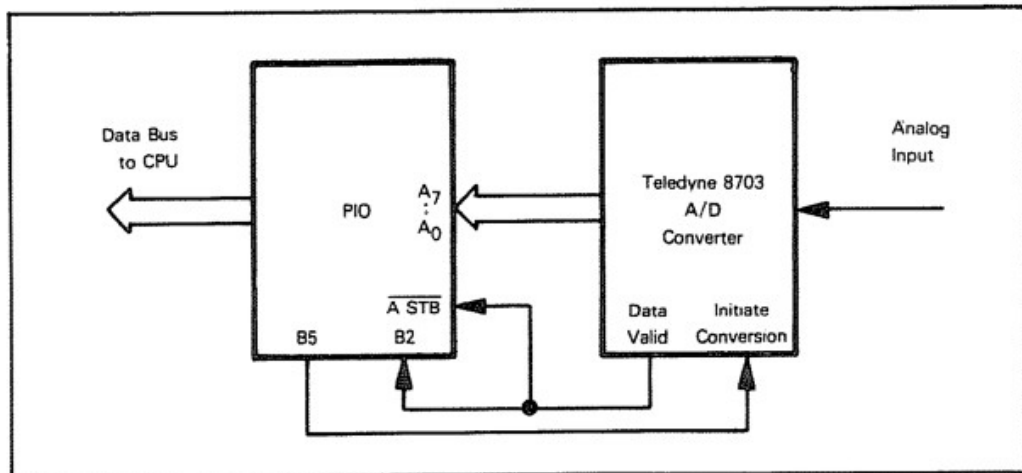
Figure 11-27. Interface for an 8-bit Analog-to-Digital Converter

Task: Input from converter.

Purpose: Start the conversion process, wait for Data Valid to go low and then high, and then read the data and store it in memory location 0040.

Flowchart:
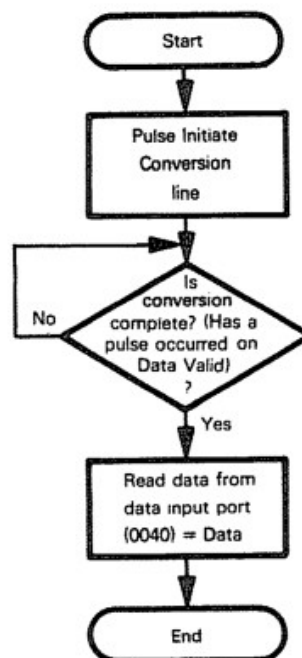
Note that here the PIO serves as a parallel data port, a serial status port, and a serial control port.

Source Program:

```
         LD     A,01001111B    ;MAKE PORT A INPUT
         OUT    (PIOCRA),A
         LD     A,11001111B    ;MAKE PORT B CONTROL
         OUT    (PIOCRB),A
         LD     A,00001111B    ;B4-7 OUTPUT. B0-3 INPUT
         OUT    (PIOCRB),A
         LD     A,00100000B    ;SEND INITIATE CONVERSION HIGH
         OUT    (PIODRB),A
         SUB    A              ;SEND INITIATE CONVERSION LOW
         OUT    (PIODRB),A
WTLOW:   IN     A,(PIODRB)     ;HAS DATA VALID GONE LOW?
         BIT    2,A
         JR     NZ,WTLOW       ;NO, WAIT
WTHI:    IN     A,(PIODRB)     ;IS DATA AVAILABLE?
         BIT    2,A
         JR     Z,WTLOW        ;NO, WAIT
         IN     A,(PIODRA)     ;YES. FETCH DATA FROM CONVERTER
         LD     (40H),A        ;SAVE CONVERTER DATA
         HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3E	LD	A,01001111B
0001	4F		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	3E	LD	A,11001111B
0005	CF		
0006	D3	OUT	(PIOCRB),A
0007	PIOCRB		
0008	3E	LD	A,00001111B
0009	0F		
000A	D3	OUT	(PIOCRB),A
000B	PIOCRB		
000C	3E	LD	A,00100000B
000D	20		
000E	D3	OUT	(PIODRB),A
000F	PIODRB		
0010	97	SUB	A
0011	D3	OUT	(PIODRB),A
0012	PIODRB		
0013	DB	WTLOW: IN	A,(PIODRB)
0014	PIODRB		
0015	CB	BIT	2,A
0016	57		
0017	20	JR	NZ,WTLOW
0018	FA		
0019	DB	WTHI: IN	A,(PIODRB)
001A	PIODRB		
001B	CB	BIT	2,A
001C	57		
001D	28	JR	Z,WTHI
001E	FA		
001F	DB	IN	A,(PIODRA)
0020	PIODRA		
0021	32	LD	(40H),A
0022	40		
0023	00		
0024	76	HALT	

One approach to configuring PIOs is to use the repeated Block Output instruction OTIR and a table in memory containing the words to be sent to the Control register. A typical routine would be:

```
LD    B,LENG       ;COUNT = NUMBER OF CONTROL WORDS
LD    C,PIOCR      ;GET CONTROL PORT NUMBER
LD    HL,CTLTAB    ;STARTING ADDRESS OF PIO CONTROL TABLE
OTIR              ;CONFIGURE PIO
```

In fact, another table (or the Stack) could be used to hold the number of control words and the port number for each PIO.
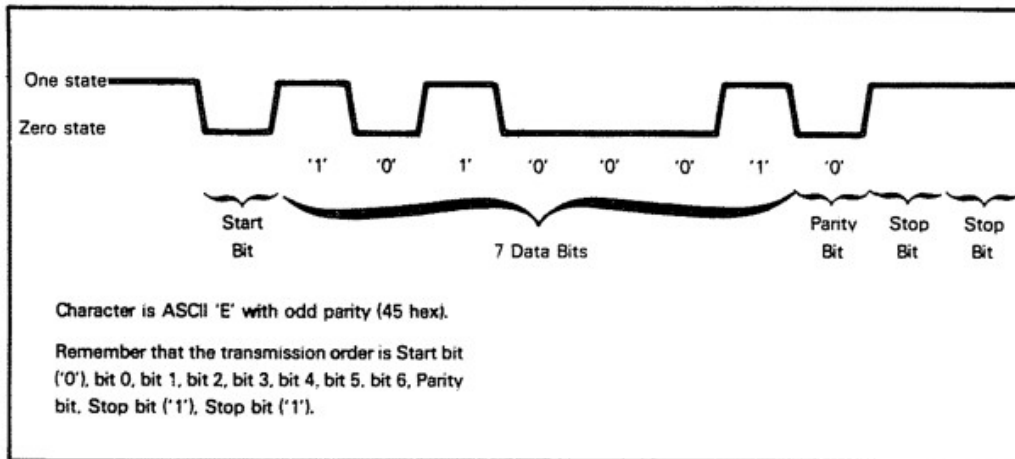
Figure 11-28. Teletypewriter Data Format

A Teletypewriter (TTY)

Purpose: Transfer data to and from a standard 10-character-per-second serial teletypewriter.

TTY
INTERFACE

The common teletypewriter transfers data in an asynchronous serial mode. The procedure is as follows:

1) The line is normally in the one state.

STANDARD
TTY

2) A Start bit (zero bit) precedes each character.

3) The character is usually 7-bit ASCII with the least significant bit transmitted first.

CHARACTER
FORMAT

4) The most significant bit is a Parity bit, which may be even, odd, or fixed at zero or one.

5) Two stop bits (logic one) follow each character.

Figure 11-28 shows the format. Note that each character requires the transmission of eleven bits, of which only seven contain information. Since the data rate is ten characters per second, the bit rate is 10 x 11, or 110 Baud. Each bit therefore has a width of 1/110 of a second, or 9.1 milliseconds. This width is an average; the teletypewriter does not maintain it to any high level of accuracy.

For a teletypewriter to communicate properly with a computer, the following procedures are necessary.

Receive (flowcharted in Figure 11-29):

TTY
RECEIVE
MODE

Step 1) Look for a Start bit (a logic zero) on the data line.

Step 2) Center the reception by waiting one-half bit time, or 4.55 milliseconds.

Step 3) Fetch the data bits, waiting one bit time before each one. Assemble the data bits into a word by first shifting the bit to the Carry and then circularly shifting the data with the Carry. Remember that the least significant bit is received first.

Step 4) Generate the received Parity and check it against the transmitted Parity. If they do not match, indicate a "Parity error."

Step 5) Fetch the Stop bits (waiting one bit time between inputs). If they are not correct (if both Stop bits are not one), indicate a "framing error."
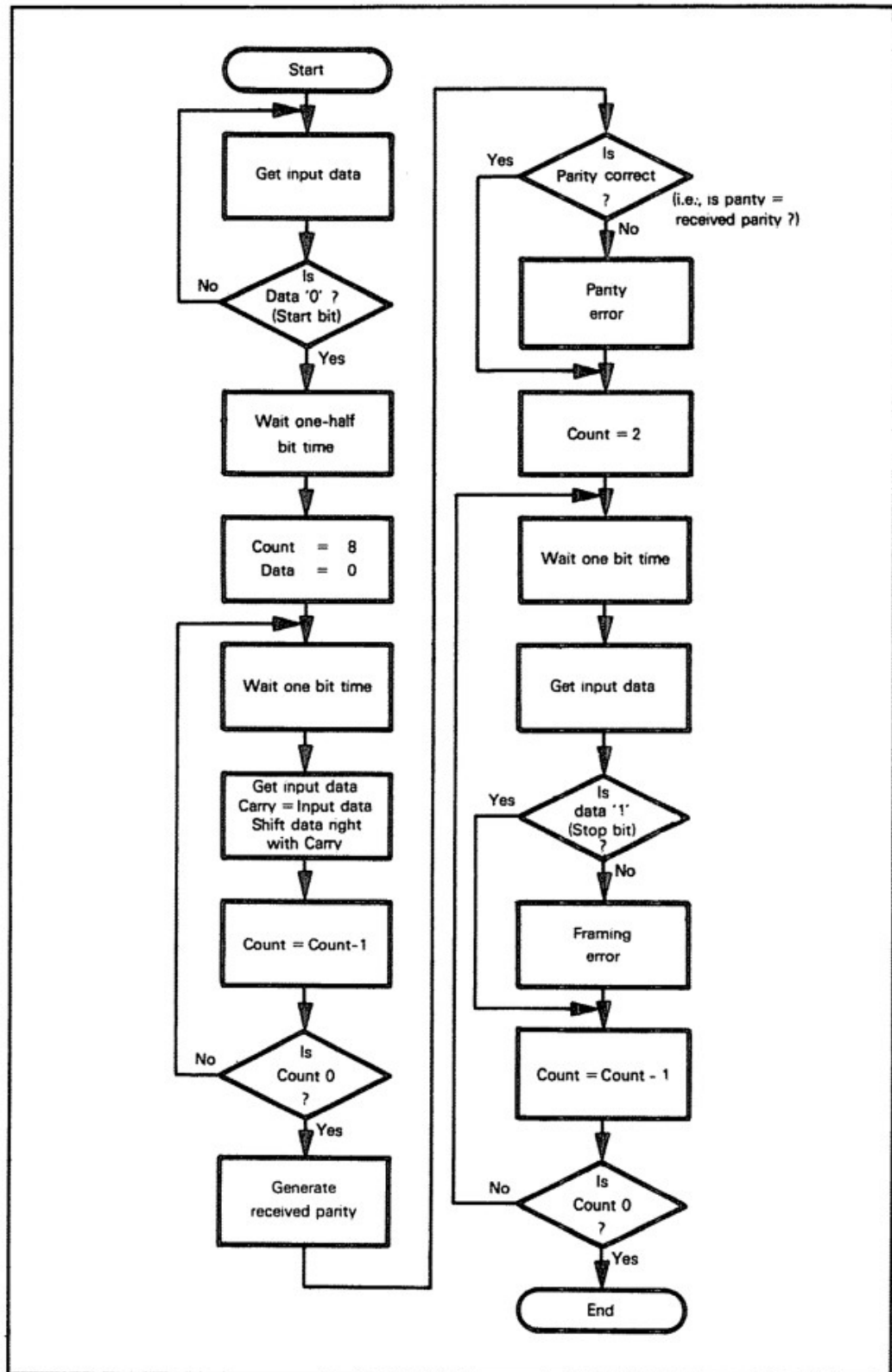
Figure 11-29. Flowchart for Receive Procedure

Task 1: Read data.

Purpose: Fetch data from a teletypewriter through bit 7 of a PIO data port and place the data into memory location 0060. For procedure, see Figure 11-29.

Source Program:

(Assume that the serial port is bit 7 of the PIO and that no parity or framing check is necessary)

```
        LD      A.01001111B    :MAKE PORT A INPUT
        OUT     (PIOCRA),A
WTSTB:  IN      A,(PIODRA)     :READ SERIAL LINE
        RLA                    :IS THERE A START BIT?
        JR      C,WTSTB        :NO, WAIT UNTIL THERE IS
        CALL    DHALF          :YES, DELAY HALF BIT TIME TO CENTER
        LD      D,10000000B    :COUNT WITH BIT IN MSB
RCVB:   CALL    DFULL          :WAIT 1 BIT TIME
        IN      A,(PIODRA)     :READ SERIAL LINE
        RLA                    :MOVE BIT TO CARRY
        RR      D              :MOVE BIT TO ASSEMBLED WORD
        JR      NC,RCVB        :CONTINUE IF COUNT BIT NOT IN CARRY
        LD      A,D
        LD      (60H),A
        HALT
```

(Delay program)

```
        ORG     30H
DHALF:  PUSH    DE             :SAVE OLD REGISTERS
        LD      D,8            :HALF BIT LENGTH COUNT
        JR      DLY16
DFULL:  PUSH    DE             :SAVE OLD REGISTERS
        LD      D,16           :FULL BIT LENGTH COUNT
DLY16:  LD      E,8DH          :DELAY 1/16TH BIT TIME
DLY1:   DEC     E
        JR      NZ,DLY1
        DEC     D
        JR      NZ,DLY16
        POP     DE             :RESTORE OLD REGISTERS
        RET
```

Remember that bit 0 of the data is received first.

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3E	LD	A,01001111B
0001	4F		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	DB	WTSTB: IN	A,(PIODRA)
0005	PIODRA		
0006	17	RLA	
0007	38	JR	C,WTSTB
0008	FB		
0009	CD	CALL	DHALF
000A	30		
000B	00		
000C	16	LD	D,10000000B
000D	80		
000E	CD	RCVB: CALL	DFULL
000F	35		
0010	00		
0011	DB	IN	A,(PIODRA)
0012	PIODRA		
0013	17	RLA	
0014	CB	RR	D
0015	1A		
0016	30	JR	NC,RCVB
0017	F6		
0018	7A	LD	A,D
0019	32	LD	(60H),A
001A	60		
001B	00		
001C	76	HALT	
0030	D5	DHALF: PUSH	DE
0031	16	LD	D,8
0032	08		
0033	18	JR	DLY16
0034	03		
0035	D5	DFULL: PUSH	DE
0036	16	LD	D,16
0037	10		
0038	1E	DLY16: LD	E,8DH
0039	8D		
003A	1D	DLY1: DEC	E
003B	20	JR	NZ,DLY1
003C	FD		
003D	15	DEC	D
003E	20	JR	NZ,DLY16
003F	F8		
0040	D1	POP	DE
0041	C9	RET	

This program assumes that the Stack can be used for subroutine calls, i.e., the monitor must initialize the Stack Pointer. Otherwise you will have to initialize the Stack Pointer as shown in Chapter 10.

The constants for the delay routine were calculated just as shown earlier in this chapter. You might try determining them for yourself. The delays do not have to be highly accurate because the reception is centered, the messages are short, the bit rate is low, and the teletypewriter is not highly accurate itself.

How would you extend this program to check for the two stop bits? They must both be one or a framing error has occurred.

You can extend this program to check odd parity by replacing the LD A,D instruction with the sequence:

```
SUB    A
AND    D        ;IS PARITY ODD?
JP     PE.PRERR ;NO, PARITY ERROR HAS OCCURRED
```
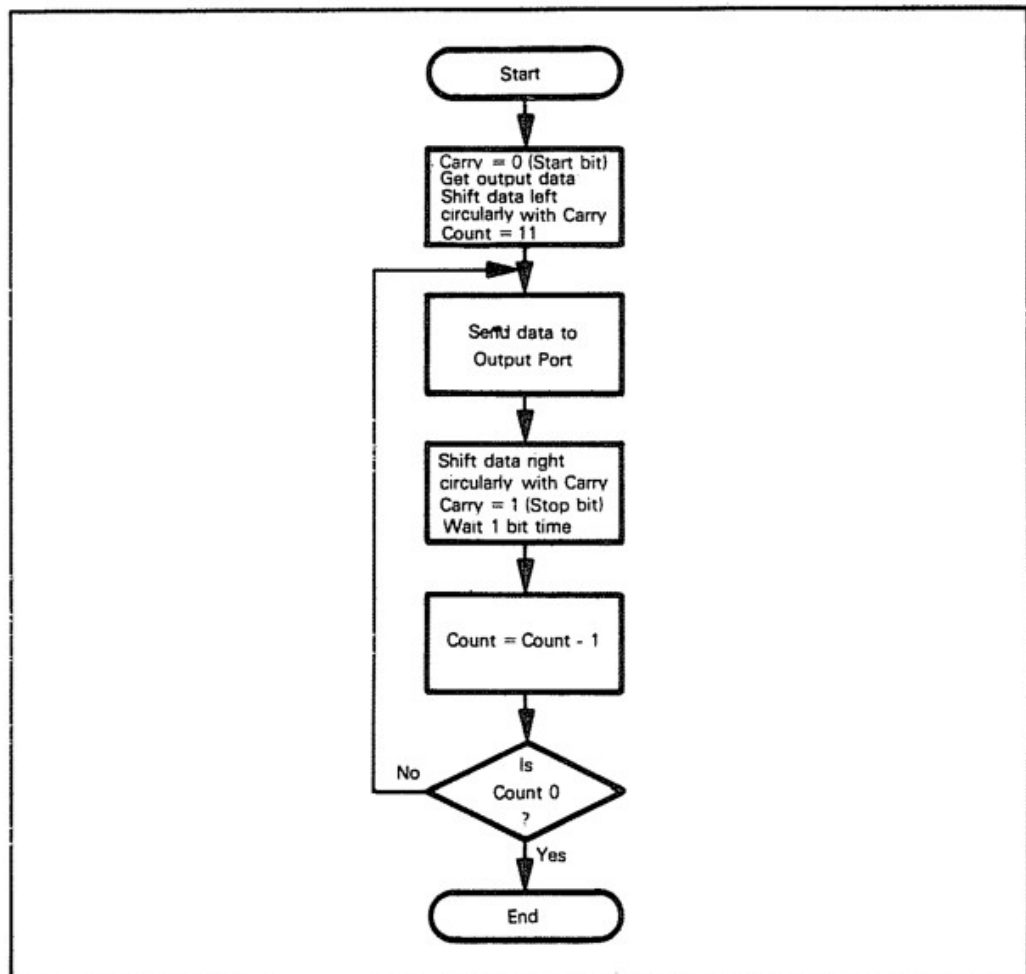
Figure 11-30. Flowchart for Transmit Procedure

Task 2: Write data.

Purpose: Transmit data to a teletypewriter through bit 0 of a PIO data register. The data is in memory location 0060.

Transmit (flowcharted in Figure 11-30)

```
TTY
TRANSMIT
MODE
```

Step 1) Transmit a Start bit (i.e., a logic zero).
Step 2) Transmit the seven data bits, starting with the least significant bit.
Step 3) Generate and transmit the Parity bit.
Step 4) Transmit two Stop bits (i.e., logic ones).

The transmission routine must wait one bit time between each operation.

11-86

Source Program: (Assume that parity need not be generated)

```
;
; MAKE PIO INTO OUTPUT PORT
;
        LD      A,00001111B   :MAKE PORT B OUTPUT
        OUT     (PIOCRB),A
;
; GET DATA AND CLEAR START BIT
;
        LD      A,(60H)       :GET DATA
        ADD     A,A           :SHIFT LEFT AND FORM START BIT
        LD      B,11          :COUNT = 11 BITS
;
; TRANSMIT A BIT AND UPDATE DATA
;
TBIT:   OUT     (PIODRB),A    ;TRANSMIT A BIT
        RRA                   :UPDATE FOR NEXT BIT
        SCF                   :FORM STOP BIT (LOGIC ONE)
;
; DELAY 9.1 MS AND COUNT BITS
;
        CALL    DFULL         :DELAY 9.1 MS
        DJNZ    TBIT          :COUNT DOWN 11 BITS
        HALT
```

The DFULL subroutine is the same as before. Remember that bit 0 of the data must be transferred first.

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	3E		LD	A,00001111B
0001	0F			
0002	D3		OUT	(PIOCRB),A
0003	PIOCRB			
0004	3A		LD	A,(60H)
0005	60			
0006	00			
0007	87		ADD	A,A
0008	06		LD	B,11
0009	0B			
000A	D3	TBIT:	OUT	(PIODRB),A
000B	PIODRB			
000C	1F		RRA	
000D	37		SCF	
000E	CD		CALL	DFULL
000F	35			
0010	00			
0011	10		DJNZ	TBIT
0012	F7			
0013	76		HALT	

ADD A.A clears the least significant bit so that it can be used as the start bit. The most significant bit is saved in the Carry. In actual applications, the startup routine should place a logic '1' on the teletypewriter line after configuration since that line should normally be in the mark (one) state.

Each character consists of 11 bits, starting with a start bit (zero) and ending with two stop bits (ones).

This program can easily be extended to generate 7-bit characters with odd parity in the most significant bit. The parity generation routine (to be inserted after LD A,(60H)) is:

```
         ANA    A          ;IS PARITY ODD?
         JP     PO,STBIT   ;YES, NO PROBLEM
         SET    7,A        ;NO, MAKE IT ODD BY SETTING MSB
STBIT:   ADD    A,A        ;SHIFT LEFT AND FORM START BIT
```

How would you generate even parity?

These procedures are sufficiently common and complex to merit a special LSI device: the UART, or Universal Asynchronous Receiver/Transmitter.[10] The UART will perform the reception procedure and provide the data in parallel form and a Data Ready signal. It will also accept data in parallel form, perform the transmission procedure, and provide a Peripheral Ready signal when it can handle more data. UARTs may have many other features, including:

<div style="border:1px solid;display:inline-block">UART</div>

1) Ability to handle various bit lengths (usually 5 to 8), parity options, and numbers of Stop bits (usually 1, 1-1/2, and 2).

2) Indicators for framing errors, parity errors, and "overrun errors" (failure to read a character before another one is received).

3) RS-232[11] compatibility; i.e., a Request-to-Send (RTS) output signal that indicates the presence of data to communications equipment and a Clear-to-Send (CTS) input signal that indicates, in response to RTS, the readiness of the communications equipment. There may be provisions for other RS-232 signals, such as Received Signal Quality, Data Set Ready, or Data Terminal Ready.

4) Tristate outputs and control compatibility with a microprocessor.

5) Clock options that allow the UART to sample incoming data several times in order to detect false Start bits and other errors.

6) Interrupt facilities and controls.

UARTs act as four parallel ports: an input data port, an output data port, an input status port, and an output control port. The status bits include error indicators as well as Ready flags. The control bits select various options. UARTs are inexpensive ($5 to $50, depending on features) and easy to use.

THE Z80 SERIAL INPUT/OUTPUT DEVICE (SIO)

The Z80 Serial Input/Output Device or SIO (see Figure 11-31) is a complete communications controller specifically designed for use in Z80-based microcomputers. It can serve a variety of communications functions, but we will only discuss its use as a simple asynchronous receiver/transmitter.[12]

The SIO has two complete channels (A and B) which can both receive and transmit serial data (see Figure 11-32). Channels that can receive and transmit simultaneously are called full-duplex.

FULL-DUPLEX

Alternatives include half-duplex (able to transmit and receive, but not at the same time), receive-only, and transmit-only.

An SIO occupies four input port addresses and four output port addresses. The B/$\overline{A}$ (Channel B or A Select) and C/$\overline{D}$ (Control or Data Select) lines choose one of the four ports as described in Ta-

SIO ADDRESSES

ble 11-7. Most often, designers attach address bit A_0 to the B/$\overline{A}$ input and address bit A_1 to the C/$\overline{D}$ input. The SIO then occupies four consecutive port addresses as described in the last column of Table 11-7.

As with the PIO, SIOs have more control registers than addresses. In fact, each SIO has eight registers in each channel for control and three registers for status. Figure 11-33 contains diagrams of each control or Write register; Figure

ADDRESSING SIO READ AND WRITE REGISTER

11-34 contains diagrams of each status or Read register. Two transfers are required to read or write any of the registers except Write Register 0. The first transfer (written into Write Register 0) contains three bits that direct the next transfer to or from the selected register. Note, in Figure 11-33, that these three bits occupy the three least significant bit positions and that zeros in the other bit positions indicate a byte that has no function other than addressing.
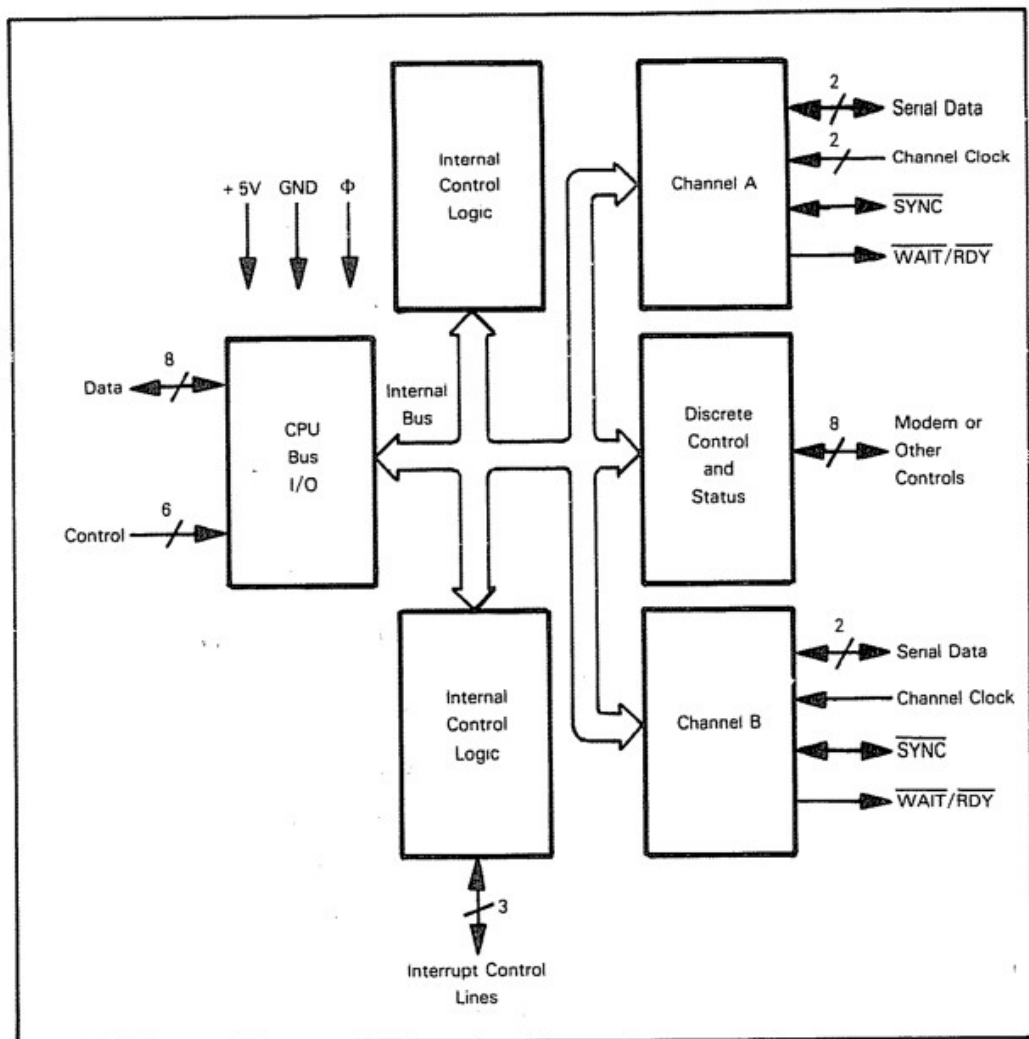
Figure 11-31. Block Diagram of the Z80 SIO

Table 11-7 SIO Addresses

CONTROL OR DATA SELECT	CHANNEL B OR A SELECT	REGISTER ADDRESSED	PORT ADDRESS (STARTING WITH SIOADD)
0	0	Data Register A	SIOADD
0	1	Data Register B	SIOADD+1
1	0	Control A	SIOADD+2
1	1	Control B	SIOADD+3
The port addresses assume that $C/\overline{D}$ is tied to A_1 and $B/\overline{A}$ to A_0			
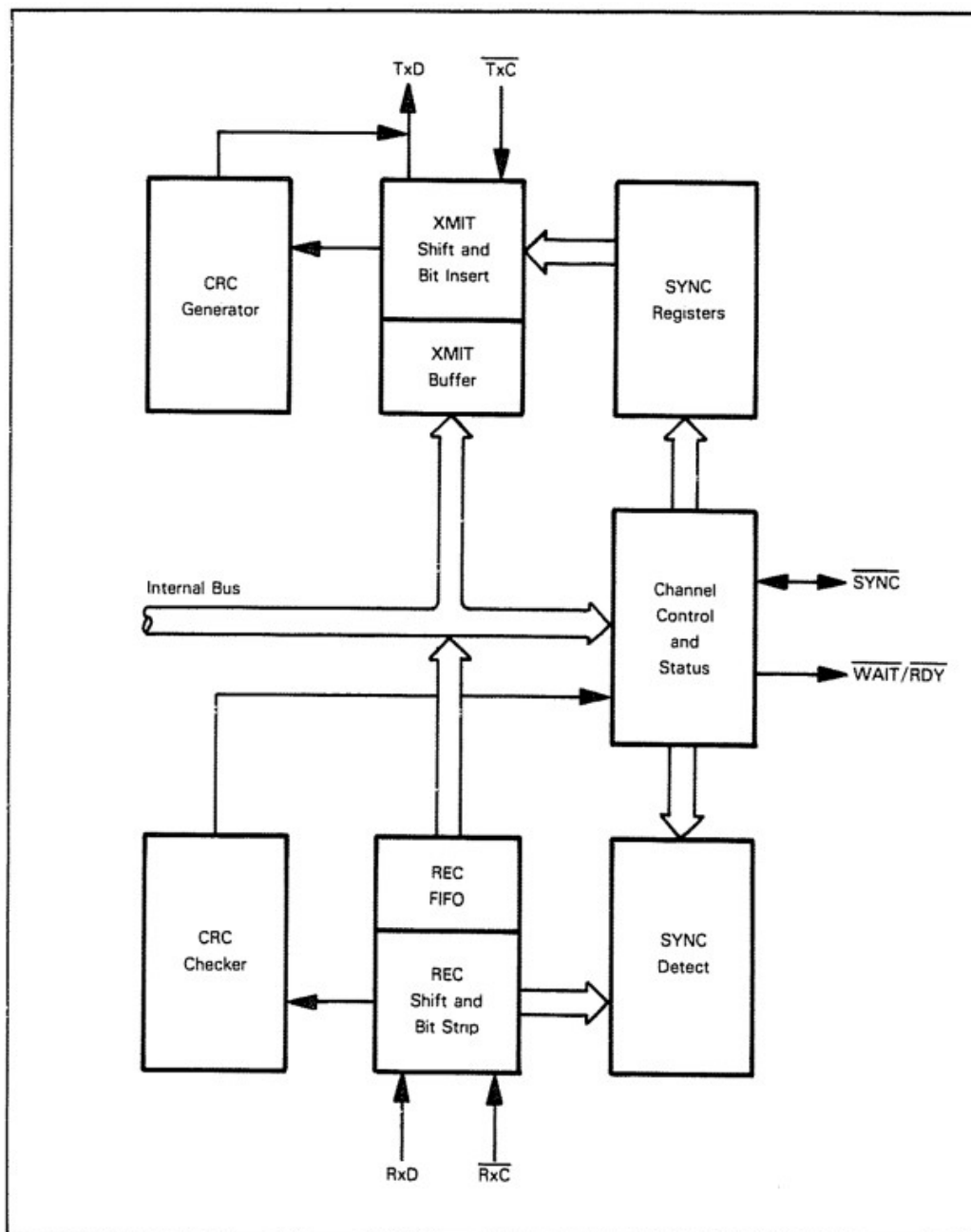
Figure 11-32. Block Diagram of SIO Channel

Write Registers

The Z80 SIO contains eight registers in each channel that are programmed (written into) by the system software to configure the functional personality of each channel. All Write registers, with the exception of Write Register 0, require two bytes to be properly programmed. The first byte contains three bits that point to the selected register (D0–D2); the second byte is the actual control word that is being written to that register to configure the SIO.

Write Register 0 is a special case. RESET (either internal command or external input) will initialize the SIO to Write Register 0. All basic commands (CMD2-CMD0) and CRC controls (CRC0, CRC1) can be accessed with a single byte using Write Register 0.

Contained in the first byte of any Write register access are the basic commands (CMD2-CMD0) and the CRC controls (CRC0,CRC1) so that maximum system control and flexibility is maintained.

Write Register 0

0	0	0	Register 0
0	0	1	Register 1
0	1	0	Register 2
0	1	1	Register 3
1	0	0	Register 4
1	0	1	Register 5
1	1	0	Register 6
1	1	1	Register 7

0	0	0	Null Code
0	0	1	Send Abort (SDLC)
0	1	0	Reset External or Status interrupts
0	1	1	Channel Reset
1	0	0	Reset Rx Interrupt on First Character
1	0	1	Reset Tx Interrupt Pending
1	1	0	Error Reset
1	1	1	Return from Interrupt (Ch-A Only)

0	0	Null Code
0	1	Reset Rx CRC Checker
1	0	Reset Tx CRC Generator
1	1	Reset CRC/SYNCS Sent/Sending Latch

Write Register 1

External Interrupt Enable
Tx Interrupt Enable
Status Affects Vector (Ch-B Only)

0	0	Rx Interrupt Disable
0	1	Rx Interrupt on First Character Only or Error
1	0	Interrupt on All Rx Characters (Parity Affects Vector)
1	1	Interrupt on All Rx Characters (Parity Does Not Affect Vector)

WAIT/READY on R/T
WAIT FN/READY FN
WAIT/READY Enable

Figure 11-33. SIO Control or Write Registers

Write Register 2

D7	D6	D5	D4	D3	D2	D1	D0

- V0 ⎫
- V1 ⎪
- V2 ⎪
- V3 ⎬ Interrupt
- V4 ⎪ Vector
- V5 ⎪
- V6 ⎪
- V7 ⎭

Write Register 3

D7	D6	D5	D4	D3	D2	D1	D0

- Rx Enable
- SYNC Character Load Inhibit
- Address Search Mode (SDLC)
- Rx CRC Enable
- Enter Hunt Mode
- Auto Enables

0	0	Rx 5 Bits/Character
0	1	Rx 6 Bits/Character
1	0	Rx 7 Bits/Character
1	1	Rx 8 Bits/Character

Write Register 4

D7	D6	D5	D4	D3	D2	D1	D0

- Parity Enable
- Parity Even/Odd

0	0	SYNC Modes Enable
0	1	1 Stop Bit/Character
1	0	1-1/2 Stop Bits/Character
1	1	2 Stop Bits/Character

0	0	8 Bit SYNC Character
0	1	16 Bit SYNC Character
1	0	SDLC Mode (01111110 SYNC Flag)
1	1	External SYNC Mode

0	0	x1 Clock Mode
0	1	x16 Clock Mode
1	0	x32 Clock Mode
1	1	x64 Clock Mode

Figure 11-33. SIO Control or Write Registers (Continued)

11-93

Write Register 5

D7	D6	D5	D4	D3	D2	D1	D0

- Tx CRC Enable
- RTS
- SDLC/CRC-16
- Tx Enable
- Send BREAK

0	0	Tx 5 Bits (or Less)/Character
0	1	Tx 7 Bits/Character
1	0	Tx 6 Bits/Character
1	1	Tx 8 Bits/Character

DTR

Write Register 6

D7	D6	D5	D4	D3	D2	D1	D0

- SYNC Bit 0
- SYNC Bit 1
- SYNC Bit 2
- SYNC Bit 3
- SYNC Bit 4
- SYNC Bit 5
- SYNC Bit 6
- SYNC Bit 7

Also SDLC Address Fiel

Write Register 7

D7	D6	D5	D4	D3	D2	D1	D0

- SYNC Bit 8
- SYNC Bit 9
- SYNC Bit 10
- SYNC Bit 11
- SYNC Bit 12
- SYNC Bit 13
- SYNC Bit 14
- SYNC Bit 15

For SDLC it must be programmed to "01111110" for Flag Recognition

Figure 11-33. SIO Control or Write Registers (Continued)

Read Registers

The Z80 SIO contains three registers that can be read to obtain the status of each channel. Status information includes error conditions, interrupt vector, and standard communication interface protocol signals. To read the contents of a selected Read register, the system software must first write out to the SIO the byte containing pointer information (D0-D2) in exactly the same manner as a Write register operation. Then. by issuing a READ operation, the contents of the addressed Read/Status register can be read by the Z80 CPU.

The real power in this type of command structure is that the programmer has complete freedom, after pointing to the selected register. of either reading or writing to initialize or test that register. By designing software to initialize the Z80 SIO in a modular, structured fashion. the programmer can use the powerful Z80 Block I/O instructions to significantly simplify and speed his software development and debug.

Read Register 0
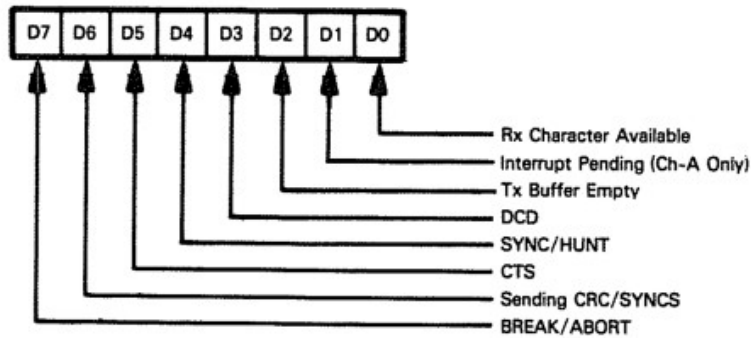
D7	D6	D5	D4	D3	D2	D1	D0

Rx Character Available
Interrupt Pending (Ch-A Only)
Tx Buffer Empty
DCD
SYNC/HUNT
CTS
Sending CRC/SYNCS
BREAK/ABORT

Read Register 1

D7	D6	D5	D4	D3	D2	D1	D0

All Sent

			I-Field Bits in Previous Byte	I-Field Bits in Second Previous Byte	
1	0	0	0	3	Residue Data for 8 Rx Bits/Character Programmed
0	1	0	0	4	
1	1	0	0	5	
0	0	1	0	6	
1	0	1	0	7	
0	1	1	0	8	
1	1	1	1	8	
0	0	0	2	8	

Parity Error
Rx Overrun Error   Special Rx
CRC/Framing Error   Condition
End of Frame (SDLC)   Interrupts

Figure 11-34. SIO Status or Read Registers

11-95
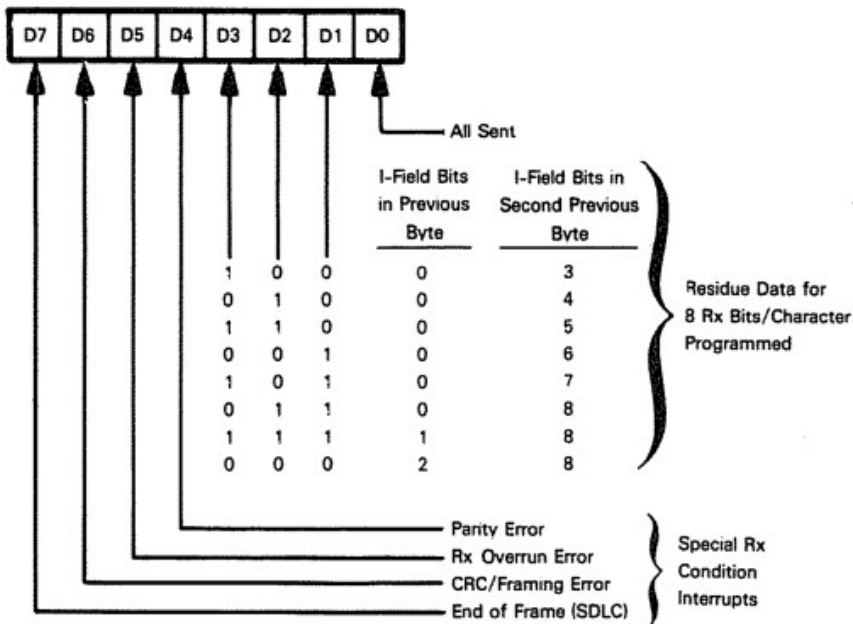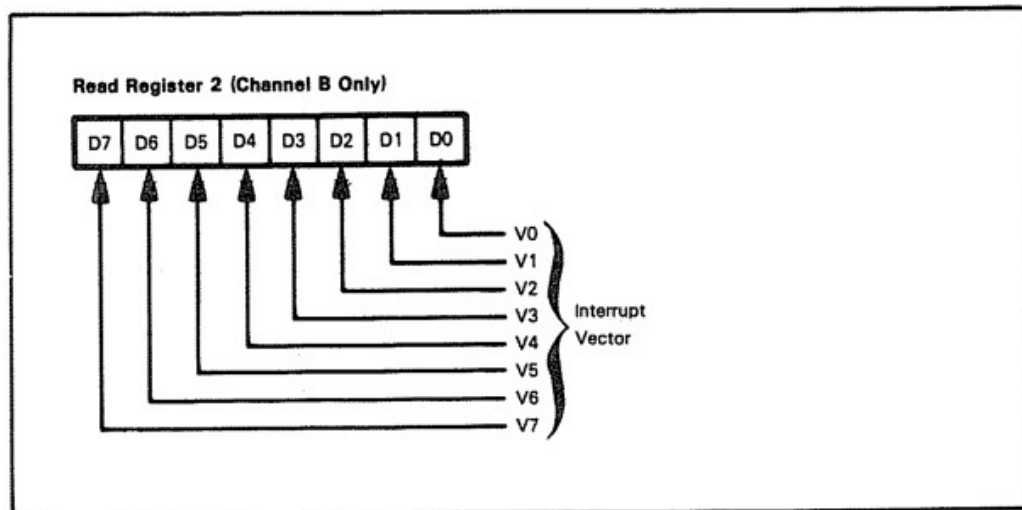
Figure 11-34. SIO Status or Read Registers (Continued)

Note the following special features of the SIO:

1) Input and output instructions address physically distinct registers. There is no way to read the control registers or write into the status registers.

2) All control registers for a channel share a single port address. Thus two bytes are required to change the contents of any control register except Register 0.

3) $\overline{\text{RESET}}$ initializes the SIO to Write Register 0. It also disables both receivers and transmitters, deactivates all control signals, and disables all interrupts. We will discuss the SIO interrupt system in Chapter 12.

4) The SIO must be configured before it can be used. The easiest way to do this is by placing the required bytes into a table and using the repeated Block I/O instruction. The table must include both the bytes needed to address the various registers and the data that must be placed into them. A typical routine would be:

```
LD      B.LENG      :NUMBER OF WORDS IN TABLE
LD      C.SIOCRA    :PORT NUMBER
LD      HL.CTLTAB   :START OF CONTROL TABLE
OTIR                :CONFIGURE SIO
```

5) The RS-232 signals are all active-low. However, the SIO control bits for these signals are active-high (i.e., a logic '1' in a control bit sends an RS-232 signal low).

6) The SIO requires an external clock. In asynchronous communications at 110 Baud, 1760 Hz is usually supplied and the X16 mode is used. The SIO will sample the bits at the clock frequency for synchronization and to avoid false start bits caused by noise on the line.

7) The Data Ready (Rx Character Available) flag is bit 0 of Read Register 0. The Peripheral Ready (Tx Buffer Empty) flag is bit 2 of Read Register 0.

8) Error status bits (parity, overrun, and framing) are in Read Register 1.

EXAMPLES

Teletypewriter I/O via a USART

Task 1: Read from teletypewriter through SIO

Purpose: Receive data from a teletypewriter through an SIO and place the data into memory location 0040. The data is 7-bit ASCII with odd parity.

Source Program:

```
        LD      A,4             ;ACCESS WRITE REGISTER 4
        OUT     (SIOCRA),A
        LD      A,01000001B     ;X16 CLOCK MODE, ODD PARITY
        OUT     (SIOCRA),A
        LD      A,3             ;ACCESS WRITE REGISTER 3
        OUT     (SIOCRA),A
        LD      A,01000001B     ;7 BIT CHARACTERS, ENABLE RECEIVER
        OUT     (SIOCRA),A
        SUB     A               ;ACCESS READ REGISTER 0
        OUT     (SIOCRA),A
WAITD:  IN      A,(SIOCRA)      ;GET STATUS
        RRA                     ;IS DATA AVAILABLE?
        JR      NC,WAITD        ;NO, WAIT
        IN      A,(SIODRA)      ;YES, GET DATA
        LD      (40H),A         ;SAVE DATA IN MEMORY
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
0000	3E	LD	A,4
0001	04		
0002	D3	OUT	(SIOCRA),A
0003	SIOCRA		
0004	3E	LD	A,01000001B
0005	41		
0006	D3	OUT	(SIOCRA),A
0007	SIOCRA		
0008	3E	LD	A,3
0009	03		
000A	D3	OUT	(SIOCRA),A
000B	SIOCRA		
000C	3E	LD	A,01000001B
000D	41		
000E	D3	OUT	(SIOCRA),A
000F	SIOCRA		
0010	97	SUB	A
0011	D3	OUT	(SIOCRA),A
0012	SIOCRA		
0013	DB	WAITD: IN	A,(SIOCRA)
0014	SIOCRA		
0015	1F	RRA	
0016	30	JR	NC,WAITD
0017	FB		
0018	DB	IN	A,(SIODRA)
0019	SIODRA		
001A	32	LD	(40H),A
001B	40		
001C	00		
001D	76	HALT	

The program establishes Write Register 4 as follows:

> Bits 7 and 6 = 01 to select X16 clock mode (1760 Hz must be supplied)
>
> Bit 1 = 0 to select odd parity
>
> Bit 0 = 1 to enable parity checking

The program establishes Write Register 3 as follows:

> Bits 7 and 6 = 01 for 7 bits per character
>
> Bit 0 = 1 to enable the receiver

The received data status bit is bit 0 of Read Register 0.

Note that any errors found will be reported in Read Register 1:

> Bit 6 = 1 for a framing error (no stop bit)
>
> Bit 5 = 1 for an overrun error (more data received before previous data read)
>
> Bit 4 = 1 for a parity error

Try adding an error checking routine to the program. Set

> (0061) = 0 if no errors occurred
>   = 1 if a parity error occurred
>   = 2 if an overrun error occurred
>   = 3 if a framing error occurred.

Note that the receiver always checks for one stop bit.

Task 2: Write to teletype through SIO.

Purpose: Send data from memory location 0040 to a teletypewriter through an SIO. The data is 7-bit ASCII with odd parity.

Source Program:

```
        LD      A,4             ;ACCESS WRITE REGISTER 4
        OUT     (SIOCRA),A
        LD      A,01001101B     ;X16 CLOCK MODE, 2 STOP BITS, ODD PARITY
        OUT     (SIOCRA),A
        LD      A,5             ;ACCESS WRITE REGISTER 5
        OUT     (SIOCRA),A
        LD      A,00101000B     ;7 BIT CHARACTERS, ENABLE TRANSMITTER
        OUT     (SIOCRA),A
        SUB     A               ;ACCESS READ REGISTER 0
        OUT     (SIOCRA),A
WAITR:  IN      A,(SIOCRA)      ;GET STATUS
        BIT     2,A             ;IS TRANSMITTER READY?
        JR      Z,WAITR         ;NO, WAIT
        LD      A,(40H)         ;YES, GET DATA
        OUT     (SIODRA),A      ;AND TRANSMIT IT
        HALT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
0000	3E		LD	A,4
0001	04			
0002	D3		OUT	(SIOCRA),A
0003	SIOCRA			
0004	3E		LD	A,01001101B
0005	4D			
0006	D3		OUT	(SIOCRA),A
0007	SIOCRA			
0008	3E		LD	A,5
0009	05			
000A	D3		OUT	(SIOCRA),A
000B	SIOCRA			
000C	3E		LD	A,00101000B
000D	28			
000E	D3		OUT	(SIOCRA),A
000F	SIOCRA			
0010	97		SUB	A
0011	D3		OUT	(SIOCRA),A
0012	SIOCRA			
0013	DB	WAITR:	IN	A,(SIOCRA)
0014	SIOCRA			
0015	CB		BIT	2,A
0016	57			
0017	28		JR	Z,WAITR
0018	FA			
0019	3A		LD	A,(40H)
001A	40			
001B	00			
001C	D3		OUT	(SIODRA),A
001D	SIODRA			
001E	76		HALT	

The program establishes Write Register 4 as follows:

      Bits 7 and 6 = 01 to select X16 clock mode (1760 Hz must be supplied)

      Bits 3 and 2 = 11 to add 2 stop bits to each character

      Bit 1 = 0 to select odd parity

      Bit 0 = 1 to enable parity generation

The program establishes Write Register 5 as follows:

      Bits 6 and 5 = 01 for 7 bits per character

      Bit 3 = 1 to enable the transmitter

The transmitter status bit is bit 2 of Read Register 1.

STANDARD INTERFACES

Other standard interfaces besides the TTY current-loop and RS-232 can also be used to connect peripherals to the microcomputer. Popular ones include:

1) The serial RS449, RS422, and RS423 interfaces.[13]

2) The 8-bit parallel General Purpose Interface Bus, also known as IEEE-488 or Hewlett-Packard Interface Bus (HPIB).[14]

3) The S-100 or Altair/Imsai hobbyist bus.[15] This is also an 8-bit bus.

4) The Intel Multibus.[16] This is another 8-bit bus that can, however, be expanded to handle 16 bits in parallel.

PROBLEMS

6) Separating Closures from an Unencoded Keyboard

Purpose: The program should read entries from an unencoded 3 x 3 keyboard and place them into an array. The number of entries required is in memory location 0040 and the array starts in memory location 0041.

Separate one closure from the next by waiting for the current closure to end. Remember to debounce the keyboard (this can be simply a 1 ms wait).

Sample Problem:

$$(0040) = 04$$

Entries are 7, 2, 2, 4

Result: (0041) = 07
(0042) = 02
(0043) = 02
(0044) = 04

7) Read a Sentence from an Encoded Keyboard

Purpose: The program should read entries from an ASCII keyboard (7 bits with a zero Parity bit) and place them into an array until it receives an ASCII period (hex 2E). The array starts in memory location 0040. Each entry is marked by a strobe as in the example given under An Encoded Keyboard.

Sample Problem:

Entries are H, E, L, L, O, .

Result: (0040) = 48   H
(0041) = 45   E
(0042) = 4C   L
(0043) = 4C   L
(0044) = 4F   O
(0045) = 2E   .

8) A Variable Amplitude Square Wave Generator

Purpose: The program should generate a square wave, as shown in the next figure, using a D/A converter. Memory location 0040 contains the scaled amplitude of the wave, memory location 0041 the length of a half cycle in milliseconds, and memory location 0042 the number of cycles.

Assume that a digital output of 80_{16} to the converter results in an analog output of zero volts. In general, a digital output of D results in an analog output of $V_{OUT} = -V_{REF}$ $(D-80)/80$ volts.

Sample Problem:

$$(0040) = A0 \text{ (hex)}$$
$$(0041) = 04$$
$$(0042) = 03$$

Result:
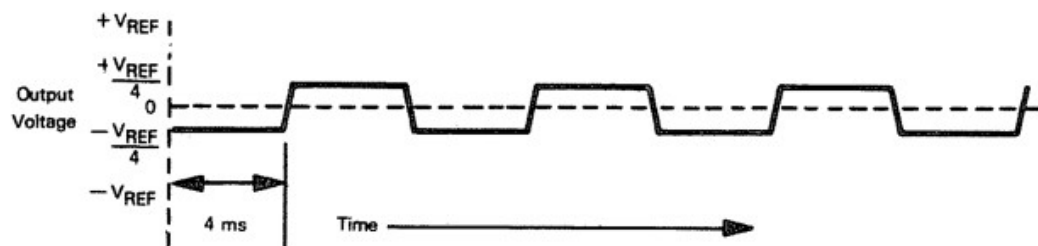
The base voltage is $80_{16} = 0$ volts.
Full scale is $100_{16} = -V_{REF}$ volts.
So $A0_{16} = (A0-80)/80 \times -V_{REF} = -V_{REF}/4$

The program produces 3 pulses of amplitude $V_{REF}/4$ with a half cycle length of 4 ms.

9) Averaging Analog Readings

Purpose: The program should take four readings from an A/D converter ten milliseconds apart and place the average in memory location 0040. Assume that the A/D conversion time can be ignored.

Sample Problem:

Readings are (hex) 86, 89, 81, 84

Result: (0040) = 85

10) A 30 Character-per-Second Terminal

Purpose: Modify the transmit and receive routines of the example given under A Teletypewriter to handle a 30 cps terminal that transfers ASCII data with one stop bit and even parity. How could you write the routines to handle either terminal depending on a flag bit in memory location 0060; e.g., (0060) = 0 for the 30 cps terminal, (0060) = 1 for the 10 cps terminal?

REFERENCES

1. Barnes, J., and V. Gregory, "Use Microcomputers to Enhance Performance with Noisy Data," EDN, August 20, 1976, pp. 71-72.

2. Swanson, R., "Understanding Cyclic Redundancy Codes," Computer Design, November 1975, pp. 93-99; and McNamara, J. E., Technical Aspects of Data Communication, Digital Equipment Corp., Maynard, Mass. 1977.

3. For example, the Z80 Direct Memory Access Controller (or DMA) for Z80 based microcomputers is described in An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors.

4. The TTL Data Book for Design Engineers, Texas Instruments Inc., P. O. Box 5012, Dallas, Texas 75222, 1976.

5. Dilatush, E., "Special Report: Numeric and Alphanumeric Displays," EDN, February 5, 1978, pp. 26-35.

6. See Reference 4.

7 Hnatek, E. R., A User's Handbook of D/A and A/D Converters, Wiley, New York, 1976.

8. See Reference 7.

9. See also D. Guzeman, "Marry Your μP to Monolithic A/Ds," Electronic Design, January 18, 1977, pp. 82-86.

10. For a discussion of UARTs, see P Rony et al., "The Bugbook IIa," E and L Instruments Inc., 61 First Street, Derby, CT. 06418; or D. G. Larsen et al., "INWAS: Interfacing with Asynchronous Serial Mode," IEEE Transactions on Industrial Electronics and Control Instrumentation, February 1977, pp. 2-12. Also see McNamara, Reference 2.

11. The official RS-232 standard is available as "Interface between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange," EIA RS-232C August, 1969. You can find introductory descriptions of RS-232 in G. Pickles, "Who's Afraid of RS-232?," Kilobaud, May 1977, pp. 50-54 and in C. A. Ogdin, "Microcomputer Buses — Part II," Mini-Micro Systems, July 1978, pp. 76-80. Ogdin also describes the newer RS-449 standard.

12. The SIO is discussed more completely in Volume 3 of An Introduction to Microcomputers; the following reference describes its use as a data link controller: Weissberger, A. J., "Data-Link Control Chips; Bringing Order to New Protocols," Electronics, June 8, 1978, pp. 104-112.

13. Electronic Industries Association, "Electrical Characteristics of Balanced Voltage Digital Interface Circuits," EIA RS-422, April 1975.

    Electronic Industries Association, "Electrical Characteristics of Unbalanced Voltage Digital Interface Circuits," EIA RS-423, April 1975.

    Electronic Industries Association, "General Purpose 37-Position and 9-Position Interface for Data Terminal Equipment and Data Circuit Terminating Equipment Employing Serial Binary Data Interchange," EIA RS-449, November 1977.

    Morris, D., "Revised Data Interface Standards," Electronic Design, September 1, 1977, pp. 138-141.

14. Institute of Electrical and Electronics Engineers, "IEEE Standard Digital Interface for Programmable Instrumentation." IEEE Std 488-1975-ANSI MC 1.1-1975.

    J. B. Peatman, Microcomputer-Based Design, McGraw-Hill, New York, 1977; Loughry, D. C. and M. S. Allen, "IEEE Standard 488 and Microprocessor Synergism," Proceedings of the IEEE, February 1978, pp. 162-172.

15. Morrow, G., and H. Fullmer, "Proposed Standard for the S-100 Bus," Computer, May 1978, pp. 84-89.

    Smith, M. L., "Build Your Own Interface," Kilobaud, June 1977, pp. 22-28.

16. Rolander, T., "Intel Multibus Interfacing," Intel Application Note AP-28, Intel Corporation, Santa Clara, CA., 1977.

Chapter 12
INTERRUPTS

Interrupts are inputs that the CPU examines as part of each instruction cycle. These inputs allow the CPU to react to asynchronous events in a more efficient manner than polling each device. When interrupts are utilized to initiate I/O, generally more hardware than ordinary, programmed I/O is required, but this provides a faster and more direct response.[1]

Why use interrupts? Interrupts allow events such as alarms, power failure, the passage of a certain amount of time, and peripherals having data or being ready to accept data to get the immediate attention of the CPU. The programmer does not need to poll every

| REASONING |
| BEHIND |
| INTERRUPTS |

device, nor need the programmer worry about the system completely missing events. An interrupt system is like the bell on a telephone — it rings when a call is received so that you don't have to pick up the receiver occasionally to see if someone is on the line. The CPU can go about its normal business (and get a lot more done). When something happens, the interrupt rouses the CPU and forces it to service the input before resuming normal operations. Of course, this simple description becomes more complicated (just like a telephone switchboard) when there are many interrupts of varying importance and there are tasks that cannot be interrupted.

The implementation of interrupt systems varies greatly. Among the questions that must be answered to characterize a particular system are:

| CHARACTERISTICS |
| OF INTERRUPT |
| SYSTEMS |

1) How many interrupt inputs are there?
2) How does the CPU respond to an interrupt?
3) How does the CPU determine the source of an interrupt if the number of sources exceeds the number of inputs?
4) Can the CPU differentiate between important and unimportant interrupts?
5) How and when is the interrupt system enabled and disabled?

There are many different answers to these questions. The aim of all the implementations, however, is to have the CPU respond rapidly to interrupts and resume normal activity afterwards.

The number of interrupt inputs on the CPU chip determines the number of different responses that the CPU can produce without any additional hardware or software. Each input can produce a different internal response. Unfortunately, most microprocessors have a very small number (one or two, typically) of separate interrupt inputs.

The ultimate response of the CPU to an interrupt must be to transfer control to the correct interrupt service routine and to save the current value of the Program Counter. The CPU must therefore execute a Jump-to-Subroutine or Call instruction with the beginning of the interrupt service routine as its address. This action will save the return address in the Stack and transfer control to the interrupt service routine. The amount of external hardware required to produce this response varies greatly. Some CPUs internally generate the instruction and the address; others require external hardware to form them. The CPU can only generate a different instruction or address for each separate input.

If the number of interrupting devices exceeds the number of inputs, the CPU will need extra hardware or software to identify the source of the interrupt. In the simplest case, the software can be a polling routine which checks the status of the devices that may be interrupting. The only advantage of such a system over normal polling is that the CPU knows that at least one device is active. The alternative solution is for additional hardware to provide a unique data input (or "vector") for each source. The two alternatives can be mixed; the vectors can identify groups of inputs from which the CPU can identify a particular one by polling.

An interrupt system that can differentiate between important and unimportant interrupts is called a "priority interrupt system." Internal hardware can provide as many priority levels as there are inputs. External hardware can provide additional levels through the use of a Priority register and comparator. The external hardware does not allow the interrupt to reach the CPU unless its priority is higher than the contents of the Priority register. A priority interrupt system may need a special way to handle low-priority interrupts that may be ignored for long periods of time.

Most interrupt systems can be enabled or disabled. In fact, most CPUs automatically disable interrupts when a RESET is performed (so that the programmer can configure the interrupt system) and on accepting an interrupt (so that the interrupt will not interrupt its own service routine). The programmer may wish to disable interrupts while preparing or processing data, performing a timing loop, or executing a multi-byte operation.

An interrupt that cannot be disabled (sometimes called a "non-maskable interrupt") may be useful to warn of power failure, an event that obviously must take precedence over all other activities.

The advantages of interrupts are obvious, but there are also disadvantages. These include:

1) Interrupt systems may require a large amount of extra hardware.

2) Interrupts still require data transfers under program control through the CPU. There is no speed advantage as there is with DMA.

3) Interrupts are random inputs, which makes debugging and testing difficult. Errors may occur sporadically, and therefore may be very hard to find.[2]

4) Interrupts may involve a large amount of overhead if many registers must be saved and the source must be determined by polling.

Z80 INTERRUPT SYSTEM

The Z80's internal response to an interrupt is fairly complex, since there are three different operating modes. The interrupt system consists of:

1) An active-low maskable interrupt input ($\overline{INT}$) and an active-low non-maskable interrupt input ($\overline{NMI}$).

2) Two enable flip-flops (IFF1 and IFF2). IFF1 can be set or reset to enable or disable interrupts. IFF2 serves as temporary storage for IFF1 during non-maskable interrupts.

12-2

The Z80 checks the current status of the interrupt system at the end of each instruction cycle. If an interrupt is active and enabled, the response is as follows:[3]

1) The CPU disables the interrupt system by clearing IFF1. IFF2, however, is left in its original state if a non-maskable interrupt has occurred. Note that RESET clears both interrupt flip-flops so that the system can be configured before interrupts are enabled.

2) The CPU executes a special Interrupt Acknowledge cycle, distinguished by the M1 signal (operation code fetch) being active. MREQ (memory request) inactive (so the CPU will not perform its normal memory access), and IORQ (input/output request) active so that an interrupt response vector can be placed on the Data Bus.

The remainder of the response depends on the interrupt mode and the source.

Note in particular that the Z80 will check for interrupts after each transfer or comparison in a Block Move, Block Compare, or Repeated Block I/O instruction.

The Z80 has the following special instructions for use with the interrupt system:

1) EI (Enable Interrupts) enables the maskable interrupt by setting the interrupt flip-flops.

2) DI (Disable Interrupts) disables the maskable interrupt by clearing the interrupt flip-flops.

3) RST (Restart) is a one-word Call instruction that saves the current value of the Program Counter in the Stack and jumps to the address specified in the instruction. Table 12-1 contains the various Restart instructions and their destination addresses. RST is often used in interrupt systems because it is a one-word instruction that is easy to form and place on the Data Bus.

4) RETI (Return from Interrupt) acts exactly like a normal Return (RET) instruction except that Z80 peripheral chips (PIOs, SIOs, and CTCs) recognize this instruction and use it as a notification that the current interrupt service routine has been completed.

5) RETN (Return from Non-Maskable Interrupt) acts exactly like a normal Return (RET) instruction except that it loads IFF1 from IFF2 so as to restore the original state of the interrupt system.

6) LD A,I loads the Accumulator with the contents of the I (Interrupt Vector) register. This instruction (and LD A,R) also places IFF2 into the P/O bit of the Flag register. That flag can then be tested or saved in the Stack.

7) LD I,A loads the I (Interrupt Vector) register with the contents of the Accumulator.

8) IM (Set Interrupt Mode) determines the mode in which interrupts are serviced. The three options are 0, 1, or 2; these are described later in this chapter.

Non-Maskable Interrupt

The non-maskable interrupt is an edge-sensitive (negative edge triggered) input. The processor therefore reacts only to the edge of a pulse on this line, and the pulse will not interrupt its own service routine. Non-maskable interrupts are useful for

applications that must respond to loss of power (i.e., must save data in a low-power memory or switch to a backup battery). Typical applications are communications equipment that must retain codes and partial messages and test equipment that must keep track of partially completed tests.

Table 12-1 The Restart (RST) Instructions

| RST Instruction | Operation Code | Destination Address | |
(Mnemonic)	Hex)	(Hex)	(Decimal)
RST 0	C7	0000	0
RST 8	CF	0008	08
RST 10H	D7	0010	16
RST 18H	DF	0018	24
RST 20H	E7	0020	32
RST 28H	EF	0028	40
RST 30H	F7	0030	48
RST 38H	FF	0038	56

The Z80 responds to a non-maskable interrupt as follows:

1) It clears IFF1, thus disabling all interrupts (but saving the old state of IFF1 in IFF2).

2) It ignores the next instruction fetched from memory and instead jumps to memory location 0066_{16}, saving the old value of the Program Counter in the Stack.

Remember that a RETN instruction at the end of the service routine will restore the old state of IFF1 from IFF2.

We will not discuss the non-maskable interrupt further. Henceforth, we will assume that all interrupt inputs are tied to $\overline{INT}$.

Z80 Interrupt Modes

The Z80 has three interrupt modes. The programmer can choose any of these modes with the appropriate IM instruction. On reset, the processor always enters Mode 0. The modes are:

INTERRUPT MODES

Mode 0

In this mode, the CPU uses the data input during the Interrupt Acknowledge cycle as an instruction. This mode is the same as the 8080 interrupt response mode.[4]

The normal data input that must be provided externally is a RST instruction (see Table 12-1).

RST is useful in interrupt systems for the following reasons:

RESTART INSTRUCTION

1) It is a one-word instruction and so requires only one fetch cycle.

2) It provides eight different destination addresses or vectors.

3) Its vectors are far enough apart to allow Jump instructions to reach the actual service routines.

4) It is easy to form, since five of the bits are always '1.' An 8-to-3 encoder can provide the other three bits quite easily.

RST has the following disadvantages:

1) It cannot provide more than eight vectors.

2) Its vectors are not far enough apart to allow space for entire interrupt service routines.

3) Its vectors are in a fixed area of memory.

4) RST 0 has the same destination address as the $\overline{RESET}$ input and is therefore very difficult to use. The system needs hardware to differentiate between $\overline{RESET}$ and RST 0, since the two cannot be distinguished by software alone.

Remember that RST saves the old Program Counter in the Stack just as CALL does.

Mode 1

In this mode, the CPU ignores the data input during the Interrupt Acknowledge cycle and always executes RST 38H, thus jumping to memory location 0038_{16} and saving the old Program Counter in the Stack. This mode is equivalent to Mode 0 if the data input is always RST 38H (FF_{16}).

The advantage of this mode is that no external hardware is required. Its disadvantages are that there is no way to directly differentiate among interrupt sources and the destination address is fixed. Mode 1 is useful in applications that have only one or two interrupt sources and in which minimum hardware cost is essential.

Mode 2

In this mode, the CPU uses the data input as part of an address from which to get the starting address of the interrupt service routine. When an interrupt is accepted, the CPU:

1) Disables further interrupts by clearing IFF1 and IFF2.

2) Stores the old Program Counter in the RAM Stack.

3) Forms a pointer from the contents of Register I (eight MSBs) and the Data Bus input during the Interrupt Acknowledge cycle (eight LSBs). The least significant bit of this pointer is forced to zero.

4) Fetches an address from the two memory locations starting with the one referred to by the pointer (see Figure 12-1).

5) Transfers control to the address obtained from memory.

Interrupt response in this mode requires 19 clock cycles.

The advantage of this mode is that it can provide a full page of 128 interrupt service vectors located anywhere in memory. The disadvantages of this approach are that the interrupt response is slower and the system must be initialized, as follows:

1) The table of vectors must be loaded into memory if it is not in ROM.

2) The I register must be loaded with the eight most significant bits (or page number) of the table address. Note that $\overline{RESET}$ clears Register I. You can load I with a value as follows:

```
        LD      A,IPGNO      ;GET INTERRUPT PAGE NUMBER
        LD      I,A          ;STORE IN VECTOR REGISTER
```

3) Interrupt Mode 2 must be set with the instruction IM 2.

Mode 2 is designed to work with Z80 PIOs, SIOs, and CTCs. PIO and SIO interrupts are described later in this chapter.

Z80/8080 INTERRUPT COMPATIBILITY

Mode 0 for the Z80 interrupt system is, as mentioned, identical to the 8080 interrupt response. The 8080 does not have Interrupt Modes 1 or 2, although Mode 1 is really just a special case of Mode 0. The 8080 also has no $\overline{NMI}$ input.

The 8085 has additional interrupt inputs, not available on either the 8080 or the Z80. The 8085 also has a non-maskable interrupt (called TRAP) that forces a call to a different address (24_{16}) than that used by the Z80 NMI input.
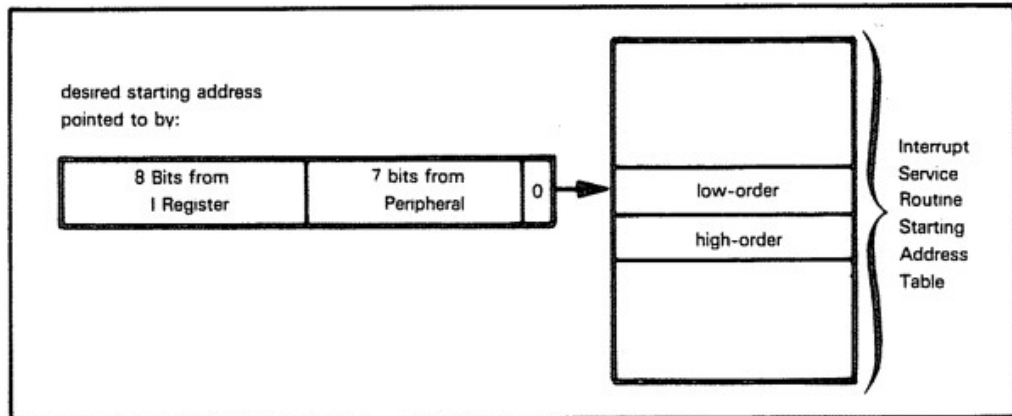
Figure 12-1. Forming an Interrupt Vector in Interrupt Mode 2

PIO INTERRUPTS

Most Z80 interrupt systems involve PIOs. Each port of the PIO has the following features for use with interrupts:

```
PIO
INTERRUPTS
```

1) An 8-bit Interrupt Vector register used to hold the eight least significant bits of the table address formed by the CPU in Interrupt Mode 2.
2) An interrupt enable bit.
3) An Interrupt Control register used to determine the logical operation performed and the active polarity monitored for generating interrupts in the control mode.
4) An Interrupt Mask register used to determine which data lines will be monitored to generate interrupts in the control mode.

The Interrupt Vector register in each port can be accessed by writing a control word with a zero in its least significant bit, as shown below (see also Table 11-2):

```
PIO
INTERRUPT
VECTOR
```


A typical sequence to establish the value in this register is:

```
LD      A,IVECT
OUT     (PIOCR),A
```

where IVECT has a '0' in its least significant bit. The starting address for the interrupt service routine is at address IVECT on the page assigned to the table of starting addresses for service routines.

12-6

D7	D6	D5	D4	D3	D2	D1	D0
Enable Interrupt	AND/ OR	High/ Low	Mask follows	0	1	1	1

used in Mode 3 only   signifies interrupt control word
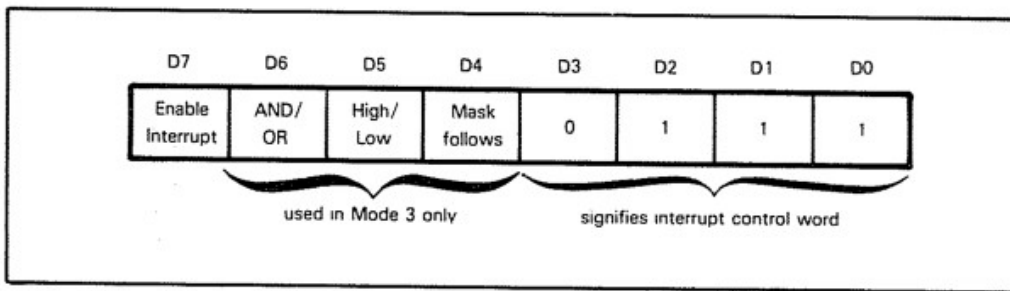
Figure 12-2. Format for a PIO Interrupt Control Word

D7	D6	D5	D4	D3	D2	D1	D0
MB7	MB6	MB5	MB4	MB3	MB2	MB1	MB0

Only those port lines whose mask bit is zero will be monitored for generating an interrupt.
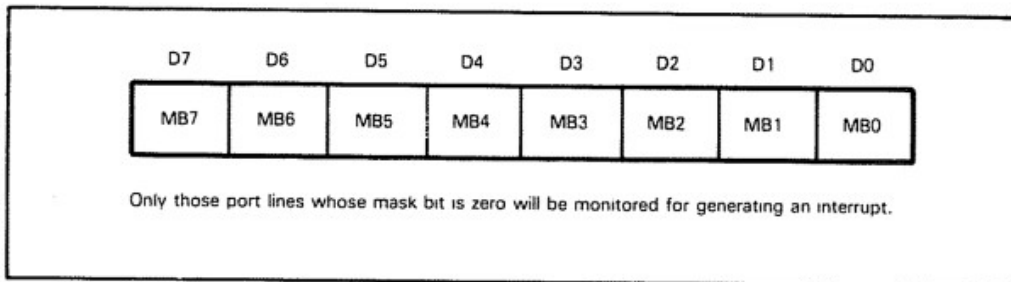
Figure 12-3. Format for a PIO Interrupt Mask

We can set the interrupt control word in each port by writing a control word with the format shown in Figure 12-2. If the port is in Mode 3, bits D6, D5, and D4 have the following meanings:

PIO INTERRUPT CONTROL MODE

1) D6 = 1 means that all monitored I/O lines must become active to cause an interrupt (i.e.. a logical AND), while D6 = 0 means that any monitored I/O line becoming active will cause an interrupt (i.e.. a logical OR).

Note that an interrupt occurs only if the logical equation is true when interrupts are enabled or if it changes from false to true while interrupts are enabled.

2) D5 defines the active polarity (high or low) of the monitored I/O lines. D5 = 1 means active high. D5 = 0 means active low.

3) D4 = 1 means that the next control word is an interrupt mask (Figure 12-3). Only lines with a mask bit of zero will be monitored. D4 = 0 means that the mask does not follow.

Bit 7 of the interrupt control word determines the value of the interrupt enable flip-flop for the port. Interrupts may be generated if the flip-flop is set. Power-on resets this flip-flop, but remember that the PIO has no RESET input. The interrupt ena-

ENABLING AND DISABLING PIO INTERRUPTS

ble flip-flop may be set or reset without affecting the rest of the interrupt control word by writing a control word with the flip-flop value in bit 7 and 0011 in the four least significant bits.

Setting bit 4 of the interrupt control word clears any pending interrupts. This can be used to clear interrupts that may have occurred inadvertently during a reset.

Examples

1) Interrupting output port with vector located at address 80_{16}. Remember that the page number is in the CPU I register.

```
LD    A.00001111B    ;MAKE PORT B OUTPUT
OUT   (PIOCRB).A
LD    A.80H          ;VECTOR ADDRESS = 80 HEX
OUT   (PIOCRA).A
LD    A.10000011B    ;ENABLE PIO INTERRUPT
OUT   (PIOCRB).A
```

An alternative that clears pending interrupts as well as enabling interrupts from the port is:

```
LD    A.10010111B    ;ENABLE PIO INTERRUPT
OUT   (PIOCRA).A
```

An interrupt will occur on the rising edge of $\overline{STB}$.

2) Interrupting input port with vector located at address 60_{16}.

```
LD    A.01001111B    ;MAKE PORT A INPUT
OUT   (PIOCRA).A
LD    A.60H          ;VECTOR ADDRESS = 60 HEX
OUT   (PIOCRA).A
LD    A.10000011B    ;ENABLE PIO INTERRUPT
OUT   (PIOCRA).A
```

An interrupt will occur on the rising edge of $\overline{STB}$.

3) Interrupting control port with vector located at address 48_{16}. An interrupt will be generated if data lines A_4 and A_7 both go low.

```
LD    A.11001111B    ;MAKE PORT A CONTROL
OUT   (PIOCRA).A
LD    A.10001000B    ;LINES 4.7 INPUTS - OTHERS OUTPUTS
OUT   (PIOCRA).A
LD    A.48H          ;VECTOR ADDRESS = 48 HEX
OUT   (PIOCRA).A
LD    A.11010111B    ;ENABLE PIO INTERRUPT
OUT   (PIOCRA).A
LD    A.01110111B    ;MONITOR LINES 4.7 ONLY
OUT   (PIOCRA).A
```

The interrupt control word has:

bit 7 = 1 to enable the interrupt
bit 6 = 1 to generate an interrupt only if all monitored lines are or become active (a logical AND)
bit 5 = 0 to specify that a logic '0' is the active state to be monitored
bit 4 = 1 to indicate that a mask word follows (and to reset pending interrupts)

4) Interrupting control port with vector located at address 28_{16}. An interrupt will be generated if any of the data lines go high.

```
LD      A,11001111B     :MAKE PORT B CONTROL
OUT     (PIOCRB),A
LD      A,0FFH          :ALL LINES INPUTS
OUT     (PIOCRB),A
LD      A,28H           :VECTOR ADDRESS = 28 HEX
OUT     (PIOCRB),A
LD      A,10110111B     :ENABLE INTERRUPTS
OUT     (PIOCRB),A
SUB     A               :MONITOR ALL LINES
OUT     (PIOCRB),A
```

The interrupt control word has:

bit 7 = 1 to enable the interrupt

bit 6 = 0 to generate an interrupt if any monitored lines become active (a logical OR)

bit 5 = 1 to specify that a logic '1' is the active state to be monitored

bit 4 = 1 to indicate that a mask word follows (and to reset pending interrupts).

Obviously a repeated Block Output instruction could be used to shorten these programs considerably.

Each PIO also has a single interrupt output and enable signals for daisy chaining. The $\overline{INT}$ output is active-low when the PIO has an interrupt request. The enable signals are:

DAISY CHAINING PIO INTERRUPTS

IEI (Interrupt Enable In) — high if no other devices of higher priority are being serviced by a CPU interrupt service routine.

IEO (Interrupt Enable Out) — high if IEI is high and the CPU is not servicing an interrupt from this PIO

IEI and IEO can be used to form a daisy chain (see Volume 1 of <u>An Introduction to Microcomputers</u>) in which PIOs and other devices that are connected to the chain closer to the CPU can block interrupt requests from devices further from the CPU. The advantages of the daisy chain are:

PIO DAISY CHAIN SIGNALS

1) It identifies each source uniquely.

2) It requires no other hardware.

3) It is easy to expand or rearrange in hardware.

The disadvantages of the daisy chain are:

ADVANTAGES AND DISADVANTAGES OF DAISY CHAIN INTERRUPTS

1) It can be varied or changed only in hardware.

2) It does not provide for eventual servicing of low priority interrupts.

3) It requires extra time because signals must ripple through the chain.

The Z80 automatically waits long enough for the signals to ripple through a chain of up to four devices when operating in Interrupt Mode 2. Additional hardware can be added to allow longer chains.

Note that a particular device in the chain operates as follows:

1) It places its interrupt vector on the bus during an Interrupt Acknowledge cycle only if it has a pending interrupt request and Interrupt Enable In is high (indicating no higher priority devices are being serviced). Interrupt Enable Out is also set low. Within a device, Port A interrupts take precedence over Port B interrupts.

2) It subsequently brings its Interrupt Enable Out high (enabling lower priority devices) only if a RETI instruction is executed while its Interrupt Enable In is high.

Thus, a particular device will be serviced only when it has the highest priority request and will block lower-priority requests until its service routine has been completed. A higher-priority device can interrupt a lower-priority service routine without any difficulty. Note that a RETI instruction at the end of the high priority routine will not be recognized by the lower-priority device.

SIO INTERRUPTS

The SIO can also serve as a source for interrupts. You should note the following features of the SIO interrupt-based systems:

1) The transmitter interrupt is enabled by setting bit 1 of Write Register 1 on each channel.

2) The interrupt vector is affected by bits 2, 3, and 4 of Write Register 1 according to Tables 12-2 and 12-3.

3) The interrupt vector is in Write Register 2 on Channel B only. It can be read from Read Register 2 on Channel B only.

4) Bit D1 of Read Register 0 on Channel A is 1 if any interrupt condition is present in the entire SIO.

Within an SIO, Channel A interrupts have priority over Channel B interrupts, receiver interrupts have priority over transmitter interrupts, and transmitter interrupts have priority over external or status interrupts.

SIOs can be used in a polling interrupt system. The CPU must check each SIO for activity by examining bit 1 of Read Register 9 on Channel A; i.e.,

```
SUB    A              ;ACCESS READ REGISTER 0
OUT    (SIOCRA),A
IN     A,(SIOCRA)     ;GET SIO STATUS
BIT    1,A            ;ANY INTERRUPTS PENDING?
JR     NZ,SERVE       ;YES, INTERRUPT ACTIVE
```

The important features of a Z80 polling system are:

1) The first interrupt examined has the highest priority, since the remaining interrupts will not be examined if the first one is active. The second interrupt has the next highest priority, and so on.

2) The service routine must clear the SIO interrupt by reading or writing the appropriate data register even if a data transfer is otherwise unnecessary.

Table 12-2. Further Vectoring of SIO Interrupts
(Bit 2 of SIO Write Register 1 on Channel B is 1)

Status Affects Vector (D2) (Channel B Only)

If this bit is 1, the vector returned from an interrupt acknowledge cycle will be variable according to the following:

	V3	V2	V1	
Ch B	0	0	0	Ch B Transmit Buffer Empty
	0	0	1	Ch B External/Status Change
	0	1	0	Ch B Receive Character Available
	0	1	1	Ch B Special Receive Condition*
Ch A	1	0	0	Ch A Transmit Buffer Empty
	1	0	1	Ch A External/Status Change
	1	1	0	Ch A Receive Character Available
	1	1	1	Ch A Special Receive Condition*

*Special Receive Conditions → Parity Error or Rx Overrun Error or CRC/Framing Error or End of Frame (SDLC)

If this bit is 0, the fixed vector programmed in the Interrupt Vector register is returned.

Table 12-3. SIO Interrupt Modes
(Bits 3 and 4 of Write Register 1)

Rec Int Mode 0 (D3), Rec Int Mode 1 (D4)

Receive Interrupt Mode 0 and Receive Interrupt Mode 1 together specify the various character available conditions:

Mode	D4 Rec Int Mode 1	D3 Rec Int Mode 0	
0	0	0	Receiver interrupts disabled
1	0	1	Receive interrupt on first character only error
2	1	0	Interrupt on all Receive Characters-Parity error affects Vector
3	1	1	Interrupt on all Receive Characters-Parity error does not affect Vector

INTERRUPT EXAMPLES

A Startup Interrupt

Purpose: The computer waits for a PIO interrupt to occur before starting actual operations.

Many systems remain inactive until the operator actually starts them or a DATA READY signal is received. On $\overline{RESET}$, such systems must initialize the Stack Pointer, enable the startup interrupt, and execute a HALT instruction. Remember that $\overline{RESET}$ disables the processor interrupt and power-on disables all PIO interrupts. In the flowchart, the decision as to whether startup is active is made in hardware (i.e., by the CPU examining the interrupt input internally) rather than in software.

Flowchart:
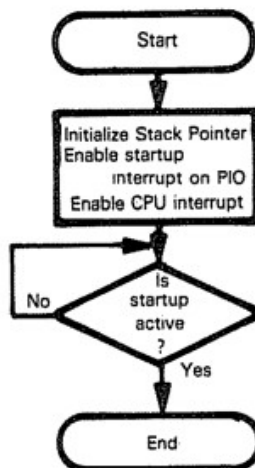
Source Program:

Main Program:

```
RESET   EQU     0
        ORG     RESET
        LD      SP.100H         ;PUT STACK AT END OF MEMORY
        LD      A,01001111B     ;PUT PIO IN INPUT MODE
        OUT     (PIOCRA),A
        LD      A,10000111B     ;ENABLE PIO INTERRUPT
        OUT     (PIOCRA),A
        EI                      ;ENABLE INTERRUPTS
        HALT                    ;AND WAIT
```

Interrupt Service Routine:

```
        ORG     INTRP
        LD      SP.100H         ;REINITIALIZE STACK POINTER
        JP      START           ;START MAIN PROGRAM
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
Main Program:			
0000	31	LD	SP.100H
0001	00		
0002	01		
0003	3E	LD	A,01001111B
0004	4F		
0005	D3	OUT	(PIOCRA),A
0006	PIOCRA		
0007	3E	LD	A.10000111B
0008	87		
0009	D3	OUT	(PIOCRA),A
000A	PIOCRA		
000B	FB	EI	
000C	76	HALT	
Interrupt Service Routine:			
INTRP	31	LD	SP,100H
INTRP+1	00		
INTRP+2	01		
INTRP+3	C3	JP	START
INTRP+4			
INTRP+5	START		

The main program must initialize the Stack Pointer, since the interrupt response always stores the old Program Counter in the Stack. Here the service routine simply reinitializes the Stack Pointer before the actual startup routine is executed. An alternative would be to increment the Stack Pointer twice before jumping to the startup routine. Remember that the Z80 comes up in Interrupt Mode 0. Any other mode would require the execution of an IM instruction.

The exact location of the interrupt service routine varies with the microcomputer. If your microcomputer has no monitor, you can start the interrupt service routine wherever the external hardware or vector table directs the CPU. Of course, you should place the routine so that it does not interfere with fixed addresses or with other programs.

INTERRUPTS ON PARTICULAR MICROCOMPUTERS

If your microcomputer has a monitor, the monitor will often occupy the RESET and interrupt service addresses. It will then supply service routines or the addresses of those routines. A typical monitor routine initialization would be:

INTERRUPT HANDLING BY MONITORS

```
MONIN:  PUSH  HL          ;SAVE OLD REGISTER CONTENTS
        LD    HL.USRINT    ;GET USER ADDRESS FOR SERVICE
        JP    (HL)         ;JUMP TO USER SERVICE ADDRESS
```

You must then place the address of your service routine into memory locations USRINT and USRINT+1. using the normal Z80 address format with the least significant bits at the lower address. Remember that MONIN is an address in the monitor program.

You can include the loading of memory locations USRINT and USRINT+1 in your main program; i.e.,

```
LD       HL,INTRP     ;GET STARTING ADDRESS OF SERVICE
                      ;  ROUTINE
LD       (USRINT),HL  ;STORE IT AS USER ADDRESS
```

These instructions come before the enabling of the interrupts.

In this example, the return address that the Z80 stores in the Stack is not useful. However, the main program still must initialize the Stack Pointer so that there is a definite place to put that address. You may not need the LD SP instruction if the monitor in your microcomputer manages the Stack Pointer.

The main program enables only the interrupt from the startup PIO. The PIO could, of course, be in any mode. The interrupt is enabled by setting bit 7 of an interrupt control word and writing that word to the PIO control port. The PIO interrupt is enabled before the overall interrupt system is enabled with the EI instruction.

Remember that $\overline{RESET}$ and accepting an interrupt automatically disable the interrupt system. This allows the real startup routine to configure all the PIOs and other interrupt sources without being interrupted.

No action is needed in the interrupt service routine, since the interrupt is automatically cleared as part of the Interrupt Acknowledge cycle involving a particular PIO.

The implementations of the instructions EI (Enable Interrupts) and DI (Disable Interrupts) differ on the Z80. DI takes effect immediately after its execution, while EI takes effect after the execution of the following instruction. The reasoning behind this fact is discussed in Chapter 3 under the description of the EI instruction.

A Keyboard Interrupt

Purpose: The computer waits for a keyboard interrupt and places the data from the keyboard into memory location 0040.

KEYBOARD INTERRUPT

Sample Problem:

Keyboard data = 06

Result: (0040) = 06

Flowchart:
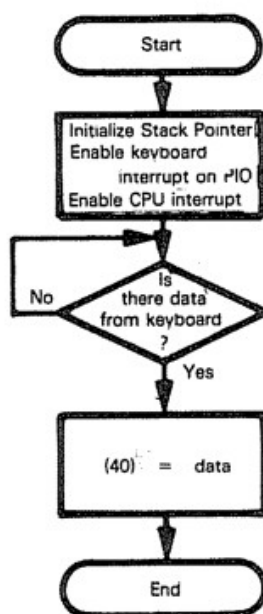
12-14

Source Program:

Main Program:

```
RESET   EQU     0
        ORG     RESET
        LD      SP.100H         :PUT STACK AT END OF MEMORY
        LD      A,01001111B     :PUT PIO IN INPUT MODE
        OUT     (PIOCRA),A
        LD      A,10000111B     :ENABLE PIO INTERRUPTS
        OUT     (PIOCRA),A
        EI                      :ENABLE CPU INTERRUPTS
HERE:   JR      HERE            :DUMMY MAIN PROGRAM
```

Interrupt Service Routine:

```
        ORG     INTRP
        EX      AF,AF'          :SAVE ACCUMULATOR, FLAGS
        IN      A,(PIODRA)      :GET KEYBOARD DATA
        LD      (40H),A         :SAVE KEYBOARD DATA
        EX      AF,AF'          :RESTORE ACCUMULATOR. FLAGS
        EI                      :RE-ENABLE INTERRUPTS
        RETI
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
Main Program:				
0000	31		LD	SP.100H
0001	00			
0002	01			
0003	3E		LD	A,01001111B
0004	4F			
0005	D3		OUT	(PIOCRA),A
0006	PIOCRA			
0007	3E		LD	A,10000111B
0008	87			
0009	D3		OUT	(PIOCRA),A
000A	PIOCRA			
000B	FB		EI	
000C	18	HERE:	JR	HERE
000D	FE			
Interrupt Service Routine:				
INTRP	08		EX	AF,AF'
INTRP+1	DB		IN	A,(PIODRA)
INTRP+2	PIODRA			
INTRP+3	32		LD	(40H),A
INTRP+4	40			
INTRP+5	00			
INTRP+6	08		EX	AF,AF'
INTRP+7	FB		EI	
INTRP+8	ED		RETI	
INTRP+9	4D			

The JR HERE is an endless loop (jump-to-self) instruction that is used to represent the main program. After interrupts are enabled in a working system, the main program goes about its business until an interrupt occurs and then resumes execution after the interrupt service routine is completed.

The RET instruction at the end of the service routine transfers control back to the JR instruction. If you want to avoid this, you can simply increment the Program Counter in the Stack, e.g..

<table>
<tr><td></td><td></td><td>**CHANGING THE RETURN ADDRESS**</td></tr>
</table>

EX	(SP),HL	:GET RETURN ADDRESS
INC	HL	:INCREMENT RETURN ADDRESS TWICE
INC	HL	
EX	(SP),HL	:RESTORE ADJUSTED ADDRESS TO STACK

The RET instruction will now transfer control to the instruction following the JR. Note the use of EX (SP),HL: this instruction exchanges the contents of Register Pair HL with the contents of the memory locations at the top of the Stack. By using it, we can adjust the return address without affecting the contents of Register Pair HL.

Since the Z80 does not automatically save its registers, you can use them to pass parameters and results between the main program and the interrupt service routine. So, you could leave the data in the Accumulator instead of in memory location 0040. This is, however, a dangerous practice that should be avoided in all but the most trivial systems. In most applications, the processor is using its registers during normal program execution; having the interrupt service routines randomly change the contents of those registers would surely cause havoc. In general, no interrupt service routine should ever alter any register unless that register's contents have been saved prior to its alteration and will be restored at the completion of the routine.

Note that you must explicitly re-enable the interrupts at the end of the service routine, since the processor disables the interrupt system when it accepts an interrupt. Servicing a PIO interrupt deactivates the interrupt signal so that the same interrupt is not serviced again.

If interrupt service routines are never themselves interrupted (i.e., there is only one level of interrupts), the instructions EX AF,AF' and EXX are a convenient way to save and restore the old contents of the user registers. EXX exchanges the contents of BC, DE, and HL with the contents of their primed equivalents. The two instructions

SAVING VALUES IN PRIMED REGISTERS

together take only two bytes of memory and eight clock cycles. However, this method cannot be used if there are other interrupt levels (since there is only a single set of primed registers) or if the primed registers are needed in either the main program or the interrupt service routine.

A more general approach to saving and restoring registers is to use the Stack. PUSH saves the contents of a register pair and POP restores the contents. However, PUSH takes 11 clock cycles and POP 10, so this approach is slower. It also uses extra memory locations in the Stack. The advantage of this method is that it can be expanded indefinitely (as long as there is room in the Stack) since nested service routines will not destroy the data saved by the earlier routines.

An alternative approach would be for the interrupt routine to maintain control until it received an entire line of text (e.g., a string of characters ending with a carriage return). The main program would be:

FILLING A BUFFER VIA INTERRUPTS

Main Program:

```
RESET   EQU     0
        ORG     RESET
        LD      SP,100H         ;PUT STACK AT END OF MEMORY
        LD      A,01001111B     ;PUT PIO IN INPUT MODE
        OUT     (PIOCRA),A
        LD      A,10000111B     ;ENABLE PIO INTERRUPTS
        OUT     (PIOCRA),A
        LD      HL,70H          ;INITIALIZE BUFFER POINTER
        LD      (40H),HL        ;SAVE BUFFER POINTER
        EI                      ;ENABLE CPU INTERRUPT
HERE:   JR      HERE            ;DUMMY MAIN PROGRAM
```

Interrupt Service Routine:

```
        ORG     INTRP
        EX      AF,AF'          ;SAVE A, FLAGS
        EXX                     ;SAVE OTHER REGISTERS
        LD      HL,(40H)        ;GET BUFFER POINTER
        IN      A,(PIODRA)      ;GET KEYBOARD DATA
        LD      (HL),A          ;SAVE DATA IN BUFFER
        CP      CR              ;IS DATA A CARRIAGE RETURN?
        JR      Z,ENDL          ;YES, END OF LINE
        INC     HL              ;NO, INCREMENT BUFFER POINTER
        LD      (40H),HL
        EXX                     ;RESTORE OTHER REGISTERS
        EX      AF,AF'          ;RESTORE A, FLAGS
        EI                      ;RE-ENABLE INTERRUPTS
        RETI
ENDL:   JP      LPROC           ;PROCESS LINE WITHOUT INTERRUPTS
```

When the processor receives a carriage return, it leaves the interrupt system disabled while it handles the line.

An alternative approach would be to fill another buffer while handling the first one; this approach is called double buffering.

```
┌──────────┐
│ DOUBLE   │
│ BUFFERING│
└──────────┘
```

The line processing routine is begun at address LPROC with interrupts disabled, the old register contents in the primed registers, and the original return address at the top of the Stack.

In a real application, the CPU could perform other tasks between interrupts. It could, for instance, edit, move, or transmit a line from one buffer while the interrupt was filling another buffer.

A Printer Interrupt

Purpose: The computer waits for a printer interrupt and sends the data from memory location 0040 to the printer.

Sample Problem:

(0040) = 51H

Result: Printer receives a 51H (ASCII Q) when it is ready.

Flowchart:
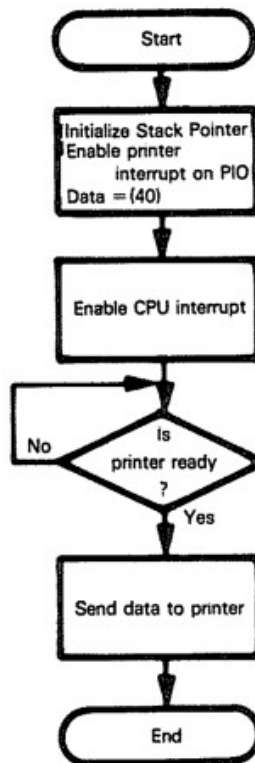
```
        ┌─────────────┐
        │    Start    │
        └──────┬──────┘
               ▼
     ┌───────────────────┐
     │Initialize Stack Pointer
     │Enable printer
     │        interrupt on PIO
     │Data = (40)        │
     └─────────┬─────────┘
               ▼
     ┌───────────────────┐
     │                   │
     │ Enable CPU interrupt
     │                   │
     └─────────┬─────────┘
               ▼
  No        ╱  Is  ╲
 ◄─────────◄ printer ready ►
            ╲   ?  ╱
               │ Yes
               ▼
     ┌───────────────────┐
     │                   │
     │ Send data to printer
     │                   │
     └─────────┬─────────┘
               ▼
        ┌─────────────┐
        │     End     │
        └─────────────┘
```

Source Program:

Main Program:

```
RESET    EQU     0
         ORG     RESET
         LD      SP,100H        :PUT STACK AT END OF MEMORY
         LD      A,00001111B    :PUT PIO IN OUTPUT MODE
         OUT     (PIOCRA),A
         LD      A,10000111B    :ENABLE PIO INTERRUPTS
         OUT     (PIOCRA),A
         EI                     :ENABLE CPU INTERRUPTS
HERE:    JR      HERE           :DUMMY MAIN PROGRAM
```

Interrupt Service Routine:

```
         ORG     INTRP
         EX      AF,AF'         :SAVE ACCUMULATOR, FLAGS
         LD      A,(40H)        :GET DATA
         OUT     (PIODRA),A     :SEND DATA TO PRINTER
         EX      AF,AF'         :RESTORE ACCUMULATOR, FLAGS
         EI                     :RE-ENABLE INTERRUPTS
         RETI
```

12-18

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
Main Program:				
0000	31		LD	SP,100H
0001	00			
0002	01			
0003	3E		LD	A,00001111B
0004	0F			
0005	D3		OUT	(PIOCRA),A
0006	PIOCRA			
0007	3E		LD	A,10000111B
0008	87			
0009	D3		OUT	(PIOCRA),A
000A	PIOCRA			
000B	FB		EI	
000C	18	HERE:	JR	HERE
000D	FE			
Interrupt Service Routine:				
INTRP	08		EX	AF,AF'
INTRP+1	3A		LD	A,(40H)
INTRP+2	40			
INTRP+3	00			
INTRP+4	D3		OUT	(PIODRA),A
INTRP+5	PIODRA			
INTRP+6	08		EX	AF,AF'
INTRP+7	FB		EI	
INTRP+8	ED		RETI	
INTRP+9	4D			

Here, as with the keyboard, you could have the printer continue to interrupt until it transferred an entire line of text. The main program and the service routine would be:

EMPTYING A BUFFER WITH INTERRUPTS

Main Program:

```
RESET   EQU     0
        ORG     RESET
        LD      SP,100H         ;PUT STACK AT END OF MEMORY
        LD      A,00001111B     ;PUT PIO IN OUTPUT MODE
        OUT     (PIOCRA),A
        LD      A,10000111B     ;ENABLE PIO INTERRUPTS
        OUT     (PIOCRA),A
        LD      HL,70H          ;INITIALIZE BUFFER POINTER
        LD      (40H),HL        ;SAVE BUFFER POINTER
        EI                      ;ENABLE CPU INTERRUPT
HERE:   JR      HERE            ;DUMMY MAIN PROGRAM
```

12-19

Interrupt Service Routine:

```
            ORG     INTRP
            EX      AF,AF'          ;SAVE A, FLAGS
            EXX                     ;SAVE OTHER REGISTERS
            LD      HL,(40H)        ;GET BUFFER POINTER
            LD      A,(HL)          ;GET A BYTE OF DATA FROM BUFFER
            OUT     (PIODRA),A      ;SEND DATA TO PRINTER
            CP      CR              ;IS DATA A CARRIAGE RETURN?
            JR      ENDL            ;YES, END OF LINE
            INC     HL              ;NO, INCREMENT BUFFER POINTER
            LD      (40H),HL
            EXX                     ;RESTORE OTHER REGISTERS
            EX      AF,AF'          ;RESTORE A, FLAGS
            EI                      ;RE-ENABLE INTERRUPTS
            RETI
ENDL:       JP      LCOMP           ;HANDLE COMPLETED LINE
```

Again, double buffering could be used to allow I/O and processing to occur at the same time without ever halting the CPU

A Real-Time Clock Interrupt

Purpose: The computer waits for an interrupt from a real-time clock.

> REAL-TIME
> CLOCK

A real-time clock simply provides a regular series of pulses. The interval between the pulses can be used as a time reference. Real-time clock interrupts can be counted to give any multiple of the basic time interval. A real-time clock can be produced by dividing down the CPU clock, by using a separate timer or a programmable timer like the CTC for Z80-based microcomputers, or by using external sources such as the AC line frequency.

Note the tradeoffs involved in determining the frequency of the real-time clock. A high frequency (say 10 kHz) allows the creation of a wide range of time intervals of high accuracy. On the other hand, the overhead involved in counting real-time clock

> FREQUENCY
> OF REAL-TIME
> CLOCK

interrupts may be considerable, and the counts will quickly exceed the capacity of a single 8-bit register or memory location. The choice of frequency depends on the precision and timing requirements of your application. The clock may, of course, consist partly of hardware; a counter may count high frequency pulses and interrupt the processor only occasionally. A program will have to read the counter to measure time to high accuracy.

One problem is synchronizing operations with the real-time clock. Clearly, there will be some effect on the precision of the timing interval if the CPU starts the measurement randomly during a clock period, rather than exactly at the beginning. Some ways to synchronize operations are:

> SYNCHRONIZATION
> WITH REAL-TIME
> CLOCK

1) Start the CPU and clock together. $\overline{\text{RESET}}$ or a startup interrupt can start the clock as well as the CPU.

2) Allow the CPU to start and stop the clock under program control.

3) Use a high-frequency clock so that an error of less than one clock period will be small.

4) Line up the clock (by waiting for an edge or interrupt) before starting the measurement.

A real-time clock interrupt should have very high priority, since the precision of the timing intervals will be affected by any delay in servicing the interrupt. The usual practice is to make the real-time clock the highest priority interrupt except for power failure.

The clock interrupt service routine is generally kept extremely short so that it does not interfere with other CPU activities.

a) Wait for Real-Time Clock

Source Program:

Main Program:

```
RESET   EQU    0
        ORG    RESET
        LD     SP,100H         ;PUT STACK AT END OF MEMORY
        LD     A,01001111B     ;PUT PIO IN INPUT MODE
        OUT    (PIOCRA),A
        LD     A,10000111B     ;ENABLE PIO INTERRUPTS
        OUT    (PIOCRA),A
        EI                     ;ENABLE CPU INTERRUPTS
HERE:   JR     HERE            ;DUMMY MAIN PROGRAM
```

Interrupt Service Routine:

```
        ORG    INTRP
        HALT                   ;END CLOCK INTERRUPT
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)		
Main Program:				
0000	31		LD	SP,100H
0001	00			
0002	01			
0003	3E		LD	A,01001111B
0004	4F			
0005	D3		OUT	(PIOCRA),A
0006	PIOCRA			
0007	3E		LD	A,10000111B
0008	87			
0009	D3		OUT	(PIOCRA),A
000A	PIOCRA			
000B	FB		EI	
000C	18	HERE:	JR	HERE
000D	FE			
Interrupt Service Routine:				
INTRP	76		HALT	

The service routine does not have to do anything, since servicing the PIO interrupt automatically clears it and there is no data to send or receive.

The real-time clock interrupt always occurs on a rising edge if a PIO STROBE signal is used for the clock input.

b) Wait for 10 Real-Time Clock Interrupts

Source Program:

Main Program:

```
RESET   EQU     0
        ORG     RESET
        LD      SP,100H         :PUT STACK AT END OF MEMORY
        LD      A,01001111B     :PUT PIO IN INPUT MODE
        OUT     (PIOCRA),A
        LD      A,10000111B     :ENABLE PIO INTERRUPTS
        OUT     (PIOCRA),A
        LD      HL,40H          :CLOCK COUNTER = ZERO
        LD      (HL),0
        LD      A,10            :NUMBER OF COUNTS = 10
        EI                      :ENABLE CPU INTERRUPTS
WTTEN:  CP      (HL)            :HAVE TEN COUNTS ELAPSED?
        JR      NZ,WTTEN        :NO, WAIT
        HALT                    :YES, DONE
```

Interrupt Service Routine:

```
        ORG     INTRP
        EXX                     :SAVE USER REGISTERS
        EX      AF,AF'          :SAVE A, FLAGS
        LD      HL,40H          :INCREMENT CLOCK COUNTER
        INC     (HL)
        EX      AF,AF'          :RESTORE A, FLAGS
        EXX                     :RESTORE USER REGISTERS
        EI                      :RE-ENABLE INTERRUPTS
        RETI
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
Main Program:			
0000	31	LD	SP,100H
0001	00		
0002	01		
0003	3E	LD	A,01001111B
0004	4F		
0005	D3	OUT	(PIOCRA),A
0006	PIOCRA		
0007	3E	LD	A,10000111B
0008	87		
0009	D3	OUT	(PIOCRA),A
000A	PIOCRA		
000B	21	LD	HL,40H
000C	40		
000D	00		
000E	36	LD	(HL),0
000F	00		
0010	3E	LD	A,10
0011	0A		
0012	FB	EI	
0013	BE	WTTEN: CP	(HL)
0014	20	JR	NZ,WTTEN
0015	FD		
0016	76	HALT	
Interrupt Service Routine:			
INTRP	D9	EXX	
INTRP+1	08	EX	AF,AF'
INTRP+2	21	LD	HL,40H
INTRP+3	40		
INTRP+4	00		
INTRP+5	34	INC	(HL)
INTRP+6	08	EX	AF,AF'
INTRP+7	D9	EXX	
INTRP+8	FB	EI	
INTRP+9	ED	RETI	
INTRP+10	4D		

An alternative approach uses the Stack to save and restore register values. To save H, L, and the flags requires:

```
PUSH    HL          :SAVE REGISTERS H AND L
PUSH    AF          :SAVE ACCUMULATOR AND FLAGS
```

To restore them requires the sequence:

```
POP     AF          :RESTORE ACCUMULATOR AND FLAGS
POP     HL          :RESTORE REGISTERS H AND L
```

Note that, if the Stack is used, registers must be restored in the opposite order from that in which they were saved. Clearly the order in which EXX and EX AF,AF' are executed does not matter.

This interrupt service routine merely updates the counter in memory location 0040. It is transparent to the main program.

A more realistic real-time clock interrupt routine could maintain real time in several memory locations. For example, the following routine uses addresses 0040 through 0043 as follows:

0040	-	hundredths of seconds
0041	-	seconds
0042	-	minutes
0043	-	hours

We assume that the routine is triggered by a 100 Hz clock.

Flowchart:

```
                    ┌──────────────┐
                    │    Start     │
                    └──────┬───────┘
                           ▼
              ┌────────────────────────┐
              │ Clear clock interrupt  │
              │ Hundredths =           │
              │     Hundredths + 1     │
              └───────────┬────────────┘
                          ▼
                    ◇ Is              No
                    ◇ Hundredths 100  ─────┐
                    ◇ ?                     │
                          │ Yes            │
                          ▼                │
              ┌────────────────────────┐  │
              │ Hundredths  =  0        │  │
              │ Seconds  =             │  │
              │     Seconds + 1        │  │
              └───────────┬────────────┘  │
                          ▼                │
                    ◇ Is              No   │
                    ◇ Seconds 60     ──────┤
                    ◇ ?                     │
                          │ Yes            │
                          ▼                │
              ┌────────────────────────┐  │
              │ Seconds  =  0           │  │
              │ Minutes  =             │  │
              │     Minutes + 1        │  │
              └───────────┬────────────┘  │
                          ▼                │
                    ◇ Is              No   │
                    ◇ Minutes 60     ──────┤
                    ◇ ?                     │
                          │ Yes            │
                          ▼                │
              ┌────────────────────────┐  │
              │ Minutes  =  0           │  │
              │ Hours  =               │  │
              │     Hours + 1          │  │
              └───────────┬────────────┘  │
                          ▼                │
                    ┌──────────────┐◄──────┘
                    │     End      │
                    └──────────────┘
```

12-24
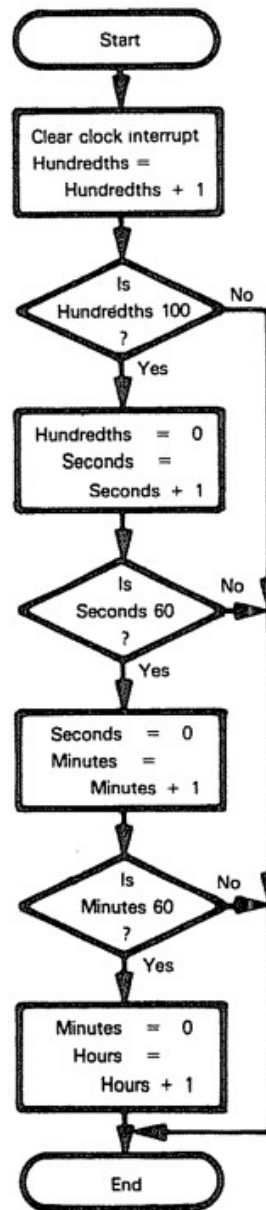
Source Program:

```
         ORG     INTRP
         PUSH    AF              :SAVE REGISTERS
         PUSH    HL
         LD      HL,40H          :UPDATE HUNDREDTHS OF SECONDS
         INC     (HL)
         LD      A,100
         CP      (HL)            ;IS THERE A CARRY TO SECONDS?
         JR      NZ,DONE         ;NO, DONE
         LD      (HL),0          ;YES, HUNDREDTHS = 0
         INC     HL              ;UPDATE SECONDS
         INC     (HL)
         LD      A,60
         CP      (HL)            ;IS THERE A CARRY TO MINUTES?
         JR      NZ,DONE         ;NO, DONE
         LD      (HL),0          ;YES, SECONDS = 0
         INC     HL              ;UPDATE MINUTES
         INC     (HL)
         CP      (HL)            ;IS THERE A CARRY TO HOURS?
         JR      NZ,DONE         ;NO, DONE
         LD      (HL),0          ;YES, MINUTES = 0
         INC     HL              ;UPDATE HOURS
         INC     (HL)
         LD      A,24            ;DAY COMPLETED?
         JR      NZ,DONE         ;NO, DONE
         LD      (HL),0          ;YES, HOURS = 0
DONE:    POP     HL              :RESTORE REGISTERS
         POP     AF
         EI                      :RE-ENABLE INTERRUPTS
         RETI
```

Now a wait of 300 ms could be produced in the main program with the routine:

```
         LD      HL,40H          ;GET PRESENT TIME (HUNDREDTHS OF SECS)
         LD      A,(HL)
         ADD     A,30            ;DESIRED TIME IS 30 COUNTS LATER
         CP      100             ;MOD 100
         JR      C,WT30
         SUB     100
WT30:    CP      (HL)            ;DESIRED TIME REACHED?
         JR      NZ,WT30         ;NO, WAIT
```

Be careful in this program of the difference between INC HL and INC (HL). INC HL adds 1 to the 16-bit contents of Register Pair HL, while INC (HL) adds 1 to the 8-bit contents of the memory location addressed by HL.

Of course, the program could perform other tasks and check the elapsed time only occasionally. How would you produce a delay of seven seconds?. Of three minutes?

Sometimes you may want to keep time either as BCD digits or as ASCII characters. How would you revise the last program to handle these alternatives?

You can disable the clock interrupt (or any other interrupt) when it is no longer needed in any of the following ways.

DISABLING INTERRUPTS

1) By executing a DI instruction in the main program. This disables the entire interrupt system.

2) By clearing bit 7 of the interrupt control word during the service routine or during the main program. This disables only the interrupt from one port of one PIO.

3) By not re-enabling the interrupt during the service routine.

Remember that the CPU automatically disables interrupts upon accepting one. Thus, the interrupt system is disabled unless the service routine explicitly re-enables it. Note, however, that you must be very careful about not re-enabling the interrupts, since the main program would be completely unaware that interrupts were no longer allowed. In general, all interrupt service routines should re-enable the interrupts before returning; any other policy means that the service routines are not transparent to the main program.

A Teletypewriter Interrupt

Purpose: The computer waits for data to be received from a teletypewriter and stores the data in memory location 0040.

a) Using an SIO

<div style="float:right; border:1px solid black; padding:4px;">
SIO
INTERRUPT
ROUTINE
</div>

(7-bit characters with odd parity and 2 stop bits).

Source Program:

Main Program:

```
RESET   EQU     0
        LD      A,4             :ACCESS WRITE REGISTER 4
        OUT     (SIOCRA),A
        LD      A,01000001B     ;X16 CLOCK MODE, PARITY
        OUT     (SIOCRA),A
        LD      A,3
        OUT     (SIOCRA),A
        LD      A,01000001B     ;7 BIT CHARACTERS, ENABLE RECEIVER
        OUT     (SIOCRA),A
        LD      A,1             ;ACCESS WRITE REGISTER 1
        OUT     (SIOCRA),A
        LD      A,00011000B     ;ENABLE RECEIVER INTERRUPT ON ALL CHARS
        OUT     (SIOCRA),A
        EI                      ;ENABLE CPU INTERRUPTS
HERE:   JR      HERE            ;DUMMY MAIN PROGRAM
```

Interrupt Service Routine:

```
        ORG     INTRP
        PUSH    AF              :SAVE ACCUMULATOR, FLAGS
        IN      A,(SIODRA)      :READ CHARACTER FROM SIO
        LD      (40H),A         :SAVE CHARACTER IN MEMORY
        POP     AF              :RESTORE ACCUMULATOR, FLAGS
        EI                      :RE-ENABLE INTERRUPTS
        RETI
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
Main Program:				
0000	3E		LD	A,4
0001	04			
0002	D3		OUT	(SIOCRA),A
0003	SIOCRA			
0004	3E		LD	A,01000001B
0005	41			
0006	D3		OUT	(SIOCRA),A
0007	SIOCRA			
0008	3E		LD	A,3
0009	03			
000A	D3		OUT	(SIOCRA),A
000B	SIOCRA			
000C	3E		LD	A,01000001B
000D	41			
000E	D3		OUT	(SIOCRA),A
000F	SIOCRA			
0010	3E		LD	A,1
0011	01			
0012	D3		OUT	(SIOCRA),A
0013	SIOCRA			
0014	3E		LD	A,00011000B
0015	18			
0016	D3		OUT	(SIOCRA),A
0017	SIOCRA			
0018	FB		EI	
0019	18	HERE:	JR	HERE
001A	FE			
Interrupt Service Routine:				
INTRP	F5		PUSH	AF
INTRP+1	DB		IN	A,(SIODRA)
INTRP+2	SIODRA			
INTRP+3	32		LD	(40H),A
INTRP+4	40			
INTRP+5	00			
INTRP+6	F1		POP	AF
INTRP+7	FB		EI	
INTRP+8	ED		RETI	
INTRP+9	4D			

This service routine assumes that only the receive interrupt from one channel of the SIO has been enabled. Otherwise, either further vectoring will be required by changing control bits D_2, D_3, and D_4 of Write Register 0 (see the discussion of SIO interrupts earlier in this chapter) or the routine will have to examine the status bits in Read Register 0. The key status bits are:

Bit 0 - Receive Character Available — 1 when at least one character is available in the receive buffers.

Bit 1 - Interrupt pending (Channel A only) — 1 if any interrupt is pending in the entire SIO.

Bit 2 - Transmit Buffer Empty — 1 if the Transmit buffer is empty.

Obviously, it would be far shorter and simpler to configure the SIO by using a table (in ROM) and the repeated Block I/O instruction, i.e.,

```
        LD      B,6             ;NUMBER OF BYTES IN CONFIGURATION
        LD      C,SIOCRA        ;SIO CONTROL PORT
        LD      HL,SIOTBL       ;START OF SIO CONFIGURATION TABLE
        OTIR                    ;CONFIGURE SIO
```

This method requires 9 bytes of memory for the program and 6 bytes for the table, as compared to the 23 bytes used in the example to configure the SIO.

The program establishes the SIO registers as follows:

WRITE REGISTER 4

Bit 7 = 0, bit 6 = 1 for X16 clock mode

Bit 1 = 0 to select odd parity

Bit 0 = 1 to enable parity generation

WRITE REGISTER 3

Bit 7 = 0, bit 6 = 1 to select 7-bit characters

Bit 0 = 1 to enable the receiver

WRITE REGISTER 1

Bit 4 = 1, bit 3 = 1 to produce an interrupt on all received characters with parity errors not affecting the vector.

The CPU clears the Received Character Available bit by reading a character from the SIO Data register. The Interrupt Pending bit is cleared automatically when the interrupt is serviced.

b) Using a PIO

(Received data tied to data bit 7 of PIO Port A).

```
┌─────────────┐
│ START BIT   │
│ INTERRUPT   │
└─────────────┘
```

Source Program:

Main Program:

```
        LD      A,11001111B     ;MAKE PORT A CONTROL
        OUT     (PIOCRA),A
        LD      A,10000000B     ;MAKE BIT 7 INPUT, OTHERS OUTPUTS
        OUT     (PIOCRA),A
        LD      A,10010111B     ;ENABLE INTERRUPT ON START BIT (0)
        OUT     (PIOCRA),A
        LD      A,01111111B     ;MASK OUT ALL OTHER BITS
        OUT     (PIOCRA),A
        EI                      ;ENABLE CPU INTERRUPTS
HERE:   JR      HERE            ;DUMMY MAIN PROGRAM
```

Interrupt Service Routine:

```
        ORG     INTRP
        PUSH    AF              ;SAVE ACCUMULATOR, FLAGS
        LD      A,00000111B     ;DISABLE START BIT INTERRUPT
        OUT     (PIOCRA),A
        CALL    TTYRCV          ;FETCH DATA FROM TTY
        LD      A,10000111B     ;ENABLE START BIT INTERRUPT
        OUT     (PIOCRA),A
        POP     AF              ;RESTORE ACCUMULATOR, FLAGS
        EI                      ;RE-ENABLE INTERRUPTS
        RETI
```

Object Program:

Memory Address (Hex)	Memory Contents (Hex)		Instruction (Mnemonic)	
Main Program:				
0000	3E		LD	A.11001111B
0001	CF			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	3E		LD	A.10000000B
0005	80			
0006	D3		OUT	(PIOCRA),A
0007	PIOCRA			
0008	3E		LD	A.10010111B
0009	97			
000A	D3		OUT	(PIOCRA),A
000B	PIOCRA			
000C	3E		LD	A.01111111B
000D	7F			
000E	D3		OUT	(PIOCRA),A
000F	PIOCRA			
0010	FB		EI	
0011	18	HERE:	JR	HERE
0012	FE			
Interrupt Service Routine:				
INTRP	F5		PUSH	AF
INTRP+1	3E		LD	A.00000111B
INTRP+2	07			
INTRP+3	D3		OUT	(PIOCRA),A
INTRP+4	PIOCRA			
INTRP+5	CD		CALL	TTYRCV
INTRP+6	TTYRCV			
INTRP+7				
INTRP+8	3E		LD	A.10000111B
INTRP+9	87			
INTRP+10	D3		OUT	(PIOCRA),A
INTRP+11	PIOCRA			
INTRP+12	F1		POP	AF
INTRP+13	FB		EI	
INTRP+14	ED		RETI	
INTRP+15	4D			

These programs assume that the monitor initializes the Stack Pointer. Otherwise, it will have to be loaded in the main program.

Subroutine TTYRCV is the TTY receive routine shown in the previous chapter.

The edge used to cause the interrupt is very important here. An interrupt must occur when the data line changes from the normal MARK or '1' state to the SPACE or '0' state, since this transition identifies the start of the transmission.

The service routine must disable the PIO interrupt, since otherwise each '1'-to-'0' transition in the character will cause an interrupt. Of course, you must re-enable the PIO interrupt after the entire character has been read.

Note the use of the PIO in the control mode:

1) The PIO is placed in the control mode by establishing Mode 3.

2) The next control word defines which data lines are to be inputs ('1') and which are to be outputs ('0').

3) The interrupt control word has, besides the usual enable in bit 7,

bit 6 = 0 to perform a logical OR of the monitored data lines for an interrupt (not used in this case, since only one line is monitored)

bit 5 = 0 to define the active polarity of the data lines as low (for the start bit in this case)

bit 4 = 1 to indicate that a mask word follows.

4) The next control word contains the interrupt masks. Only those port lines with a mask bit of zero will be monitored for generating an interrupt.

The net result is for an interrupt to be generated if bit 7 is zero or changes from one to zero. Note that further interrupts occur only when a change occurs in the status of the logical equation. Here again, the PIO could be configured by using a table and the repeated block output instruction.

MORE GENERAL SERVICE ROUTINES

More general service routines that are part of a complete interrupt-driven system must handle the following tasks:

> **TASKS FOR GENERAL SERVICE ROUTINES**

1) Saving all registers that are used in the interrupt service routine in the Stack so that the interrupted program can be correctly resumed.

Remember that the Z80 Push instruction transfers a register pair (or an index register) to the Stack. PUSH AF (F is the Flag register) transfers the Accumulator and flags to the Stack.

A routine to save all the registers in the Stack would be:

```
PUSH    AF          :SAVE ACCUMULATOR. FLAGS
PUSH    BC          :SAVE REGISTERS B.C
PUSH    DE          :SAVE REGISTERS D.E
PUSH    HL          :SAVE REGISTERS H.L
PUSH    IX          :SAVE INDEX REGISTER IX
PUSH    IY          :SAVE INDEX REGISTER IY
EX      AF,AF'
EXX
PUSH    AF          :SAVE PRIMED ACCUMULATOR, FLAGS
PUSH    BC          :SAVE PRIMED REGISTERS B,C
PUSH    DE          :SAVE PRIMED REGISTERS D,E
PUSH    HL          :SAVE PRIMED REGISTERS H,L
```

Of course, only those registers that are used by the interrupt service routine must be saved.

2) Restoring all registers from the Stack after completing the interrupt service routine. Remember that registers must be restored in the opposite order from that in which they were saved.

3) Enabling and disabling interrupts appropriately. Remember that the CPU automatically disables its interrupts upon accepting one.

The service routines should be transparent as far as the interrupted program is concerned (i.e.. they should have no incidental effects).

Any standard subroutines that are used by an interrupt service routine must be reentrant. If some subroutines cannot be made reentrant, the interrupt service routine must have separate versions to use.[5]

PROBLEMS

1) A Test Interrupt

Purpose: The computer waits for a PIO interrupt to occur, then executes the endless loop instruction:

HERE: JR   HERE

until the next interrupt occurs.

2) A Keyboard Interrupt

Purpose: The computer waits for a 4-digit entry from a keyboard and places the digits into memory locations 0040 through 0043 (first one received in 0040). Each digit entry causes an interrupt. The fourth entry should also result in the disabling of the keyboard interrupt.

Sample Problem:

Keyboard data = 04, 06, 01, 07

Result: (0040) = 04
         (0041) = 06
         (0042) = 01
         (0043) = 07

3) A Printer Interrupt

Purpose: The computer sends four characters from memory locations 0040 to 0043 (starting with 0040) to the printer. Each character is requested by an interrupt. The fourth transfer also disables the printer interrupt.

4) A Real-Time Clock Interrupt

Purpose: The computer clears memory location 0040 initially and then complements memory location 0040 each time the real-time clock interrupt occurs.

How would you change the program so that it complements memory location 0040 after every ten interrupts? How would you change the program so that it leaves memory location 0040 at zero for ten clock periods, FF_{16} for five clock periods, and so on continuously? You may want to use a display rather than memory location 0040 so that it will be easier to see.

5) A Teletypewriter Interrupt

Purpose: The computer receives TTY data from an interrupting SIO and stores the characters in a buffer starting in memory location 0040. The process continues until the computer receives a carriage return $(0D_{16})$.

Assume that the characters are 7-bit ASCII with odd parity. How would you change your program to use a PIO? Assume that subroutine TTYRCV is available, as in the example. Include the carriage return as the final character in the buffer.

REFERENCES

1. You may want to review the discussion of interrupts in Volume 1 of <u>An Introduction to Microcomputers</u>.

2. For a discussion of designing with interrupts, see R. L. Baldridge, "Interrupts Add Power, Complexity to Microcomputer System Design," <u>EDN</u>, August 5, 1977, pp. 67-73.

3. See Volume 2 of <u>An Introduction to Microcomputers</u>.

4. See <u>An Introduction to Microcomputers, Volume 2</u> and <u>8080A/8085 Assembly Language Programming</u>.

5. For further discussion and some real-life examples of designing Z80-based systems with interrupts, see pp. 5-24 through 5-37 of <u>Z80 Programming for Logic Design</u> and the following:

   Baldridge, R. L.,•"Interrupts Add Power, Complexity to Microcomputer System Design," <u>EDN</u>, August 5, 1977, pp. 67-73.

   Pond, R. M., "Let Microprocessors Communicate," <u>Electronic Design</u>, November 8, 1977, pp. 88-90.

   Shima, M., and R. Blacksher, "Improved Microprocessor Interrupt Capability," <u>Electronic Design</u>, April 26, 1978, pp. 96-100.

   Weller, W. J., <u>Practical Microcomputer Programming: the Z80</u>, Northern Technology Books, Evanston, Ill., 1978.

   Winston, A. W., and T. B. Smith, "Use of the Z-80 in Data Collection and Control," <u>IECI '78 Proceedings - Industrial Applications of Microprocessors</u>, March 20-22, 1978, pp. 208-214.

The Proceedings of the IEEE's Industrial Electronics and Control Instrumentation Group's Annual Meeting on "Industrial Applications of Microprocessors" contains many interesting articles. Volumes (starting with 1975) are available from IEEE Service Center, CP Department, 445 Hoes Lane, Piscataway, NJ 08854.

Chapter 13
PROBLEM DEFINITION AND
PROGRAM DESIGN

THE TASKS OF SOFTWARE DEVELOPMENT

In the previous chapters, we have concentrated on the writing of short programs in assembly language. While this is an important topic, it is only a small part of software development. Although writing assembly language programs is a major task for the beginner, it soon becomes simple. By now, you should be familiar with standard methods for programming in assembly language on the Z80 microprocessor. **The next four chapters will describe how to formulate tasks as programs and how to combine short programs to form a working system.**

Software development consists of many stages. Figure 13-1 is a flowchart of the software development process. **Its stages are:**

<div style="float:right; border:1px solid black; padding:4px">
STAGES OF
SOFTWARE
DEVELOPMENT
</div>

- Problem definition
- Program design
- Coding
- Debugging
- Testing
- Documentation
- Maintenance and redesign

Each of these stages is important in the construction of a working system. Note that coding, the writing of programs in a form that the computer understands, is only one of seven stages.

In fact, **coding is usually the easiest stage to define and perform.** The rules for writing computer programs are easy to learn. They vary somewhat from computer to computer, but the basic techniques remain the same. Few software projects run into trou-

<div style="float:right; border:1px solid black; padding:4px">
RELATIVE
IMPORTANCE
OF CODING
</div>

ble because of coding; indeed, coding is not the most time-consuming part of software development. Experts estimate that a programmer can write one to ten fully debugged and documented statements per day. Clearly, the mere coding of one to ten statements is hardly a full day's effort. On most software projects, coding occupies less than 25% of the programmer's time.

Measuring progress in the other stages is difficult. You can say that half of the program has been written, but you can hardly say that half of the errors have been removed or half of the problem has been defined. Timetables for such stages as program design,

<div style="float:right; border:1px solid black; padding:4px">
MEASURING
PROGRESS
IN STAGES
</div>

debugging, and testing are difficult to produce. Many days or weeks of effort may result in no clear progress. Furthermore, an incomplete job in one stage may result in tremendous problems later. For example, poor problem definition or program design can make debugging and testing very difficult. Time saved in one stage may be spent many times over in later stages.
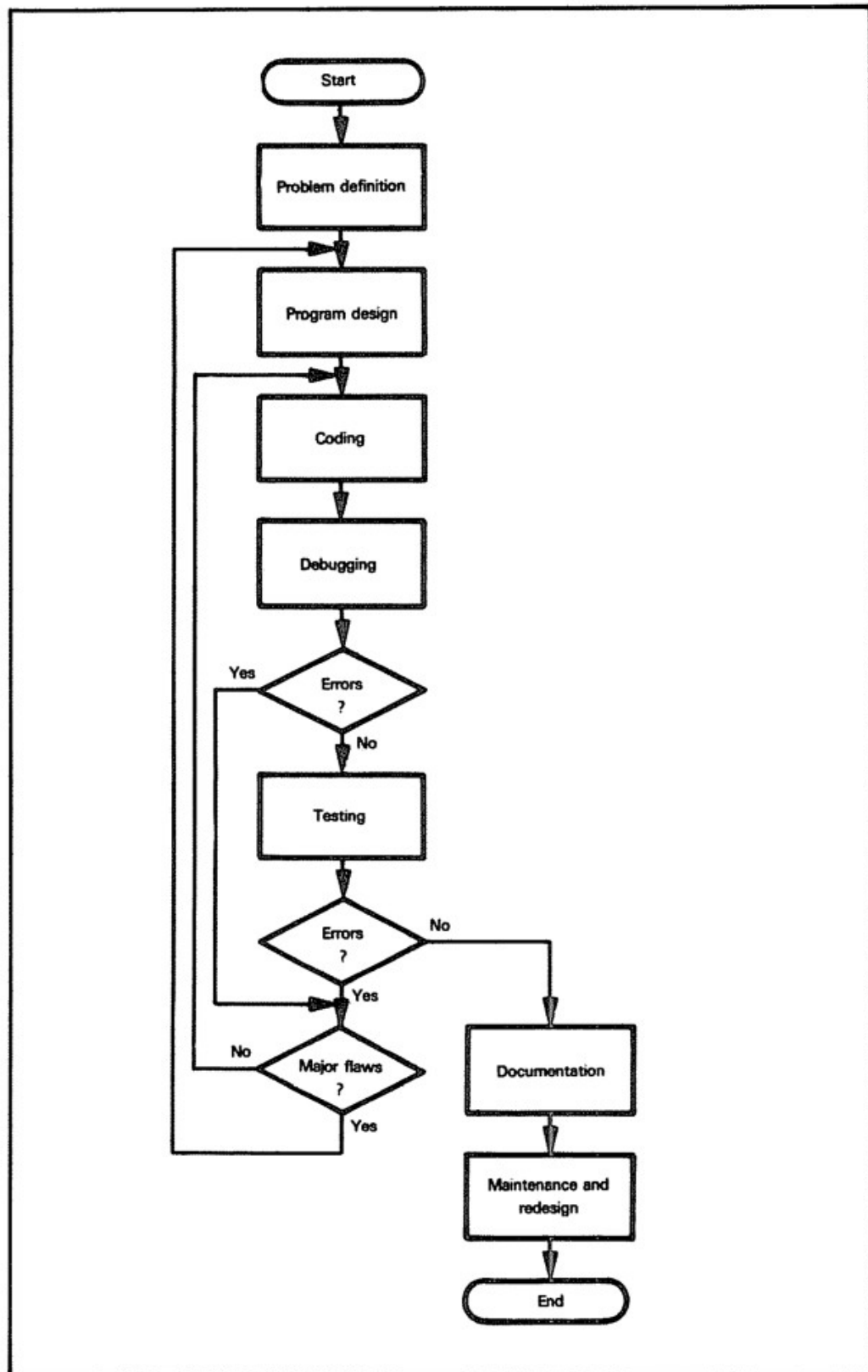
Figure 13-1. Flowchart of Software Development

DEFINITION OF THE STAGES

Problem definition is the formulation of the task in terms of the requirements that it places on the computer. For example, what is necessary to make a computer control a tool, run a series

| PROBLEM |
| DEFINITION |

of electrical tests, or handle communications between a central controller and a remote instrument? Problem definition requires that you determine the forms and rates of inputs and outputs, the amount and speed of processing that is needed, and the types of possible errors and their handling. Problem definition takes a vague idea of building a computer-controlled system and defines the tasks and requirements for the computer.

Program design is the outline of the computer program which will perform the tasks that have been defined. In the design

| PROGRAM |
| DESIGN |

stage, the tasks are described in a way that can easily be converted into a program. **Among the useful techniques in this stage are flowcharting, structured programming, modular programming, and top-down design.**

Coding is the writing of the program in a form that the computer can either directly understand or translate. The form may

| CODING |

be machine language, assembly language, or a high-level language.

Debugging, also called program verification, **is making the program do what the design specified that it would do.** In this

| DEBUGGING |

stage, you use such tools as breakpoints, traces, simulators, logic analyzers, and in-circuit emulators. The end of the debugging stage is hard to define, since you never know when you have found the last error.

Testing, also referred to as program validation, **is ensuring that the program performs the overall system tasks correctly.** The

| TESTING |

designer uses simulators, exercisers, and various statistical techniques to get some measure of the program's performance.

Documentation is the description of the program in the proper form for users and maintenance personnel. Docu-

| DOCUMENTATION |

mentation also allows the designer to develop a program library so that subsequent tasks will be far simpler. Flowcharts, comments, memory maps, and library forms are some of the tools used in documentation.

Maintenance and redesign are the servicing, improvement, and extension of the program. Clearly, the designer must be ready to handle field problems in computer-based equipment. Special diagnostic modes or programs and other maintenance

| MAINTENANCE |
| AND |
| REDESIGN |

tools may be required. Upgrading or extension of the program may be necessary to meet new requirements or handle new tasks.

The rest of this chapter will consider only the problem definition and program design stages. Chapter 14 will discuss debugging and testing, and Chapter 15 will discuss documentation, extension, and redesign. We will bring all the stages together in some simple systems examples in Chapter 16.

PROBLEM DEFINITION

Typical microprocessor tasks require a lot of definition. For example, what must a program do to control a scale, a cash register, or a signal generator? Clearly, we have a long way to go just to define the tasks involved.

DEFINING THE INPUTS

How do we start the definition? The obvious place to begin is with the inputs. **We should begin by listing all the inputs that the computer may receive in this application.**

Examples of inputs are:

- Data blocks from transmission lines
- Status words from peripherals
- Data from A/D converters

Then, we may ask the following questions about each input:

1) What is its form; i.e., what signals will the computer actually receive?
2) When is the input available and how does the processor know it is available? Does the processor have to request the input with a strobe signal? Does the input provide its own clock?
3) How long is it available?
4) How often does it change, and how does the processor know that it has changed?
5) Does the input consist of a sequence or block of data? Is the order important?
6) What should be done if the data contains errors? These may include transmission errors, incorrect data, sequencing errors. extra data, etc.
7) Is the input related to other inputs or outputs?

DEFINING THE OUTPUTS

The next step to define is the output. **We must list all the outputs that the computer must produce.** Examples of outputs include:

- Data blocks to transmission lines
- Control words to peripherals
- Data to D/A converters

Then, we may ask the following questions about each output:

1) What is its form; i.e.. what signals must the computer produce?
2) When must it be available, and how does the peripheral know it is available?
3) How long must it be available?
4) How often must it change, and how does the peripheral know that it has changed?
5) Is there a sequence of outputs? Is the order important?
6) What should be done to avoid transmission errors or to sense and recover from peripheral failures?
7) How is the output related to other inputs and outputs?

PROCESSING SECTION

Between the reading of input data and the sending of output results is the processing section. Here **we must determine exactly how the computer must process the input data. The questions are:**

1) What is the basic procedure (algorithm) for transforming input data into output results?

<div style="float:right; border:1px solid black; padding:2px;">

FACTORS IN PROCESSING

</div>

2) What time constraints exist? These may include data rates, delay times, the time constants of input and output devices, etc.

3) What memory constraints exist? Do we have limits on the amount of program memory or data memory, or on the size of buffers?

4) What standard programs or tables must be used? What are their requirements?

5) What special cases exist, and how should the program handle them?

6) How accurate must the results be?

7) How should the program handle processing errors or special conditions such as overflow, underflow, or loss of significance?

ERROR HANDLING

An important factor in many applications is the handling of errors. Clearly, the designer must make provisions for recovering from common errors and for diagnosing malfunctions. **Among the questions that the designer must ask at the definition stage are:**

1) What errors could occur?

<div style="float:right; border:1px solid black; padding:2px;">

ERROR CONSIDERATIONS

</div>

2) Which errors are most likely? If a person operates the system, human error is the most common. Following human errors, communications or transmission errors are more common than mechanical, electrical, mathematical, or processor errors.

3) Which errors will not be immediately obvious to the system? A special problem is the occurrence of errors that the system or operator may not recognize as incorrect.

4) How can the system recover from errors with a minimum loss of time and data and yet be aware that an error has occurred?

5) Which errors or malfunctions cause the same system behavior? How can these errors or malfunctions be distinguished for diagnostic purposes?

6) Which errors involve special system procedures? For example, do parity errors require retransmission of data?

Another question is: How can the field technician systematically find the source of malfunctions without being an expert? Built-in test programs, special diagnostics, or signature analysis can help.[1]

HUMAN FACTORS

Many microprocessor-based systems involve human interaction. **Human factors must be considered throughout the development process for** such **systems. Among the questions that the designer must ask are:**

1) What input procedures are most natural for the human operator?

2) Can the operator easily determine how to begin, continue and end the input operations?

3) How is the operator informed of procedural errors and equipment malfunctions?

4) What errors is the operator most likely to make?

5) How does the operator know that data has been entered correctly?

6) Are displays in a form that the operator can easily read and understand?

7) Is the response of the system adequate for the operator?

8) Is the system easy for the operator to use?

9) Are there guiding features for an inexperienced operator?

10) Are there shortcuts and reasonable options for the experienced operator?

11) Can the operator always determine or reset the state of the system after interruptions or distractions?

Building a system for people to use is difficult. The microprocessor can make the system more powerful, more flexible, and more responsive. However, the designer still must add the human touches that can greatly increase the usefulness and attractiveness of the system and the productivity of the human operator.[2]

EXAMPLES

Response to a Switch

Figure 13-2 shows a simple system in which the input is from a single SPST switch and the output is to a single LED display. In response to a switch closure, the processor turns the display on for one second. This system should be easy to define.

Let us first examine the input and answer each of the questions previously presented:

1) The input is a single bit, which may be either '0' (switch closed) or '1' (switch open).

2) The input is always available and need not be requested.

3) The input is available for at least several milliseconds after the closure.

4) The input will seldom change more than once every few seconds. The processor has to handle only the bounce in the switch. The processor must monitor the switch to determine when it is closed.

5) There is no sequence of inputs.

6) The obvious input errors are switch failure, failure in the input circuitry, and the operator attempting to close the switch again before a sufficient amount of time has elapsed. We will discuss the handling of these errors later.

7) The input does not depend on any other inputs or outputs.

The switch input is a '1' if the switch is open, '0' if the switch is closed. The CPU applies the output to the cathode of the LED; a '0' lights the display.
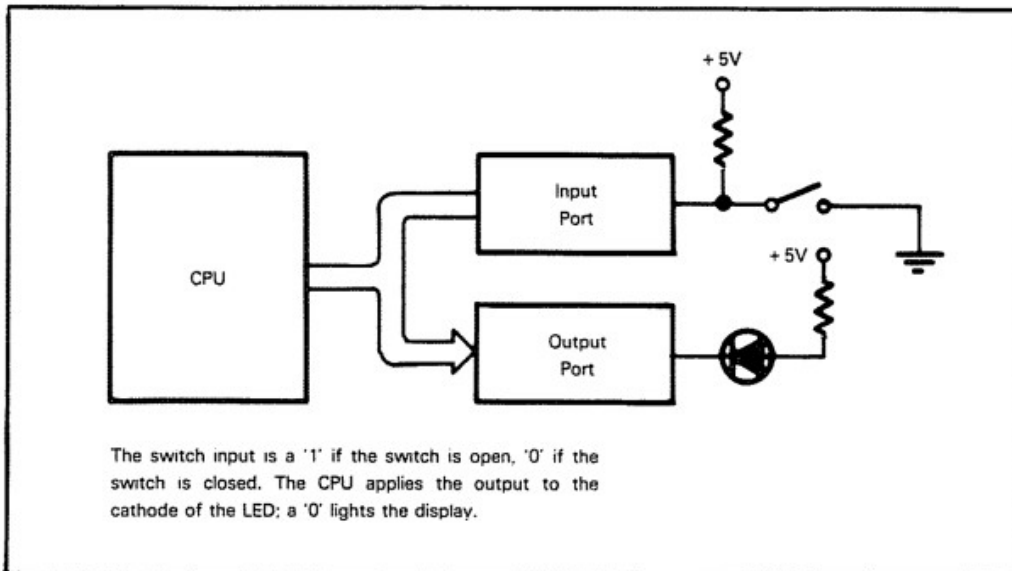
Figure 13-2. The Switch and Light System

The next requirement in defining the system is to examine the output. The answers to our questions are:

SWITCH
AND LIGHT
OUTPUTS

1) The output is a single bit, which is '0' to turn the display on, '1' to turn it off.

2) There are no time constraints on the output. The peripheral does not need to be informed of the availability of data.

3) If the display is an LED, the data need be available for only a few milliseconds at a pulse rate of about 100 times per second. The observer will see a continuously lit display.

4) The data must change (go off) after one second.

5) There is no sequence of outputs.

6) The possible output errors are display failure and failure in the output circuitry.

7) The output depends only on the switch input and time.

The processing section is extremely simple. As soon as the switch input becomes a logic '0', the CPU turns the light on (a logic '0') for one second. No time or memory constraints exist.

Let us now look at the possible errors and malfunctions. These are:

SWITCH AND
LIGHT ERROR
HANDLING

- Another switch closure before one second has elapsed
- Switch failure
- Display failure
- Computer failure

Surely the first error is the most likely. The simplest solution is for the processor to ignore switch closures until one second has elapsed. This brief unresponsive period will hardly be noticeable to the human operator. Furthermore, ignoring the switch during this period means that no debouncing circuitry or software is necessary, since the system will not react to the bounce anyway.

13-7

Clearly, the last three failures can produce unpredictable results. The display may stay on, stay off, or change state randomly. Some possible ways to isolate the failures would be:

- Lamp-test hardware to check the display; i.e., a button that turns the light on independently of the processor
- A direct connection to the switch to check its operation
- A diagnostic program that exercises the input and output circuits

If both the display and switch are working, the computer is at fault. A field technician with proper equipment can determine the cause of the failure.

A Switch-Based Memory Loader

Figure 13-3 shows a system that allows the user to enter data into any memory location in a microcomputer. One input port, DPORT, reads data from eight toggle switches. The other input port, CPORT, is used to read control information. There are three momentary switches: High Address, Low Address and Data. The output is the value of the last completed entry from the data switches; eight LEDs are used for the display.

DEFINING A
SWITCH-BASED
MEMORY LOADER

The system will also, of course, require various resistors, buffers, and drivers.

We shall first examine the inputs. The characteristics of the switches are the same as in the previous example; however, here there is a distinct sequence of inputs, as follows:

1) The operator must set the data switches according to the eight most significant bits of an address, then
2) press the High Address button. The high address bits will appear on the lights, and the program will interpret the data as the high byte of the address.
3) Then the operator must set the data switches with the value of the least significant byte of the address and
4) press the Low Address button. The low address bits will appear on the lights, and the program will consider the data to be the low byte of the address.
5) Finally, the operator must set the desired data into the data switches and
6) press the Data button. The display will now show the data, and the program stores the data in memory at the previously entered address.

The operator may repeat the process to enter an entire program. Clearly, even in this simplified situation, we will have many possible sequences to consider. How do we cope with erroneous sequences and make the system easy to use?

Output is no problem. After each input, the program sends to the displays the complement (since the displays are active-low) of the input bits. The output data remains the same until the next input operation.

The processing section remains quite simple. There are no time or memory constraints. The program can debounce the switches by waiting for a few milliseconds, and must provide complemented data to the displays.
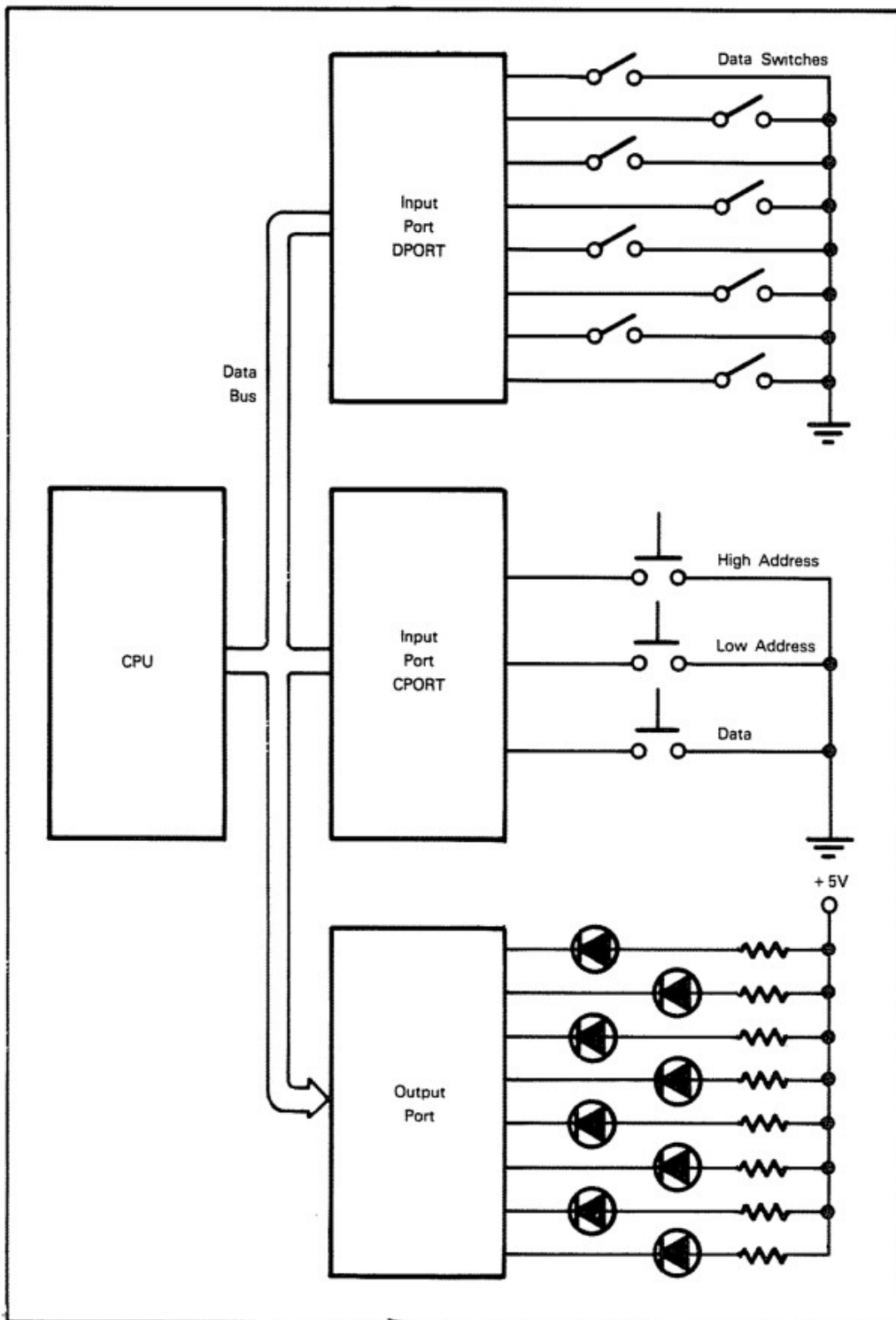
Figure 13-3. The Switch-Based Memory Loader

The most likely errors are operator mistakes. These include:

- Incorrect entries
- Incorrect order
- Incomplete entries; for example, forgetting the data

The system must be able to handle these problems in a reasonable way, since they are certain to occur in actual operation.

The designer must also consider the effects of equipment failure. Just as before, the possible difficulties are:

- Switch failure
- Display failure
- Computer failure

In this system, however, we must pay more attention to how these failures affect the system. A computer failure will presumably cause very unusual behavior by the system, and will be easy to detect. A display failure may not be immediately noticeable; here a Lamp Test feature will allow the operator to check the operation. Note that we would like to test each LED separately, in order to diagnose the case in which output lines are shorted together. In addition, the operator may not immediately detect switch failure; however, the operator should soon notice it and establish which switch is faulty by a process of elimination.

Let us look at some of the possible operator errors. Typical errors will be:

- Erroneous data
- Wrong order of entries or switches
- Trying to go on to the next entry without completing the current one

The operator will presumably notice erroneous data as soon as it appears on the displays. What is a viable recovery procedure for the operator? Some of the options are:

1) The operator must complete the entry procedure; i.e., enter Low Address and Data if the error occurs in the High Address. Clearly, this procedure is wasteful and would only serve to annoy the operator.

2) The operator may restart the entry process by returning to the high address entry steps. This solution is useful if the error was in the High Address, but forces the operator to re-enter earlier data if the error was in the Low Address or Data stage.

3) The operator may enter any part of the sequence at any time simply by setting the Data switches with the desired data and pressing the corresponding button. This procedure allows the operator to make corrections at any point in the sequence.

This type of procedure should always be preferred over one that does not allow immediate error correction, has a variety of concluding steps, or enters data into the system without allowing the operator a final check. Any added complication in hardware or software will be justified in increased operator efficiency. You should always prefer to let the microcomputer do the tedious work and recognize arbitrary sequences; it never gets tired and never forgets what was in the operating manual.

A further helpful feature would be status lights that would define the meaning of the display. Three status lights, marked "High Address", "Low Address", and "Data", would let the operator know what had been entered without having to remember which button was pressed. The processor would have to monitor the sequence, but the added complication in software would simplify the operator's task. Clearly, three separate sets of displays plus the ability to examine a memory location would be even more helpful to the operator.
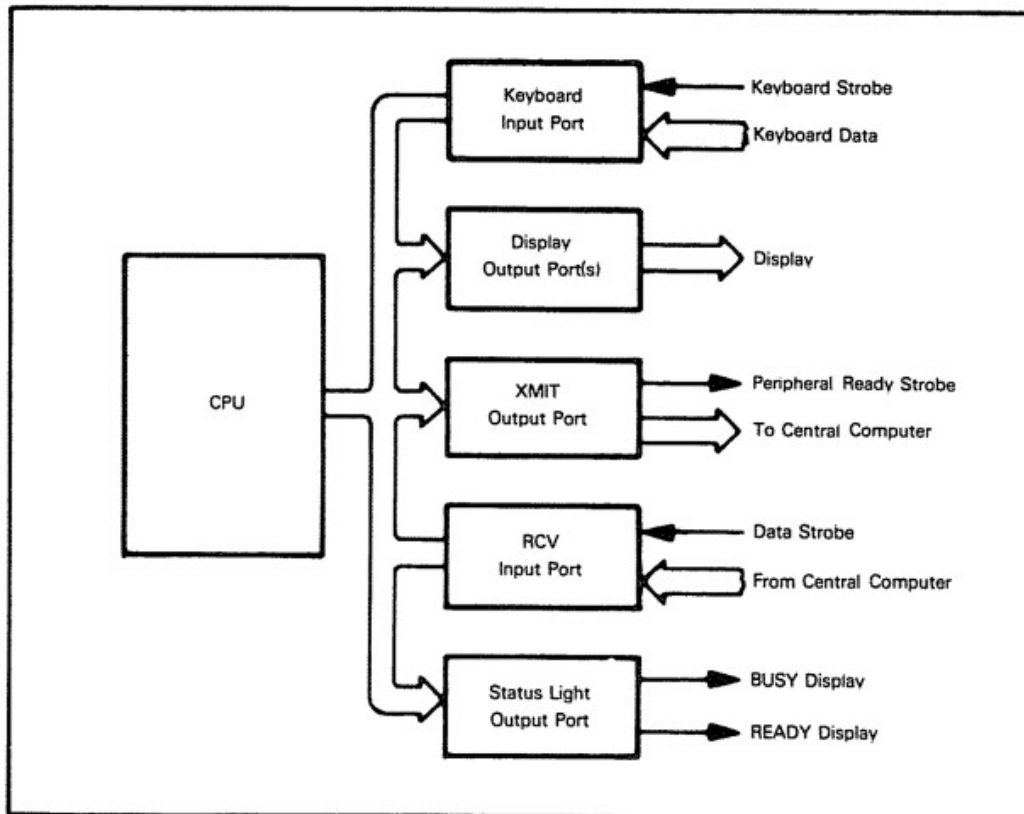
Figure 13-4. Block Diagram of a Verification Terminal

We should note that, although we have emphasized human interaction, machine or system interaction has many of the same characteristics. The microprocessor should do the work. If complicating the microprocessor's task makes error recovery simple and the causes of failure obvious, the entire system will work better and be easier to maintain. Note that you should not wait until after the software has been completed to consider system use and maintenance: instead, you should include these factors in the problem definition stage.

A Verification Terminal

Figure 13-4 is a block diagram of a simple credit-verification terminal. One input port derives data from a keyboard (see Figure 13-5); the other input port accepts verification data from a transmission line. One output port sends data to a set of

> **DEFINING A VERIFICATION TERMINAL**

displays (see Figure 13-6); another sends the credit card number to the central computer. A third output port turns on one light whenever the terminal is ready to accept an inquiry, and another light when the operator sends the information. The "Busy" light turns off when the response returns. Clearly, the input and output of data will be more complex than in the previous case, although the processing is still simple.

The digit keys allow digit entries.
CLEAR deletes the entire entry.
SEND transmits the entry to the central computer.

Figure 13-5. Verification Terminal Keyboard

The display consists of ten 7-segment displays, which may be multiplexed, controlled by a shift register, or addressed separately. Two additional lights, READY and BUSY, are also present.
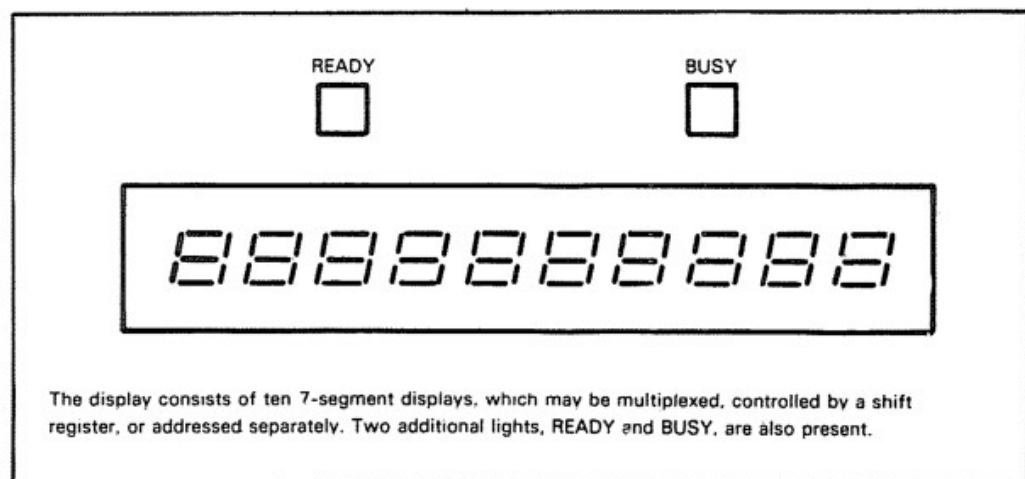
Figure 13-6. Verification Terminal Display

Additional displays may be useful to emphasize the meaning of the response. Many terminals use a green light for "Yes", a red light for "No", and a yellow light for "Consult Store Manager." Note that these lights will still have to be clearly marked with their meanings to allow for a color-blind operator.

Let us first look at the keyboard input. This is, of course, different from the switch input, since the CPU must have some way of distinguishing new data. **We will assume that each key closure provides a unique hexadecimal code (we can code**

```
VERIFICATION
TERMINAL
INPUTS
```

each of the 12 keys into one digit) and a strobe. The program will have to recognize the strobe and fetch the hexadecimal number that identifies the key. There is a time constraint, since the program cannot miss any data or strobes. The constraint is not serious, since keyboard entries will be at least several milliseconds apart.

The transmission input similarly consists of a series of characters, each identified by a strobe (perhaps from a UART). The program will have to recognize each strobe and fetch the character. The data being sent across the transmission lines is usually organized into messages. A possible message format is:

- Introductory characters, or header
- Terminal destination address
- Coded yes or no
- Ending characters, or trailer

The terminal will check the header, read the destination address, and see if the message is intended for it. If the message is for the terminal, the terminal accepts the data. The address could be (and often is) hard-wired into the terminal so that the terminal receives only messages intended for it. This approach simplifies the software at the cost of some flexibility.

The output is also more complex than in the earlier examples. If the displays are multiplexed, the processor must not only send the data to the display port but must also direct the data to a particular display. We will need either a separate control port

```
VERIFICATION
TERMINAL
OUTPUTS
```

or a counter and decoder to handle this. Note that hardware blanking controls can blank leading zeros as long as the first digit in a multi-digit number is never zero. Software can also handle this task. Time constraints include the pulse length and frequency required to produce a continuous display for the operator.

The communications output will consist of a series of characters with a particular format. The program will also have to consider the time required between characters. A possible format for the output message is:

- Header
- Terminal address
- Credit card number
- Trailer

A central communications computer may poll the terminals, checking for data ready to be sent.

The processing in this system involves many new tasks, such as:

- Identifying the control keys by number and performing the proper actions
- Adding the header, terminal address, and trailer to the outgoing message
- Recognizing the header and trailer in the returning message
- Checking the incoming terminal address

Note that none of the tasks involve any complex arithmetic or any serious time or memory constraints.

The number of possible errors in this system is, of course, much larger than in the earlier examples. Let us first consider the possible operator errors. These include:

- Entering the credit card number incorrectly
- Trying to send an incomplete credit card number
- Trying to send another number while the central computer is processing one
- Clearing non-existent entries

Some of these errors can be easily handled by correctly structuring the program. For example, the program should not accept the Send key until the credit card number has been completely entered, and it should ignore any additional keyboard entries until the response comes back from the central computer. Note that the operator will know that the entry has not been sent, since the Busy light will not go on. The operator will also know when the keyboard has been locked out (the program is ignoring keyboard entries), since entries will not appear on the display and the Ready light will be off.

Incorrect entries are an obvious problem. If the operator recognizes an error, he can use the Clear key to make corrections. The operator would probably find it more convenient to have two Clear keys, one that cleared the most recent key and one that cleared the entire entry. This would allow both for the situation in which the operator recognizes the error immediately and for the situation in which the operator recognizes the error late in the procedure. The operator should be able to correct errors immediately and have to repeat as few keys as possible. The operator will, however, make a certain number of errors without recognizing them. Most credit card numbers include a self-checking digit; the terminal could check the number before permitting it to be sent to the central computer. This step would save the central computer from wasting precious processing time checking the number.

This requires, however, that the terminal have some way of informing the operator of the error, perhaps by flashing one of the displays or by providing some other special indicator that the operator is sure to notice.

Still another problem is how the operator knows that an entry has been lost or processed incorrectly. Some terminals simply unlock after a maximum time delay. The operator notes that the Busy light has gone off without an answer being received. The operator is then expected to try the entry again. After one or two retries, the operator should report the failure to supervisory personnel.

Many equipment failures are also possible. Besides the displays, keyboard, and processor, there now exist the problems of communications errors or failures and central computer failures.

The data transmission will probably have to include error checking and correcting procedures. Some possibilities are:

1) Parity provides an error detection facility but no correction mechanism. The receiver will need some way of requesting retransmission, and the sender will have to save a copy of the data until proper reception is acknowledged. Parity is, however, very simple to implement.

<div style="float:right; border:1px solid black; padding:4px">

**CORRECTING
TRANSMISSION
ERRORS**

</div>

2) Short messages may use more elaborate schemes. For example, the yes/no response to the terminal could be coded so as to provide error detection and correction capability.

3) An acknowledgement and a limited number of retries could trigger an indicator that would inform the operator of a communications failure (inability to transfer a message without errors) or central computer failure (no response at all to the message within a certain period of time). Such a scheme, along with the Lamp Test, would allow simple failure diagnosis.

A communications or central computer failure indicator should also "unlock" the terminal, i.e., allow it to accept another entry. This is necessary if the terminal will not accept entries while a verification is in progress. The terminal may also unlock after a certain maximum time delay. Certain entries could be reserved for diagnostics; i.e., certain credit card numbers could be used to check the internal operation of the terminal and test the displays.

REVIEW OF PROBLEM DEFINITION

Problem definition is as important a part of software development as it is of any other engineering task. Note that it does not require any programming or knowledge of the computer; rather, it is based on an understanding of the system and sound engineering judgment. Microprocessors can offer flexibility that the designer can use to provide a range of features which were not previously available.

Problem definition is independent of any particular computer, computer language, or development system. It should, however, provide guidelines as to what type or speed of computer the application will require and what kind of hardware/software trade-offs the designer can make. The problem definition stage is in fact independent of whether or not a computer is used at all, although a knowledge of the capabilities of the computer can help the designer in suggesting possible implementations of procedures.

PROGRAM DESIGN

Program design is the stage in which the problem definition is formulated as a program. If the program is small and simple, this stage may involve little more than the writing of a one-page flowchart. If the program is larger or more complex, the designer should consider more elaborate methods

We will discuss flowcharting, modular programming, structured programming, and top-down design. We will try to indicate the reasoning behind these methods, and their advantages and disadvantages. We will not, however, advocate any particular method since there is no evidence that one method is always superior to all others. You should remember that the goal is to produce a good working system, not to follow religiously the tenets of one methodology or another.

All the methodologies do, however, have some obvious principles in common. Many of these are the same principles that apply to any kind of design, such as:

| BASIC |
| PRINCIPLES |
| OF PROGRAM |
| DESIGN |

1) Proceed in small steps. Do not try to do too much at one time.

2) Divide large jobs into small, logically separate tasks. Make the sub-tasks as independent of one another as possible, so that they can be tested separately and so that changes can be made in one without affecting the others.

3) Keep the flow of control as simple as possible so as to make it easier to find errors.

4) Use pictorial or graphic descriptions as much as possible. They are easier to visualize than word descriptions. This is the great advantage of flowcharts.

5) Emphasize clarity and simplicity at first. You can improve performance (if necessary) once the system is working.

6) Proceed in a thorough and systematic manner. Use checklists and standard procedures.

7) Do not tempt fate. Either do not use methods that you are not sure of, or use them very carefully. Watch for situations that might cause confusion, and clarify them as soon as possible.

8) Keep in mind that the system must be debugged, tested and maintained. Plan for these later stages.

9) Use simple and consistent terminology and methods. Repetitiveness is no fault in program design, nor is complexity a virtue.

10) Have your design completely formulated before you start coding. Resist the temptation to start writing down instructions: it makes no more sense than making parts lists or laying out circuit boards before you know exactly what will be in the system.

11) Be particularly careful of factors that may change. Make the implementation of likely changes as simple as possible.

⏢	Input/Output
▭	Processing operation (Arithmetic, Logic, Data Movement)
◇	Decision logic
⬡	Subroutine
○	Connector point
→	Connector arrows
⬭	Terminal point (Beginning or Ending)
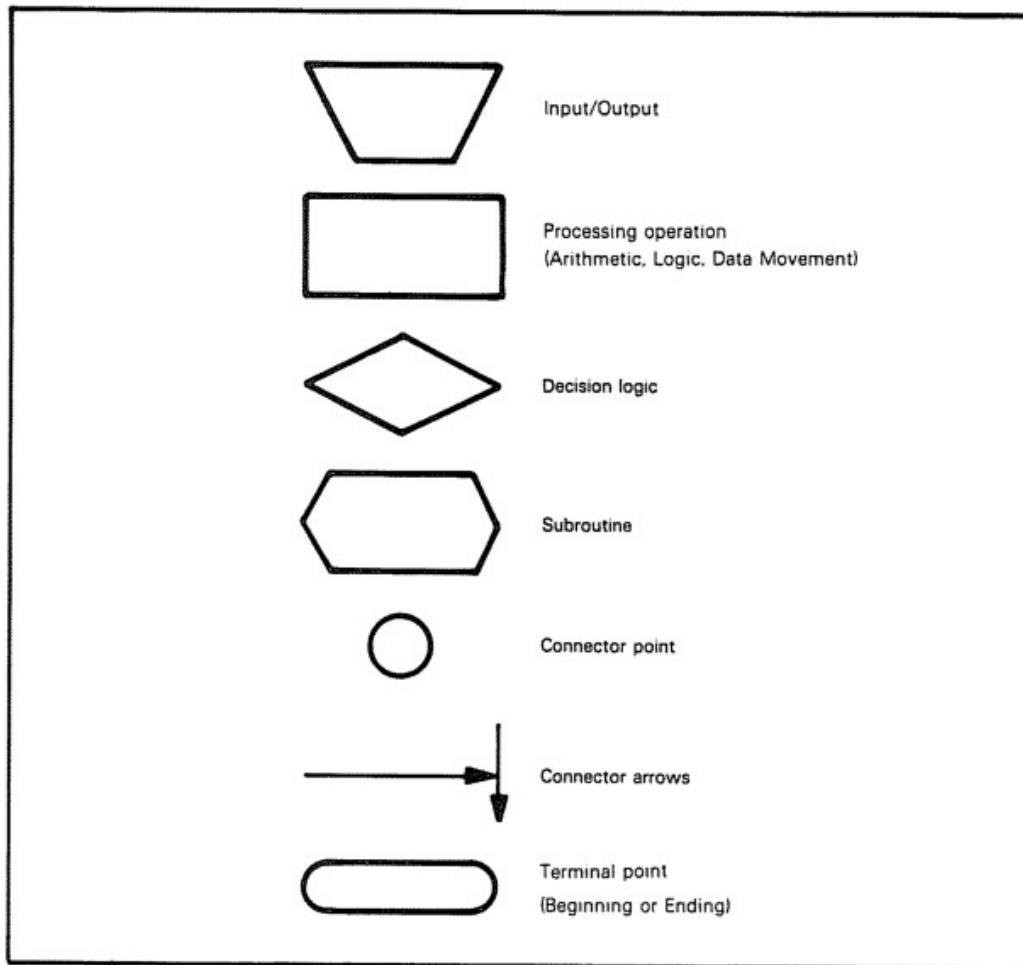
Figure 13-7. Standard Flowchart Symbols

FLOWCHARTING

Flowcharting is certainly the best-known of all program design methods. Programming textbooks describe how programmers first write complete flowcharts and then start writing the actual program. In fact, few programmers have ever worked this way, and flowcharting has often been more of a joke or a nuisance to programmers than a design method. We will try to describe both the advantages and disadvantages of flowcharts, and show the place of this technique in program design.

The basic advantage of the flowchart is that it is a pictorial representation. People find such representations much more meaningful than written descriptions. The designer can visual- | ADVANTAGES OF FLOWCHARTING
ize the whole system and see the relationships of the various parts. Logical errors and inconsistencies often stand out instead of being hidden in a printed page. **At its best, the flowchart is a picture of the entire system.**

Some of the more specific advantages of flowcharts are:

1) Standard symbols exist (see Figure 13-7) so that flowcharting forms are widely recognized.

2) Flowcharts can be understood by someone without a programming background.

3) Flowcharts can be used to divide the entire project into sub-tasks. The flowchart can then be examined to measure overall progress.

4) Flowcharts show the sequence of operations and can therefore aid in locating the source of errors.

5) Flowcharting is widely used in other areas besides programming.

6) There are many tools available to aid in flowcharting, including programmer's templates and automated drawing packages.

These advantages are all important. There is no question that flowcharting will continue to be widely used. But **we should note some of the disadvantages of flowcharting as a program design method, e.g.:**

DISADVANTAGES OF FLOWCHARTING

1) Flowcharts are difficult to design, draw, or change in all except the simplest situations.

2) There is no easy way to debug or test a flowchart.

3) Flowcharts tend to become cluttered. Designers find it difficult to balance between the amount of detail needed to make the flowchart useful and the amount that makes the flowchart little better than a program listing.

4) Flowcharts show only the program organization. They do not show the organization of the data or the structure of the input/output modules.

5) Flowcharts do not help with hardware or timing problems or give hints as to where these problems might occur.

6) Flowcharts allow for highly unstructured design. Lines and arrows backtracking and looping all over the chart are the antithesis of good structured design principles.

Thus, **flowcharting is a helpful technique** that you should not try to extend too far. **Flowcharts are useful as program documentation, since they have standard forms and are comprehensible to non-programmers.** As a design tool, **however,** flowcharts cannot provide much more than a starting outline; **the programmer cannot debug a detailed flowchart** and the flowchart is often more difficult to design than the program itself.

EXAMPLES

Response to a Switch

This simple task, in which a single switch turns on a light for one second, is easy to flowchart. In fact, such tasks are typical examples for flowcharting books, although they form a small part of most systems. The data structure here is so simple that it can be safely ignored.

> **FLOWCHARTING SWITCH AND LIGHT SYSTEM**

Figure 13-8 is the flowchart. There is little difficulty in deciding on the amount of detail required. The flowchart gives a straightforward picture of the procedure, which anyone could understand.

Note that the most useful flowcharts may ignore program variables and ask questions directly. Of course, compromises are often necessary here. **Two versions of the flowchart are sometimes helpful — one general version in layman's language, which will be useful to non-programmers, and one programmer's version in terms of the program variables, which will be useful to other programmers.**

A third type of flowchart, a data flowchart, may also be helpful. This flowchart serves as a cross-reference for the other flowcharts, since it shows how the program handles a particular

> **DATA FLOWCHARTS**

type of data. Ordinary flowcharts show how the program proceeds, handling different types of data at different points. Data flowcharts, on the other hand, show how particular types of data move through the system, passing from one part of the program to another. Such flowcharts are very useful in debugging and maintenance, since errors most often show up as a particular type of data being handled incorrectly.
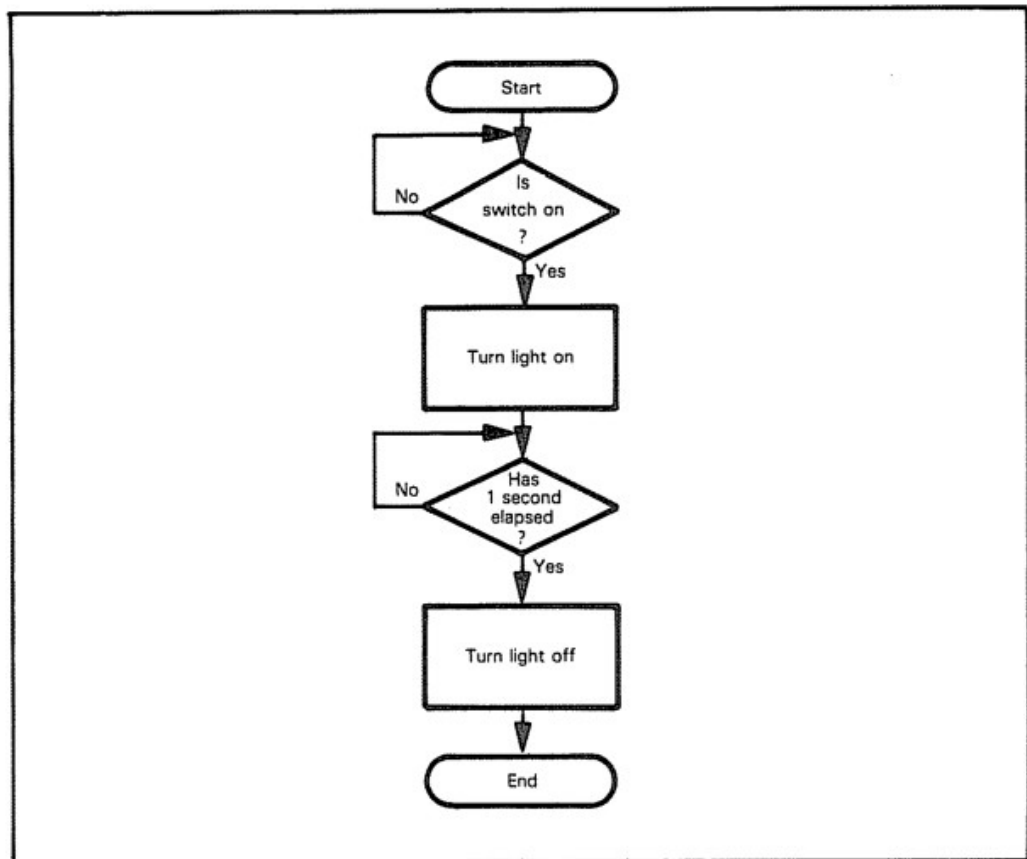
Figure 13-8. Flowchart of One-Second Response to a Switch

The Switch-Based Memory Loader

This system (see Figure 13-3) is considerably more complex than the previous example, and involves many more decisions. **The flowchart (see Figure 13-9) is more difficult to write and not as straightforward as the previous example.** In this example, we face the problem that there is no way to debug or test the flowchart.

FLOWCHARTING THE SWITCH-BASED MEMORY LOADER

The flowchart in Figure 13-9 includes the improvements we suggested as part of the problem definition. Clearly, **this flowchart is beginning to get cluttered and lose its advantages over a written description.** Adding other features that define the meaning of the entry with status lights and allow the operator to check entries after completion would make the flowchart even more complex. Writing the complete flowchart from scratch could quickly become a formidable task. However, once the program has been written, the flowchart is useful as documentation.
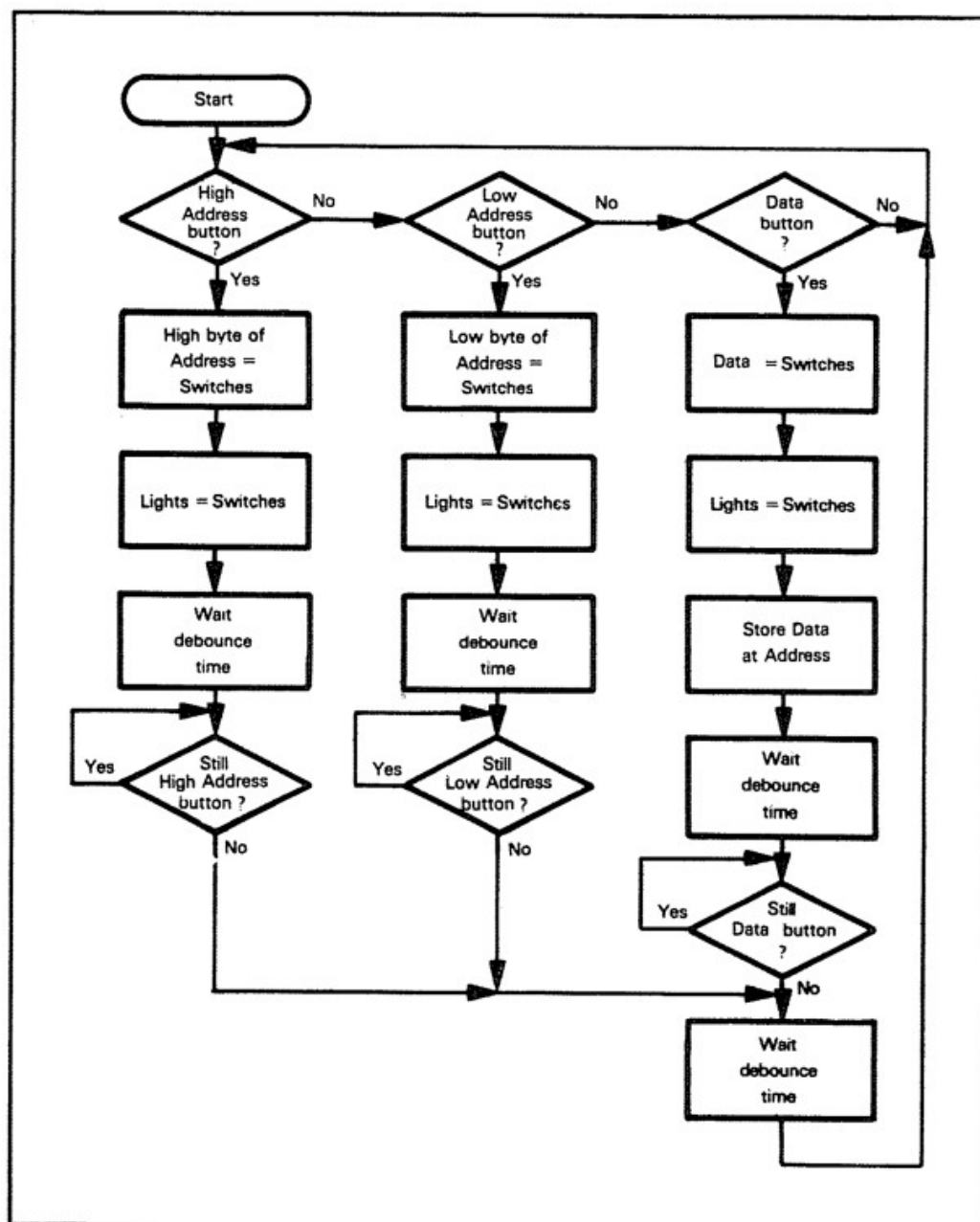
Figure 13-9. Flowchart of Switch-Based Memory Loader
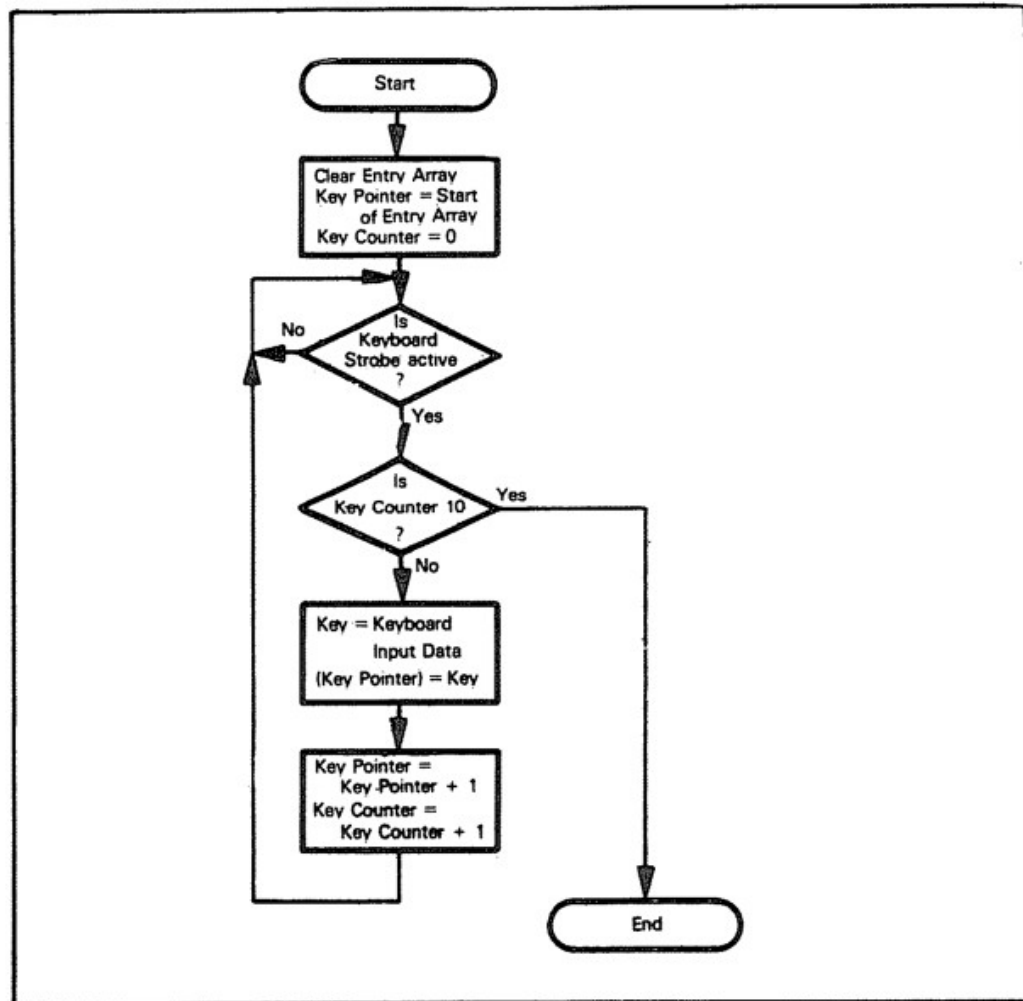
Figure 13-10. Flowchart of Keyboard Entry Process

The Credit-Verification Terminal

In this application (see Figures 13-4 through 13-6), the flowchart will be even more complex than in the switch-based memory loader case. Here, **the best idea is to flowchart sections separately so that the flowcharts remain manageable.** However, the presence of data structures (as in the multi-digit display and the messages) will make the gap between flowchart and program much wider.

Let us look at some of the sections. **Figure 13-10 shows the keyboard entry process for the digit keys.** The program must fetch the data after each strobe and place the digit into the display array if there is room for it. If there are already ten digits in the array, the program simply ignores the entry.

The actual program will have to handle the displays at the same time. Note that either software or hardware must de-activate the keyboard strobe after the processor reads a digit.

13-22
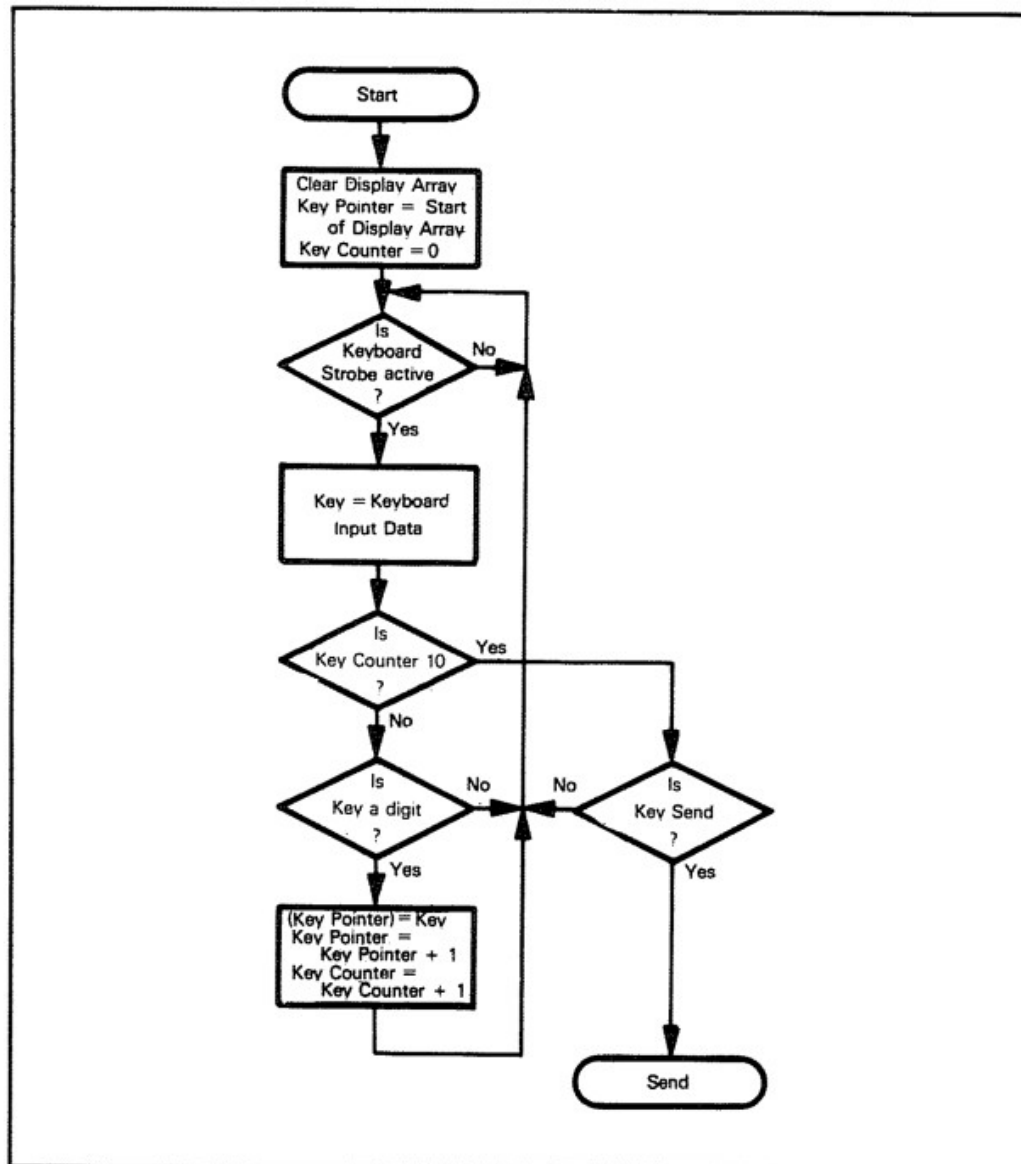
Figure 13-11. Flowchart of Keyboard Entry Process with Send Key

Figure 13-11 adds the Send key. This key, of course, is optional. The terminal could just send the data as soon as the operator enters a complete number. However, that procedure would not give the operator a chance to check the entire entry. The flowchart with the Send key is more complex because there are two alternatives.

1) If the operator has not entered ten digits, the program must ignore the Send key and place any other key into the entry.

2) If the operator has entered ten digits, the program must respond to the Send key by transferring control to the Send routine, and ignore all other keys.

Note that the flowchart has become much more difficult to organize and to follow. There is also no obvious way to check the flowchart.
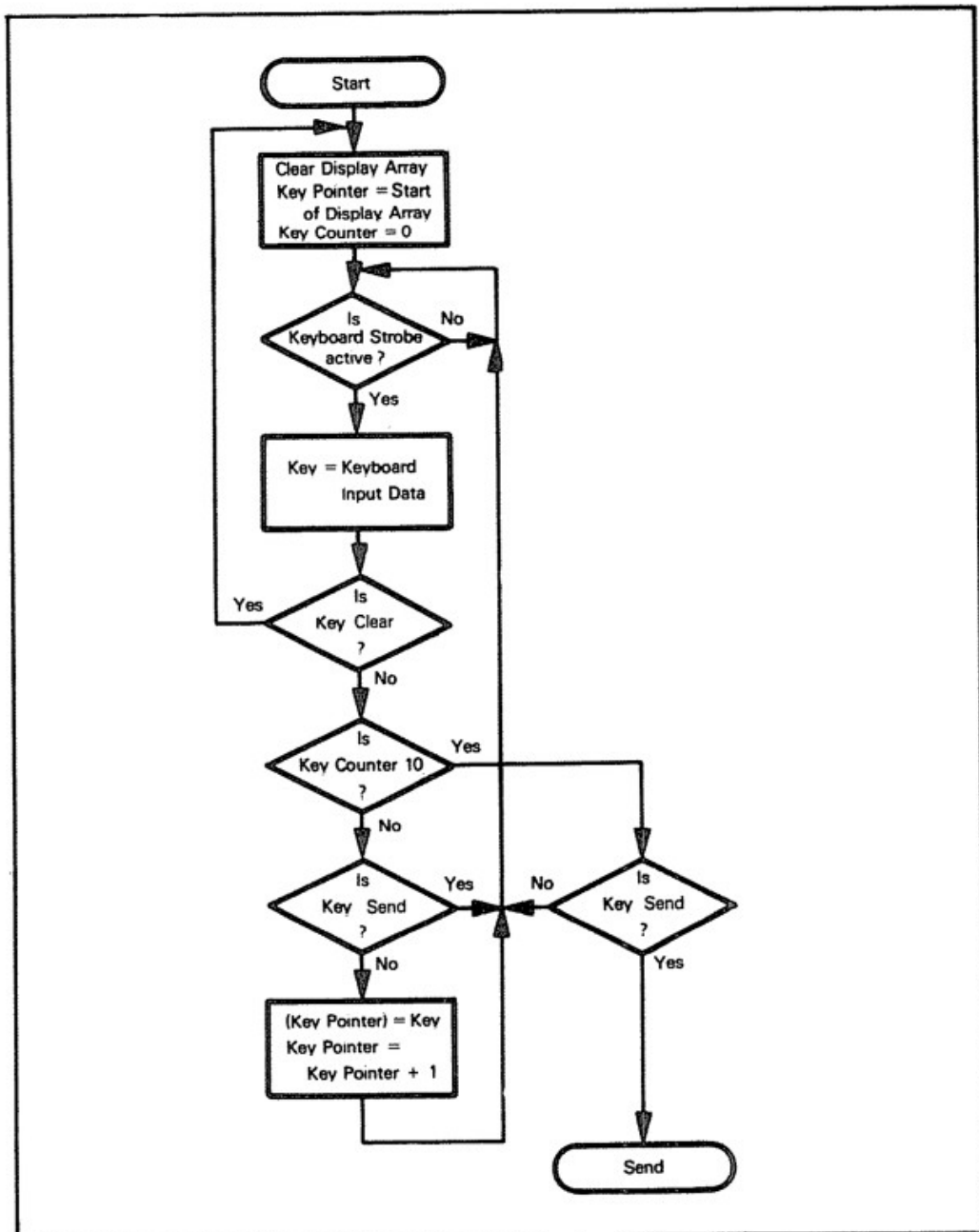
Figure 13-12. Flowchart of Keyboard Entry Process with Function Keys

Figure 13-12 shows the flowchart of the keyboard entry process with all the function keys. In this example, the flow of control is not simple. Clearly, some written description is necessary. The organization and layout of complex flowcharts requires careful planning. We have followed the process of adding features to the flowchart one at a time, but this still results in a large amount of redrawing. Again we should remember that throughout the keyboard entry process, the program must also refresh the displays if they are multiplexed and not controlled by shift registers or other hardware.

13-24
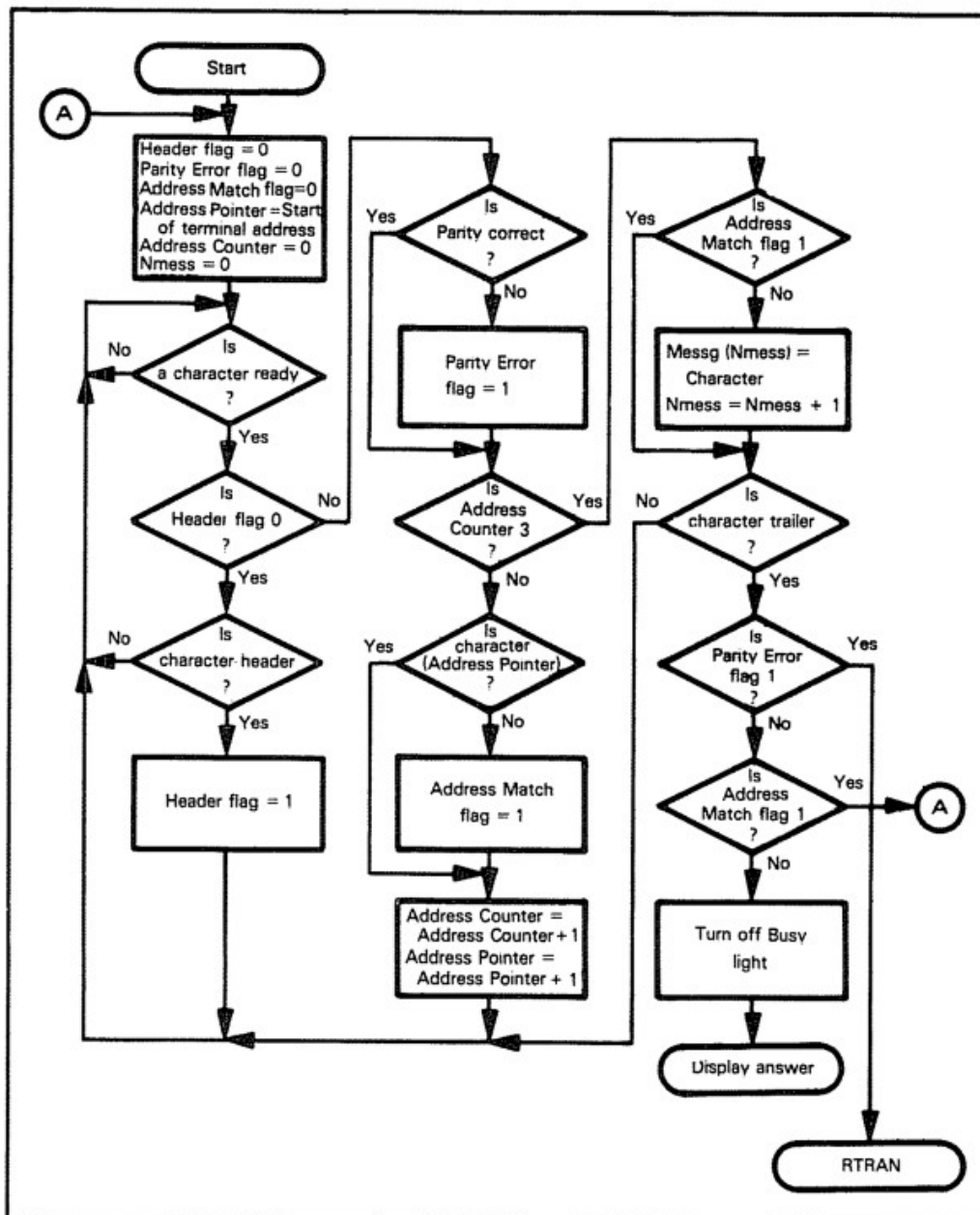
Figure 13-13. Flowchart of Receive Routine

13-25

Figure 13-13 is the flowchart of a receive routine. We assume that the serial/parallel conversion and error checking are done in hardware (e.g., by a UART). The processor must:

1) Look for the header (we assume that it is a single character).

2) Read the destination address (we assume that it is three characters long) and see if the message is meant for this terminal; i.e., if the three characters agree with the terminal address.

3) Wait for the trailer character.

4) If the message is meant for the terminal, turn off the Busy light and go to Display Answer routine.

5) In the event of any errors, request retransmission by going to RTRAN routine.

This routine involves a large number of decisions, and the flowchart is neither simple nor obvious.

Clearly, we have come a long way from the simple flowchart (Figure 13-8) of the first example. A complete set of flowcharts for the transaction terminal would be a major task. It would consist of several interrelated charts with complex logic, and would require a large amount of effort. Such an effort would be just as difficult as writing a preliminary program, and not as useful, since you could not check it on the computer.

MODULAR PROGRAMMING

Once programs become large and complex, flowcharting is no longer a satisfactory design tool. However, the problem definition and the flowchart can give you some idea as to how to divide the program into reasonable sub-tasks. **The division of the entire program into sub-tasks or modules is called "modular programming."** Clearly, most of the programs we presented in earlier chapters would typically be modules in a large system program. **The problems that the designer faces in modular programming are how to divide the program into modules and how to put the modules together.**

The advantages of modular programming are obvious:

> **ADVANTAGES OF MODULAR PROGRAMMING**

1) A single module is easier to write, debug, and test than an entire program.

2) A module is likely to be useful in many places and in other programs, particularly if it is reasonably general and performs a common task. You can build up a library of standard modules.

3) Modular programming allows the programmer to divide tasks and use previously written programs.

4) Changes can be incorporated into one module rather than into the entire system.

5) Errors can often be isolated and then attributed to a single module.

6) Modular programming gives an idea of how much progress has been made and how much of the work is left.

The idea of modular programming is such an obvious one that its disadvantages are often ignored. These include:

1) Fitting the modules together can be a major problem, particularly if different people write the modules.

2) Modules require very careful documentation, since they may affect other parts of the program, such as data structures used by all the modules.

3) Testing and debugging modules separately is difficult, since other modules may produce the data used by the module being debugged and still other modules may use the results. You may have to write special programs (called "drivers") just to produce sample data and test the programs. These drivers require extra programming effort that adds nothing to the system.

4) Programs may be very difficult to modularize. If you modularize the program poorly, integration will be very difficult, since almost all errors and changes will involve several modules.

5) Modular programs often require extra time and memory, since the separate modules may repeat functions.

Therefore, while modular programming is certainly an improvement over trying to write the entire program from scratch, it does have some disadvantages as well.

Important considerations include restricting the amount of information shared by modules, limiting design decisions that are subject to change to a single module and restricting the access of one module to another.[3]

An obvious problem is that there are no proven, systematic methods for modularizing programs. We should mention the following principles:[4]

PRINCIPLES OF MODULARIZATION

1) Modules that reference common data should be parts of the same overall module.

2) Two modules in which the first uses or depends on the second, but not the reverse, should be separate.

3) A module that is used by more than one other module should be part of a different overall module than the others.

4) Two modules in which the first is used by many other modules and the second is used by only a few other modules should be separate.

5) Two modules whose frequencies of usage are significantly different should be part of different modules.

6) The structure or organization of related data should be hidden within a single module.

If you find it very difficult to modularize your program, it is a strong indication that the problem is poorly defined, and redefinition is called for. Too many special cases, each requiring special handling, or the use of a large number of variables, each requiring special processing, are problems that can be most efficiently handled, by redefining the tasks at hand.

EXAMPLES

Response to a Switch

This simple program can be divided into two modules:

Module 1 waits for the switch to be turned on and turns the light on in response.

Module 2 provides the one-second delay.

> MODULARIZING THE SWITCH AND LIGHT SYSTEM

Module 1 is likely to be specific to the system, since it will depend on how the switch and light are attached. Module 2 will be generally useful, since many tasks require delays. Clearly, it would be advantageous to have a standard delay module that could provide delays of varying lengths. The module will require careful documentation so that you will know how to specify the length of the delay, how to call the module, and what registers and memory locations the module affects.

A general version of Module 1 would be far less useful, since it would have to deal with different types and connections of switches and lights.

You would probably find it simpler to write a module for a particular configuration of switches and lights rather than try to use a standard routine. Note the difference between this situation and Module 2.

The Switch-Based Memory Loader

The switch-based memory loader is difficult to modularize, since all the programming tasks depend on the hardware configuration and the tasks are so simple that modules hardly seem worthwhile. The flowchart in Figure 13-9 suggests that one module might be the one that waits for the operator to press one of the three pushbuttons.

> MODULARIZING THE SWITCH- BASED MEMORY LOADER

Some other modules might be:

- A delay module that provides the delay required to debounce the switches
- A switch and display module that reads the data from the switches and sends it to the displays
- A Lamp Test module

Highly system-dependent modules such as the last two are unlikely to be generally useful. This example is not one in which modular programming offers great advantages.

The Verification Terminal

The verification terminal, on the other hand, lends itself very well to modular programming. The entire system **can easily be divided into three main modules:**

> MODULARIZING THE VERIFICATION TERMINAL

- **Keyboard and display module**
- **Data transmission module**
- **Data reception module**

A general keyboard and display module could handle many keyboard- and display-based systems. The sub-modules would perform such tasks as:

- Recognizing a new keyboard entry and fetching the data
- Clearing the array in response to a Clear key
- Entering digits into storage
- Looking for the terminator or Send key
- Displaying the digits

Although the key interpretations and the number of digits will vary, the basic entry, data storage, and data display processes will be the same for many programs. Such function keys as Clear would also be standard. Clearly, **the designer must consider which modules will be useful in other applications, and pay careful attention to those modules.**

The data transmission module could also be divided into such sub-modules as:

1) Adding the header character.
2) Transmitting characters as the output line can handle them.
3) Generating delay times between bits or characters.
4) Adding the trailer character.
5) Checking for transmission failures; i.e., no acknowledgement or inability to transmit without errors.

The data reception module could include sub-modules which:

1) Look for the header character.
2) Check the message destination address against the terminal address.
3) Store and interpret the message.
4) Look for the trailer character.
5) Generate bit or character delays.

Note here how important it is that each design decision (such as the bit rate, message format, or error-checking procedure) be implemented in only one module. A change in any of these decisions will then require changes only to that single module. The other

> **INFORMATION HIDING PRINCIPLE**

modules should be written so that they are totally unaware of the values chosen or the methods used in the implementing module. **An important concept here is the "information-hiding principle,"[5] whereby modules share only information that is absolutely essential to getting the task done. Other information is hidden within a single module.**

An important use of this principle is in error handling. Whenever a module detects a lethal error, it should not undertake recovery procedures. Instead, it should pass the error status back up to the calling module and allow it to make the decision of how to recover from the error. The reason for this is that the lower level procedure often does not have enough information to adequately decide what recovery procedures are necessary. For example, suppose we have a module that accepts numeric input from a user. This module terminates normally when the user enters a string of numeric digits terminated by a carriage return. Entry of any non-numeric characters causes the module to immediately terminate abnormally. Since the module does not know in what context it is being used (i.e., is it part of an assembler, an interactive editor, or a file management system?) it cannot make a valid decision of what action to take when encountering an invalid character. If a single error recovery method was designed into the module, it would lose its generality and become specific to those situations that employ this error recovery technique.

REVIEW OF MODULAR PROGRAMMING

Modular programming can be very helpful if you abide by the following rules:

1) **Use modules of 20 to 50 lines.** Shorter modules are usually a waste of time, while longer modules are seldom general and may be difficult to integrate.

2) **Try to make modules reasonably general.** Differentiate between common features like ASCII code or asynchronous transmission formats, which will be the same for many applications and key identifications, and number of displays or number of characters in a message, which are likely to be unique to a particular application. Make the changing of the latter parameters simple. Major changes like different character codes should be handled by separate modules.

3) **Take extra time on modules** like delays, display handlers, keyboard handlers, etc. **that will be useful in other projects or in many different places in the present program.**

4) **Try to keep modules as distinct and logically separate as possible.** Restrict the flow of information between modules and implement each design decision in a single module.

5) **Do not try to modularize simple tasks** where rewriting the entire task may be easier than assembling or modifying the module.

STRUCTURED PROGRAMMING

How do you keep modules distinct and stop them from interacting? How do you write a program that has a clear sequence of operations so that you can isolate and correct errors? One answer is to use the methods known as "structured programming", whereby each part of the program consists of elements from a limited set of structures and each structure has a single entry and a single exit.

Figure 13-14 shows a flowchart of an unstructured program. If an error occurs in Module B, we have five possible sources for that error. Not only must we check each sequence, but we also have to make sure that any changes made to correct the error do not affect any of the other sequences. The usual result is that debugging becomes like wrestling an octopus. Every time you think the situation is under control, there is another loose tentacle somewhere.
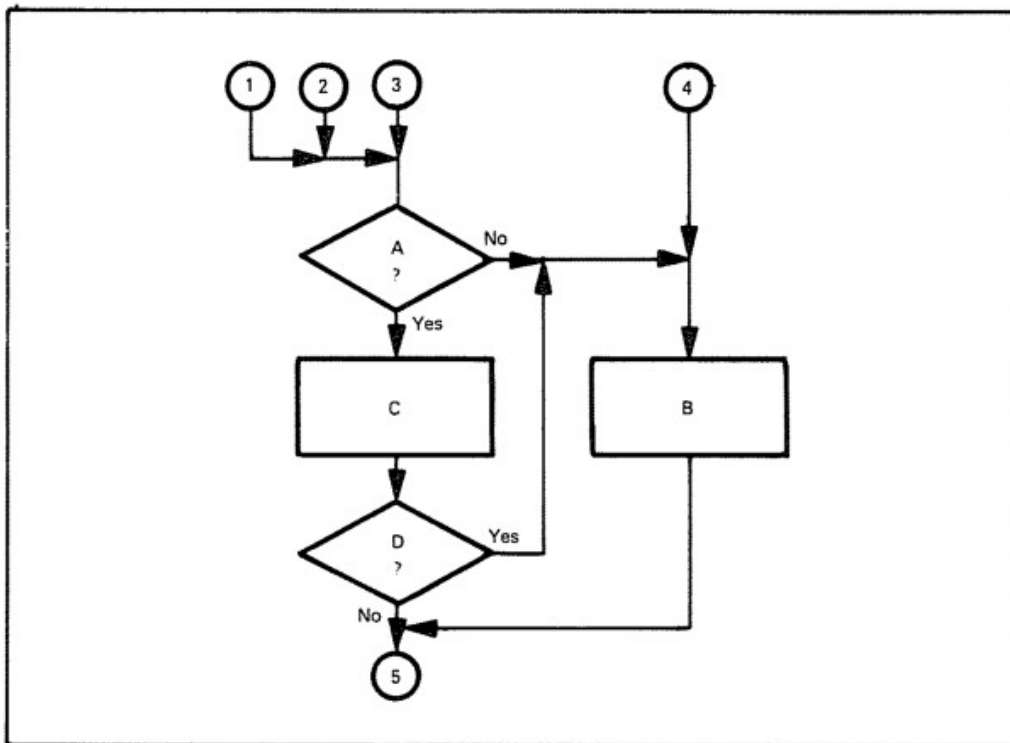
Figure 13-14. Flowchart of an Unstructured Program

The solution is to establish a clear sequence of operations so that you can isolate errors. Such a sequence uses single-entry, single-exit modules. **The basic modules that are needed are:**

1) **An ordinary sequence;** i.e., a linear structure in which statements or structures are executed consecutively. In the sequence:

> S1
> S2
> S3

the computer executes S1 first, S2 second, and S3 third. S1, S2, and S3 may be single instructions or entire programs.

2) **A conditional structure.**

The common one is "if C then S1 else S2," where C is a condition and S1 and S2 are statements or sequences of statements. The computer executes S1 if C is true, and S2 if C is false. Figure 13-15 shows the logic of this structure. Note that the structure has a single entry and a single exit; there is no way to enter or leave S1 or S2 other than through the structure.

3) **A loop structure.**

The common loop structure is "while C do S." where C is a condition and S is a statement or sequence of statements. The computer checks C and executes S if C is true. This structure (see Figure 13-16) also has a single entry and a single exit. Note that the computer will not execute S at all if C is originally false, since the value of C is checked before S is executed.
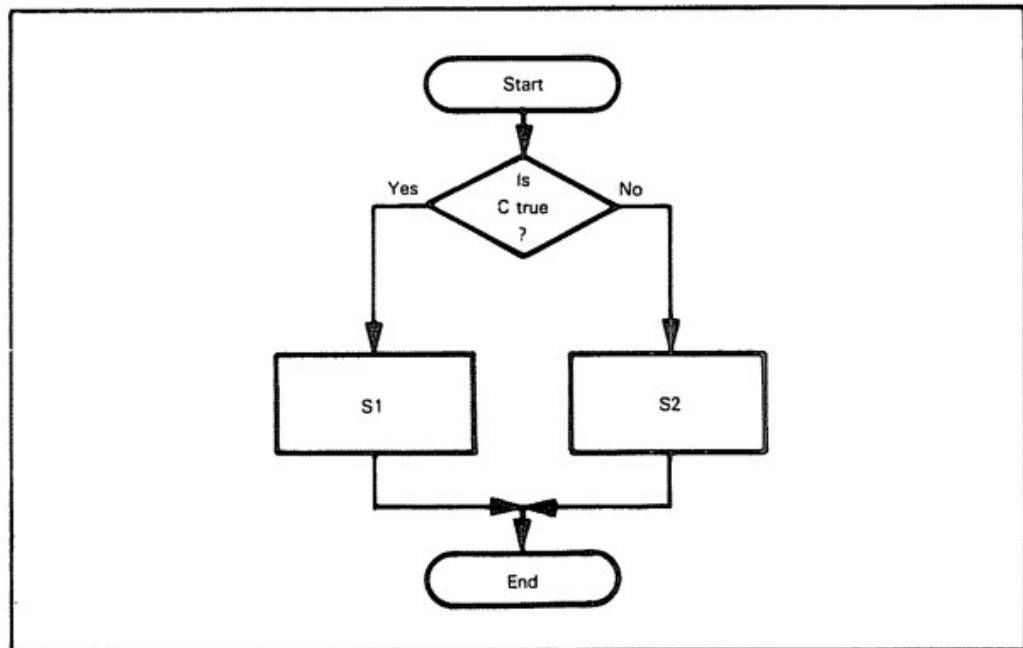
Figure 13-15. Flowchart of the If-Then-Else Structure
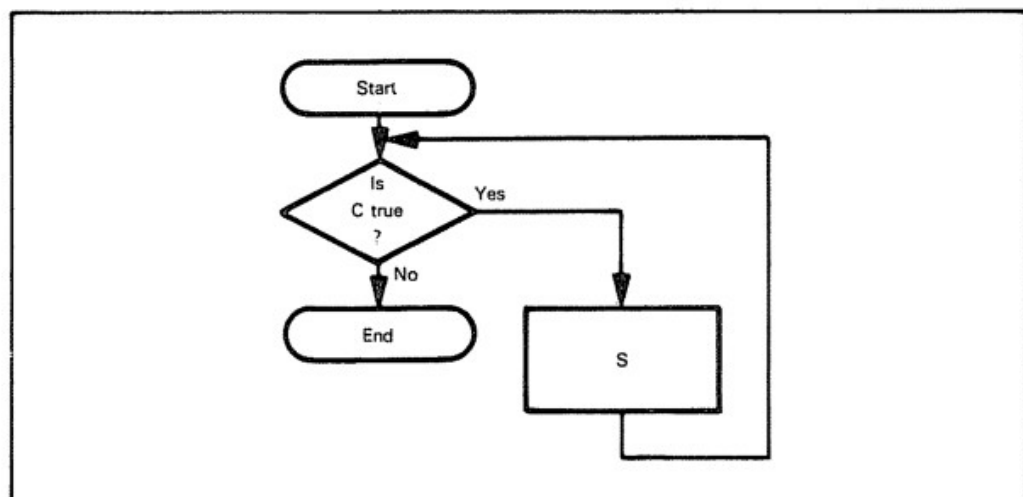
Figure 13-16. Flowchart of the Do-While Structure

In most structured programming languages, an alternative looping construct is provided. This construct is known as the do-until clause, Its basic structure is "do S until C", where C is a condition and S is a statement or sequence of statements. It is similar to the do-while construct except that the test of the looping condition C is performed at the end of the loop. This has the effect of guaranteeing that the loop is always executed at least once. This is illustrated by the flowchart in Figure 13-17. The common index-controlled or DO loop can be implemented as a special case of either of these two basic looping constructs.
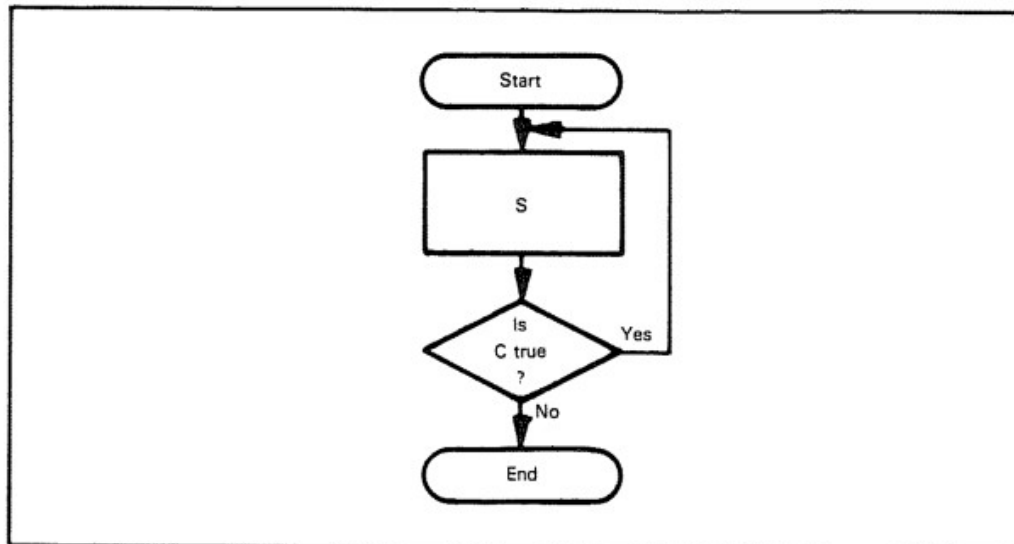
Figure 13-17. Flowchart of the Do-Until Structure

4) **A case structure.**

Although not a primitive structure like sequential, if-then-else, and do-while, the case structure is so commonly used that we include it here as an adjunct to the basic structure descriptions. The case structure is "case I of S0, S1. . . .Sn", where I is an index and S0, S1, . . .Sn are statements or sequences of statements. If I is equal to zero then statement S0 is executed; if I is equal to 1 then statement S1 is executed, etc. Only one of the n statements is executed. After its execution, control passes to the next sequential statement following the case statement group. If I is greater than n (i.e., the number of statements in the case statement), then none of the statements in the case statement is executed, and control is passed directly to the next sequential statement following the case statement. This is illustrated by the flowchart in Figure 13-18.

Note the following features of structured programming:

1) **Only the three basic structures, and possibly a small number of auxiliary structures, are permitted.**

2) **Structures may be nested to any level of complexity so that any program can, in turn, contain any of the structures.**

3) **Each structure has a single entry and a single exit.**

Some examples of the conditional structure illustrated in Figure 13-15 are:

> EXAMPLES
> OF
> STRUCTURES

1) S2 included:

   if $X \geq 0$ then NPOS = NPOS + 1
   else NNEG = NNEG + 1

Both S1 and S2 are single statements.

2) S2 omitted:

   if $X \neq 0$ then Y = 1/X

Here no action is taken if C ($X \neq 0$) is false. S2 and "else" can be omitted in this case.

13-33
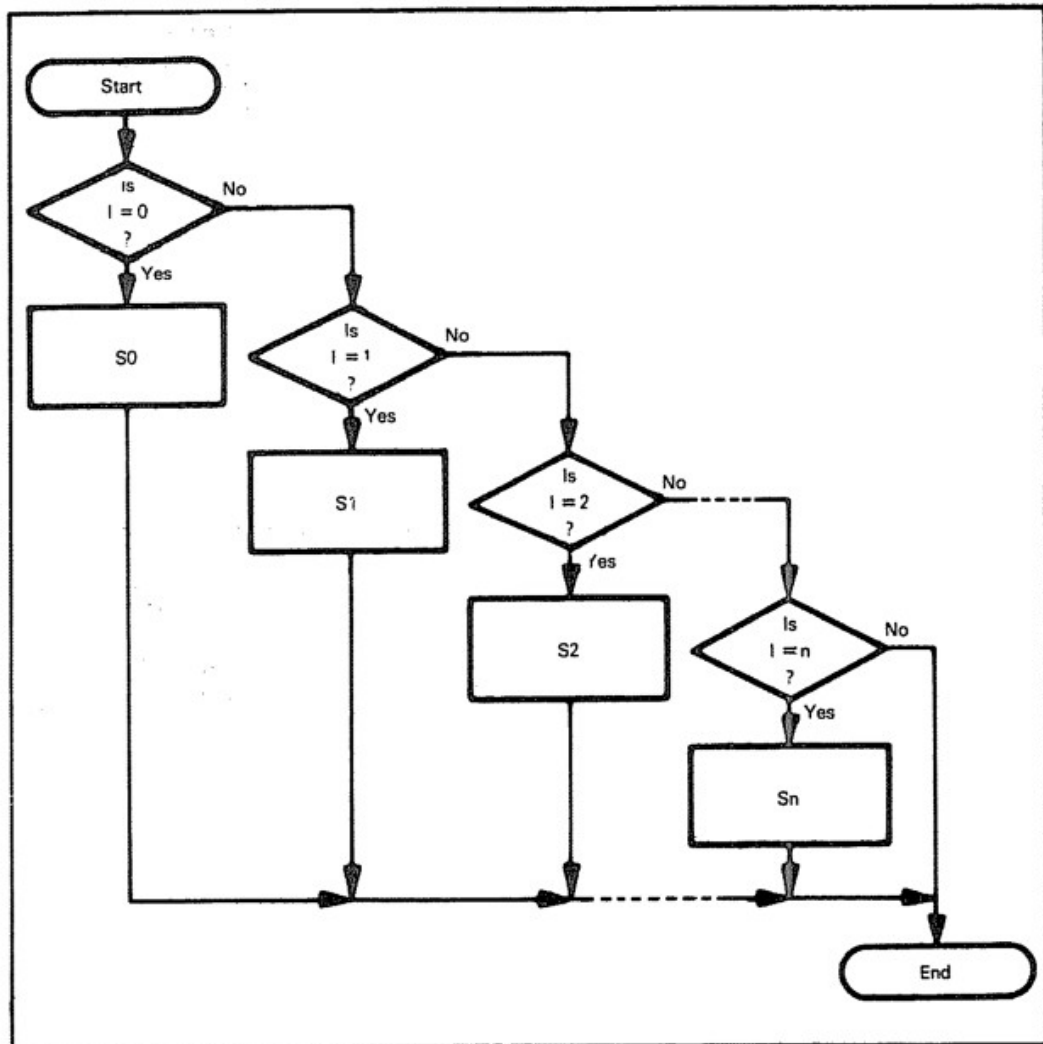
Figure 13-18. Flowchart of the Case Structure

Some examples of the loop structure illustrated in Figure 13-16 are:

1) Form the sum of integers from 1 to N.

    I = 0
    SUM = 0
    do while I < N
      I = I + 1
      SUM = SUM + I
    end

The computer executes the loop as long as I < N. If N = 0, the program within the "do-while" is not executed at all.

2) Count characters in an array SENTENCE until you find an ASCII period.

    NCHAR = 0
    do while SENTENCE (NCHAR) ≠ PERIOD
      NCHAR = NCHAR + 1
    end

The computer executes the loop as long as the character in SENTENCE is not an ASCII period. The count is zero if the first character is a period.

13-34

The advantages of structured programming are:

1) The sequence of operations is simple to trace. This allows you to test and debug easily.

2) The number of structures is limited and the terminology is standardized.

3) The structures can easily be made into modules.

4) Theoreticians have proved that the given set of structures is complete; that is, all programs can be written in terms of the three structures.

5) The structured version of a program is partly self-documenting and fairly easy to read.

6) Structured programs are easy to describe with program outlines.

7) Structured programming has been shown in practice to increase programmer productivity.

Structured programming basically forces much more discipline on the programmer than does modular programming. The result is more systematic and better-organized programs.

The disadvantages of structured programming are:

1) Only a few high-level languages (e.g., PL/M, PASCAL) will directly accept the structures. The programmer therefore has to go through an extra translation stage to convert the structures to assembly language code. The structured version of the program, however, is often useful as documentation.

2) Structured programs often execute more slowly and use more memory than unstructured programs.

3) Limiting the structures to the four basic forms makes some tasks very awkward to perform. The completeness of the structures only means that all programs can be implemented with them; it does not mean that a given program can be implemented efficiently or conveniently.

4) The standard structures are often quite confusing, e.g., nested "if-then-else" structures may be very difficult to read, since there may be no clear indication of where the inner structures end. A series of nested "do-while" loops can also be difficult to read.

5) Structured programs consider only the sequence of program operations, not the flow of data. Therefore, the structures may handle data awkwardly.

6) Few programmers are accustomed to structured programming. Many find the standard structures awkward and restrictive.

We are neither advocating nor discouraging the use of structured programming. It is one way of systematizing program design. In general, structured programming is most useful in the following situations:

- Larger programs, perhaps exceeding 1000 instructions
- Applications in which memory usage is not critical.
- Low-volume applications where software development costs, particularly testing and debugging, are important factors.
- Applications involving string manipulation, process control, or other algorithms rather than simple bit manipulations.

> ADVANTAGES OF STRUCTURED PROGRAMMING

> DISADVANTAGES OF STRUCTURED PROGRAMMING

> WHEN TO USE STRUCTURED PROGRAMMING

In the future, we expect the cost of memory to decrease, the average size of microprocessor programs to increase, and the cost of software development to increase. Therefore, methods like structured programming, which decrease software development costs for larger programs but use more memory, will become more valuable.

Just because structured programming concepts are usually expressed in high-level languages does not mean that structured programming is not applicable to assembly language programming. On the contrary, the assembly language programmer, with the total freedom of expression that assembly level programming allows, needs the structuring concepts provided by structured programming. Creating modules with single entry and exit points, using simple control structures and keeping the complexity of each module minimal makes assembly language coding more efficient.

EXAMPLES

Response to a Switch

The structured version of this example is:

<div style="float:right; border:2px solid black; padding:4px; font-weight:bold;">
STRUCTURED
PROGRAMMING
IN THE
SWITCH AND
LIGHT SYSTEM
</div>

```
SWITCH = OFF
do while SWITCH = OFF
   READ SWITCH
end
LIGHT = ON
DELAY 1
LIGHT = OFF
```

ON and OFF must have the proper definitions for the switch and light. We assume that DELAY is a module that provides a delay given by its parameter in seconds.

A statement in a structured program may actually be a subroutine. However, in order to conform to the rules of structured programming, the subroutine cannot have any exits other than the one that returns control to the main program.

Since "do-while" checks the condition before executing the loop, we set the variable SWITCH to OFF before starting. The structured program is straightforward, readable, and easy to check by hand. However, it would probably require somewhat more memory than an unstructured program, which would not have to initialize SWITCH and could combine the reading and checking procedures.

The Switch-Based Memory Loader

The switch-based memory loader is a more complex structured programming problem. We may implement the flowchart of Figure 13-9 as follows (an · indicates a comment):

<div style="float:right; border:2px solid black; padding:4px; font-weight:bold;">
STRUCTURED
PROGRAMMING
FOR THE
SWITCH-BASED
MEMORY LOADER
</div>

```
·
· INITIALIZE VARIABLES
·
HIADDRESS = 0
LOADDRESS = 0
·
· THIS PROGRAM USES A DO-WHILE CONSTRUCT WITH NO CONDITION
· (CALLED SIMPLY DO-FOREVER). THEREFORE, THE SYSTEM CONTINUALLY
· EXECUTES THE PROGRAM CONTAINED IN THIS DO-WHILE LOOP.
·
do forever
```

```
•
• TEST FOR HIADDRESS BUTTON; PERFORM THE REQUIRED PROCESSING
• IF IT IS ON.
•
    if HIADDRBUTTON = 1 then
      begin
        HIADDRESS = SWITCHES
        LIGHTS = SWITCHES
        do
          DELAY (DEBOUNCE TIME)
        until HIADDRBUTTON ≠ 1
      end
•
• TEST FOR LOADDRESS BUTTON; PERFORM LOW ADDRESS PROCESSING
• IF IT IS ON.
•
    if LOADDRBUTTON = 1 then
      begin
        LOADDRESS = SWITCHES
        LIGHTS = SWITCHES
        do
          DELAY (DEBOUNCE TIME)
        until LOADDRBUTTON ≠ 1
      end
•
• TEST FOR DATABUTTON, AND STORE DATA INTO MEMORY
• IF IT IS ON.
•
    if DATABUTTON = 1 then
      begin
        DATA = SWITCHES
        LIGHTS = SWITCHES
        (HIADDRESS, LOADDRESS) = DATA
        do
          DELAY (DEBOUNCE TIME)
        until DATABUTTON ≠ 1
      end
end
•
• THE LAST END ABOVE TERMINATES THE
•      do forever LOOP
•
```

Structured programs are not easy to write, but they can give a great deal of insight into the overall program logic. You can check the logic of the structured program by hand before writing any actual code.

The Credit-Verification Terminal

Let us look at the keyboard entry for the transaction terminal.
We will assume that the display array is ENTRY, the keyboard
strobe is KEYSTROBE, and the keyboard data is KEYIN. **The struc-
tured program without the function keys is:**

**STRUCTURED
PROGRAM FOR
THE CREDIT-
VERIFICATION
TERMINAL**

**STRUCTURED
KEYBOARD
ROUTINE**

```
NKEYS = 10
.
. CLEAR ENTRY TO START
.
  do while NKEYS > 0
    NKEYS = NKEYS - 1
    ENTRY (NKEYS) = 0        .
  end
.
. FETCH A COMPLETE ENTRY FROM KEYBOARD
.
  do while NKEYS < 10
    if KEYSTROBE = ACTIVE then
      begin
        KEYSTROBE = INACTIVE
        ENTRY (NKEYS) = KEYIN
        NKEYS = NKEYS + 1
      end
  end
```

**Adding the SEND key means that the program must ignore extra digits after it has
a complete entry, and must ignore the SEND key until it has a complete entry. The
structured program is:**

```
NKEYS = 10
.
. CLEAR ENTRY TO START
.
  do while NKEYS > 0
    NKEYS = NKEYS - 1
    ENTRY (NKEYS) = 0
  end
.
. WAIT FOR COMPLETE ENTRY FOLLOWED BY SEND KEY
.
  do while KEY ≠ SEND or NKEYS ≠ 10
    if KEYSTROBE = ACTIVE then
      begin
        KEYSTROBE = INACTIVE
        KEY = KEYIN
        if NKEYS ≠ 10 and KEY ≠ SEND then
          begin
            ENTRY (NKEYS) = KEY
            NKEYS = NKEYS + 1
          end
      end
  end
```

13-38

Note the following features of this structured program.

1) The second if-then is nested within the first one, since keys are only entered after a strobe is recognized. If the second if-then were on the same level as the first, a single key could fill the entry, since its value would be entered into the array during each iteration of the do-while loop.

2) KEY need not be defined initially, since NKEYS is set to zero as part of the clearing of the entry.

Adding the CLEAR key allows the program to clear the entry originally by simulating the pressing of CLEAR; i.e., by setting NKEYS to 10 and KEY to CLEAR before starting. The structured program must also clear only digits that have previously been filled. **The new structured program is:**

```
•

• SIMULATE COMPLETE CLEARING

•

NKEYS = 10
KEY = CLEAR

•

• WAIT FOR COMPLETE ENTRY AND SEND KEY

•

do while KEY ≠ SEND or NKEYS ≠ 10

•

• CLEAR WHOLE ENTRY IF CLEAR KEY STRUCK

•

    if KEY = CLEAR then
      begin
        KEY = 0
        do while NKEYS > 0
          NKEYS = NKEYS - 1
          ENTRY(NKEYS) = 0
        end
      end

•

• GET DIGIT IF ENTRY INCOMPLETE

•

    if KEYSTROBE = ACTIVE then
      begin
        KEYSTROBE = INACTIVE
        KEY = KEYIN
        if KEY < 10 and NKEYS ≠ 10 then
          begin
            ENTRY(NKEYS) = KEY
            NKEYS = NKEYS + 1
          end
      end
end
```

Note that the program resets KEY to zero after clearing the array, so that the operation is not repeated.

We can similarly build a structured program for the receive routine. An initial program could look just for the header and trailer characters. We will assume that RSTB is the indicator that a character is ready. **The structured program is:**

```
•
• CLEAR HEADER FLAG TO START
•
HFLAG = 0
•
• WAIT FOR HEADER AND TRAILER
•
do while HFLAG = 0 or CHAR ≠ TRAILER
•
• GET CHARACTER IF READY, LOOK FOR HEADER
•
    if RSTB = ACTIVE then
      begin
        RSTB = INACTIVE
        CHAR = INPUT
        if CHAR = HEADER then HFLAG = 1
      end
```

Now we can add the section that checks the message address against the three digits in TERMINAL ADDRESS (TERMADDR). If any of the corresponding digits are not equal, the ADDRESS MATCH flag (ADDRMATCH) is set to 1.

```
•
• CLEAR HEADER FLAG, ADDRESS MATCH FLAG, ADDRESS COUNTER TO START
•
HFLAG = 0
ADDRMATCH = 0
ADDRCTR = 0
•
• WAIT FOR HEADER, DESTINATION ADDRESS AND TRAILER
•
do while HFLAG = 0 or CHAR ≠ TRAILER OR ADDRCTR ≠ 3
•
• GET CHARACTER IF READY
•
    if RSTB = ACTIVE then
      begin
        RSTB = INACTIVE
        CHAR = INPUT
      end
•
• CHECK FOR TERMINAL ADDRESS AND HEADER
•
    if HFLAG = 1 and ADDRCTR ≠ 3 then
      begin
        ADDRMATCH = 1
        ADDRCTR = ADDRCTR + 1
      end
    if CHAR = HEADER then HFLAG = 1
end
```

The program must now wait for a header, a three-digit identification code, and a trailer. You must be careful of what happens during the iteration when the program finds the header, and of what happens if an erroneous identification code character is the same as the trailer.

A further addition can store the message in MESSG. NMESS is the number of characters in the message; if it is not zero at the end, the program knows that the terminal has received a valid message. We have not tried to minimize the logic expressions in this program.

```
•
• CLEAR FLAGS, COUNTERS TO START
•
HFLAG = 0
ADDRMATCH = 0
ADDRCTR = 0
NMESS = 0
•
• WAIT FOR HEADER, DESTINATION ADDRESS AND TRAILER
•
do while HFLAG = 0 or CHAR ≠ TRAILER or ADDRCTR ≠ 3
•
• GET CHARACTER IF READY
•
    if RSTB = ACTIVE then
      begin
        RSTB = INACTIVE
        CHAR = INPUT
      end
•
• READ MESSAGE IF DESTINATION ADDRESS = TERMINAL ADDRESS
•
    if HFLAG = 1 and ADDRCTR = 3 then
    if ADDRMATCH = 0 and CHAR ≠ TRAILER then
    begin
      MESSG(NMESS) = CHAR
      NMESS = NMESS + 1
    end
•
• CHECK FOR TERMINAL ADDRESS
•
    if HFLAG = 1 and ADDRCTR ≠ 3 then
    if CHAR ≠ TERMADDR(ADDRCTR) then
    begin
      ADDRMATCH = 1
      ADDRCTR = ADDRCTR + 1
    end
•
• LOOK FOR HEADER
•
    if CHAR = HEADER then HFLAG = 1
end
```

The program checks for the identification code only if it found a header during a previous iteration. It accepts the message only if it has previously found a header and a complete, matching destination address. The program must work properly during the iterations when it finds the header, the trailer and the last digit of the destination address. It must not try to match the header with the terminal address or place the trailer or the final digit of the destination address in the message. **You might try adding the rest of the logic from the flowchart (Figure 13-13) to the structured program. Note that the order of operations is often critical. You must be sure that the program does not complete one phase and start the next one during the same iteration.**

REVIEW OF STRUCTURED PROGRAMMING

Structured programming brings discipline to program design. It forces you to limit the types of structures you use and the sequence of operations. It provides single-entry, single-exit structures, which you can check for logical accuracy. Structured programming often makes the designer aware of inconsistencies or possible combinations of inputs. Structured programming is not a cure-all, but it does bring some order into a process that can be chaotic. The structured program should also aid in debugging, testing, and documentation.

Structured programming is not simple. The programmer must not only define the problem adequately, but must also work through the logic carefully. This is tedious and difficult, but it results in a clearly written, working program.

The particular structures we have presented are not ideal and are often awkward. In addition, it can be difficult to distinguish where one structure ends and another begins, particularly if they are nested. Theorists may provide better structures in the future, or designers may wish to add some of their own. Some kind of terminator for each structure seems necessary, since indenting does not always clarify the situation. "End" is a logical terminator for the "do-while" loop. There is no obvious terminator, however, for the "if-then-else" statement; some theorists have suggested "endif" or "fi" ("if" backwards), but these are both awkward and detract from the readability of the program.

`TERMINATORS FOR STRUCTURES`

We suggest the following rules for applying structured programming:

`RULES FOR STRUCTURED PROGRAMMING`

1) **Begin by writing a basic flowchart** to help define the logic of the program.

2) **Start with the "sequential," "if-then-else," and "do-while" constructs.** They are known to be a complete set, i.e., any program can be written in terms of these structures.

3) **Indent each level** a few spaces from the previous level, so that you will know which statements belong where.

4) **Use terminators for each structure;** e.g., "end" for the "do-while" and "endif" or "fi" for the "if-then-else". The terminators plus the indentation should make the program reasonably clear.

5) **Emphasize simplicity and readability.** Leave lots of spaces, use meaningful names, and make expressions as clear as possible. Do not try to minimize the logic at the cost of clarity.

6) **Comment the program** in an organized manner.

7) **Check the logic.** Try all the extreme cases or special conditions and a few sample cases. Any logical errors you find at this level will not plague you later.

TOP-DOWN DESIGN

The remaining problem is how to check and integrate modules or structures. Certainly we want to divide a large task into sub-tasks. But how do we check the sub-tasks in isolation and put them together? The standard procedure, called "bottom-up design," requires extra work in testing and debugging and leaves the entire integration task to the end. What we need is a method that allows testing and debugging in the actual program environment and modularizes system integration.

> BOTTOM-UP DESIGN

This method is "top-down design." Here we start by writing the overall supervisor program. We replace the undefined sub-programs by program "stubs," temporary programs that may either record the entry, provide the answer to a selected test problem, or do nothing. We then test the supervisor program to see that its logic is correct.

> TOP-DOWN DESIGN METHODS
>
> STUBS

We proceed by expanding the stubs. Each stub will often contain sub-tasks, which we will temporarily represent as stubs. This process of expansion, debugging, and testing continues until all the stubs are replaced by working programs. Note that testing and integration occur at each level, rather than all at the end. No special driver or data generation programs are necessary. We get a clear idea of exactly where we are in the design. **Top-down design assumes modular programming, and is compatible with structured programming as well.**

> EXPANDING STUBS
>
> ADVANTAGES OF TOP-DOWN DESIGN

The disadvantages of top-down design are:

> DISADVANTAGES OF TOP-DOWN DESIGN

1) The overall design may not mesh well with system hardware.

2) It may not take good advantage of existing software.

3) Stubs may be difficult to write, particularly if they must work correctly in several different places.

4) Top-down design may not result in generally useful modules.

5) Errors at the top level can have catastrophic effects, whereas errors in bottom-up design are usually limited to a particular module.

In large programming projects, top-down design has been shown to greatly improve programmer productivity. However, almost all of these projects have used some bottom-up design in cases where the top-down method would have resulted in a large amount of extra work.

Top-down design is a useful tool that should not be followed to extremes. It provides the same discipline for system testing and integration that structured programming provides for module design. The method, however, has more general applicability, since it does not assume the use of programmed logic. However, top-down design may not result in the most efficient implementation.

EXAMPLES

Response to a Switch

The first structured programming example actually demonstrates top-down design as well. The program was:

```
SWITCH = OFF
do while SWITCH = OFF
   READ SWITCH
end
LIGHT = ON
DELAY 1
LIGHT = OFF
```

These statements are really stubs, since none of them is fully defined. For example, what does READ SWITCH mean? If the switch were one bit of input port SPORT, it really means:

```
SWITCH = SPORT AND SMASK
```

where SMASK has a '1' bit in the appropriate position. The masking may, of course, be implemented with a Bit Test instruction.

Similarly, DELAY 1 actually means (if the processor itself provides the delay):

```
REG = COUNT
do while REG ≠ 0
   REG = REG - 1
end
```

COUNT is the appropriate number to provide a one-second delay. **The expanded version of the program is:**

```
SWITCH = 0
do while SWITCH = 0
   SWITCH = SPORT AND MASK
end
LIGHT = ON
REG = COUNT
do while REG ≠ 0
   REG = REG - 1
end
LIGHT = NOT (LIGHT)
```

Certainly this program is more explicit, and could more easily be translated into actual instructions or statements.

The Switch-Based Memory Loader

This example is more complex than the first example, so we must proceed systematically. Here again, **the structured program contains stubs.**

For example, if the HIGH ADDRESS button is one bit of input port CPORT, "if HIADDRBUTTON = 1" really means:

1) Input from CPORT
2) Complement
3) Logical AND with HAMASK

where HAMASK has a '1' in the appropriate bit position and '0s' elsewhere. Similarly the condition "if DATABUTTON = 1" really means:

1) Input from CPORT
2) Complement
3) Logical AND with DAMASK

So, the initial stubs could just assign values to the buttons, e.g.,

    HIADDRBUTTON = 0
    LOADDRBUTTON = 0
    DATABUTTON = 0

A run of the supervisor program should show that it takes the implied "else" path through the "if-then-else" structures, and never reads the switches. Similarly, if the stub were:

    HIADDRBUTTON = 1

the supervisor program should stay in the "do while HIADDRBUTTON = 1" loop waiting for the button to be released. These simple runs check the overall logic.

Now **we can expand each stub and see if the expansion produces a reasonable overall result. Note how debugging and testing proceed in a straightforward and modular manner.** We expand the HIADDRBUTTON = 1 stub to:

    READ CPORT
    HIADDRBUTTON = NOT (CPORT) AND HAMASK

The program should wait for the HIGH ADDRESS button to be closed. The program should then display the values of the switches on the lights. This run checks for the proper response to the HIGH ADDRESS button.

We then expand the LOW ADDRESS button module to:

    READ CPORT
    LOADDRBUTTON = NOT (CPORT) AND LAMASK

With the LOW ADDRESS button in the closed position, the program should display the values of the switches on the lights. This run checks for the proper response to the LOW ADDRESS button.

Similarly, we can expand the DATA button module and check for the proper response to that button. The entire program will then have been tested.

When all the stubs have been expanded, the coding, debugging, and testing stages will all be complete. Of course, we must know exactly what results each stub should produce. However, many logical errors will become obvious at each level without any further expansion.
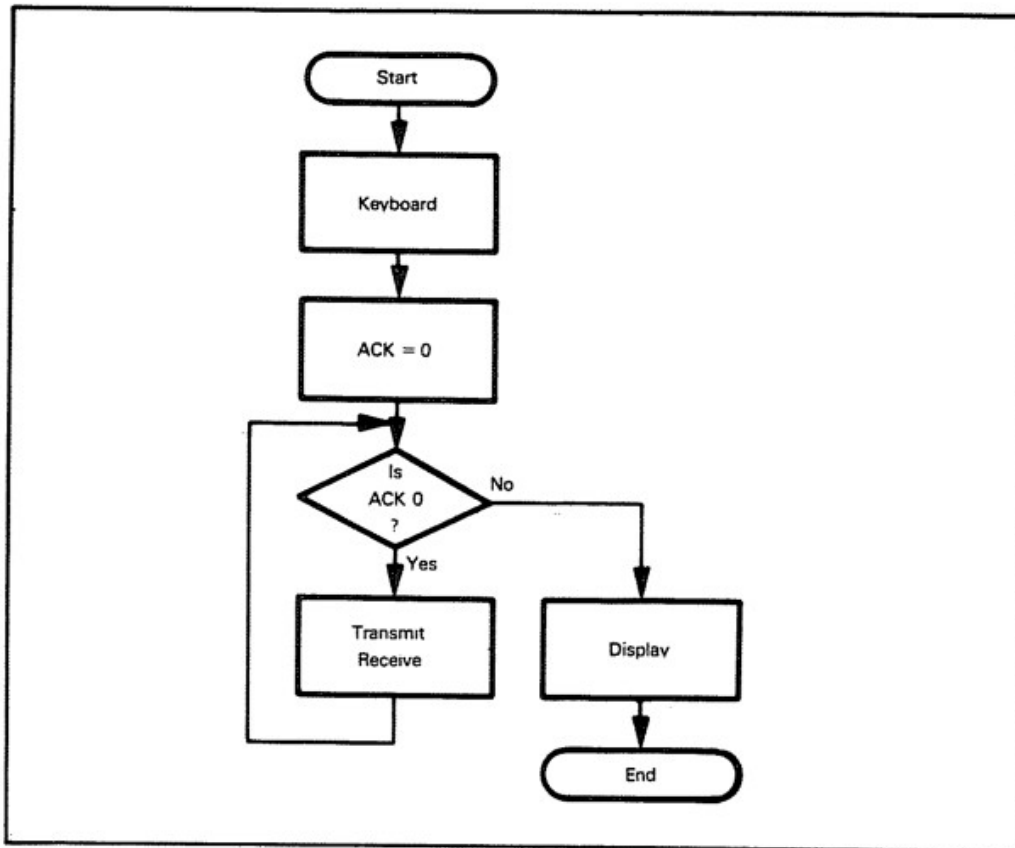
Figure 13-19 Initial Flowchart for Transaction Terminal

The Transaction Terminal

This example, of course, will have more levels of detail. **We could start with the following program** (see Figure 13-19 for a flowchart):

```
KEYBOARD
ACK = 0
do while ACK = 0
  TRANSMIT
  RECEIVE
end
DISPLAY
```

Here KEYBOARD, TRANSMIT, RECEIVE, and DISPLAY are program stubs that will be expanded later. KEYBOARD, for example, could simply place a ten-digit verified number into the appropriate buffer.

13-47
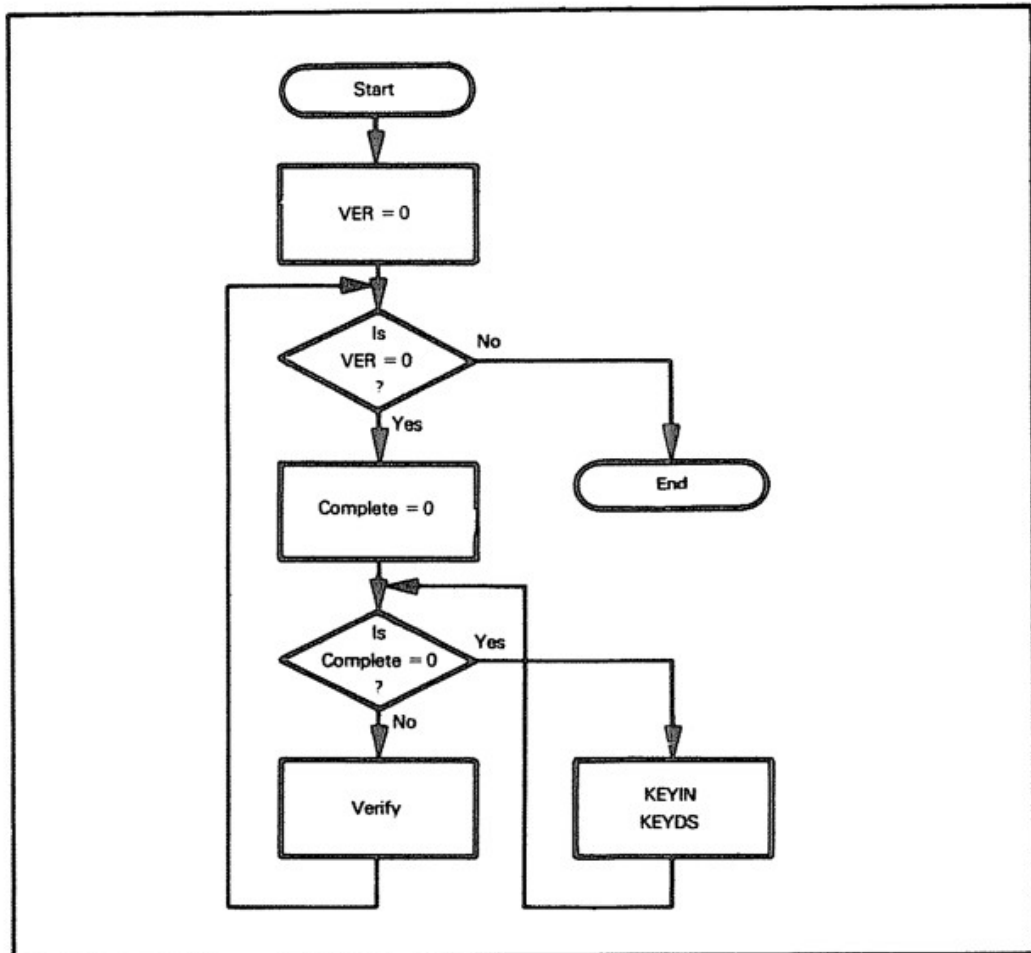
Figure 13-20. Flowchart for Expanded KEYBOARD Routine

The next stage of expansion could produce the following program for **KEYBOARD** (see Figure 13-20):

```
VER = 0
do while VER = 0
  COMPLETE = 0
  do while COMPLETE = 0
    KEYIN
    KEYDS
  end
  VERIFY
end
```

Here VER = 0 means that an entry has not been verified; COMPLETE = 0 means that the entry is incomplete. KEYIN and KEYDS are the keyboard input and display routines respectively. VERIFY checks the entry. A stub for KEYIN would simply place a random entry (from a random number table or generator) into the buffer and set COMPLETE to 1.

We would continue by similarly expanding, debugging, and testing **TRANSMIT, RECEIVE, and DISPLAY**. Note that you should expand each program by one level so that you do not perform the integration of an entire program at any one time. You must use your judgment in defining levels. Too small a step wastes time, while too large a step gets you back to the problems of system integration that top-down design is supposed to solve.

13-48

REVIEW OF TOP-DOWN DESIGN

Top-down design brings discipline to the testing and integration stages of program design. It provides a systematic method for expanding a flowchart or problem definition to the level required to actually write a program. Together with structured programming, it forms a complete set of design techniques.

Like structured programming, top-down design is not simple. The designer must have defined the problem carefully and must work systematically through each level. Here again the methodology may seem tedious, but the payoff can be substantial if you follow the rules.

We recommend the following approach to top-down design:

| FORMAT |
| FOR |
| TOP-DOWN |
| DESIGN |

1) **Start with a basic flowchart.**
2) **Make the stubs as complete and as separate as possible.**
3) **Define precisely all the possible outcomes from each stub** and select a test set.
4) **Check each level carefully** and systematically.
5) **Use** the structures from **structured programming.**
6) **Expand each stub by one level.** Do not try to do too much in one step.
7) **Watch carefully for common tasks and data structures.**
8) **Test and debug after each stub expansion.** Do not try to do an entire level at a time.
9) **Be aware of what the hardware can do. Do not hesitate to stop and do a little bottom-up design where that seems necessary.**

REVIEW OF PROBLEM DEFINITION AND PROGRAM DESIGN

You should note that we have spent an entire chapter without mentioning any specific microprocessor or assembly language, and without writing a single line of actual code. Hopefully, though, you now know a lot more about the examples than you would have if we had just asked you to write the programs at the start. Although we often think of the writing of computer instructions as a key part of software development, it is actually one of the easiest stages.

Once you have written a few programs, coding will become simple. You will soon learn the instruction set, recognize which instructions are really useful, and remember the common sequences that make up the largest part of most programs. You will then find that many of the other stages of software development remain difficult and have few clear rules.

We have suggested here some ways to systematize the important early stages. In the problem definition stage, you must define all the characteristics of the system — its inputs, outputs, processing, time and memory constraints, and error handling. You must particularly consider how the system will interact with the larger system of which it is a part, and whether that larger system includes electrical equipment, mechanical equipment, or a human operator. You must start at this stage to make the system easy to use and maintain.

In the program design stage, several techniques can help you to systematically specify and document the logic of your program. Modular programming forces you to divide the total program into small, distinct modules. Structured programming provides a systematic way of defining the logic of those modules, while top-down design is a systematic method for integrating and testing them. Of course, no one can compel you to follow all of these techniques; they are, in fact, guidelines more than anything else. But they do provide a unified approach to design, and you should consider them a basis on which to develop your own approach.

REFERENCES

1. See, for example, V. P. Srini, "Fault Diagnosis of Microprocessor Systems," _Computer_, January 1977, pp. 60-65. For a description of signature analysis, see G. Gordon and H. Nadig, "Hexadecimal Signatures Identify Trouble-spots in Microprocessor Systems," _Electronics_, March 3, 1977, pp. 89-96. There is also an Application Note (#222) entitled "A Designer's Guide to Signature Analysis" available from Hewlett-Packard.

2. For a brief discussion of human factors considerations, see G. Morris, "Make Your Next Instrument Design Emphasize User Needs and Wants," _EDN_, October 20, 1978, pp. 100-105.

3. D. L. Parnas (see the references below) has been a leader in the area of modular programming.

4. Collected by B. W. Unger (see reference below).

5. Formulated by D. L. Parnas.

The following references provide additional information on problem definition and program design:

Chaplin, N., _Flowcharts_, Auerbach, Princeton, N. J., 1971.

Dahl, O. J., C. A. R. Hoave, and E. W. Dijkstra, _Structured Programming_, Academic Press, New York, N. Y., 1972.

Dalton, W. F., "Design Microcomputer Software like Other Systems — Systematically," _Electronics_, January 19, 1978, pp. 97-101.

Dijkstra, E. W., _A Discipline of Programming_, Prentice-Hall, Englewood Cliffs, N. J., 1976.

Halstead, M. H., _Elements of Software Science_, American Elsevier, New York, 1977.

Hughes, J. K. and J. I. Michtom, _A Structured Approach to Programming_, Prentice-Hall, Englewood Cliffs, N. J., 1977.

Morgan, D. E. and D. J. Taylor, "A Survey of Methods for Achieving Reliable Software," _Computer_, February 1977, pp. 44-52.

Myers, W., "The Need for Software Engineering," _Computer_, February 1978, pp. 12-25.

Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," _Communications of the ACM_, December 1972, pp. 1053-1058.

Parnas, D. L., "A Technique for the Specification of Software Modules with Examples," _Communications of the ACM_, May 1973, pp. 330-336.

Schneider, V., "Prediction of Software Effort and Project Duration — Four New Formulas," _SIGPLAN Notices_, June 1978, pp. 49-59.

Shneiderman, B. et al., "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," _Communications of the ACM_, June 1977, pp. 373-381.

Ulrickson, R. W., "Software Modules Are the Building Blocks," _Electronic Design_, February 1, 1977, pp. 62-66.

Ulrickson, R. W., "Solve Software Problems Step-by-Step," _Electronic Design_, January 18, 1977, pp. 54-58.

Unger, B. W., "Programming Languages for Computer System Simulation," _Simulation_, April 1978, pp. 101-110.

Wirth, N., Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, N. J., 1976.

Wirth, N., Systematic Programming; an Introduction, Prentice-Hall, Englewood Cliffs, N. J., 1973.

Yourdon, E. U., Techniques of Program Structure and Design, Prentice-Hall, Englewood Cliffs, N. J., 1975.

Chapter 14
DEBUGGING AND TESTING

As we noted at the beginning of the previous chapter, debugging and testing are among the most time-consuming stages of software development. **Even though such methods as modular programming, structured programming, and top-down design can simplify programs and reduce the frequency of errors, debugging and testing still are difficult** because they are so poorly defined. The selection of an adequate set of test data is seldom a clear or scientific process. Finding errors sometimes seems like a game of "pin the tail on the donkey," except that the donkey is moving and the programmer must position the tail by remote control. Surely, few tasks are as frustrating as debugging programs.

This chapter will first describe the tools available to aid in debugging. It will then discuss basic debugging procedures, describe the common types of errors, and present some examples of program debugging. The last sections will describe how to select test data and test programs.

We will not do much more than describe the purposes of most of the debugging tools. There is very little standardization in this area, and not enough space to discuss all the devices and programs that are currently available. The examples should give you some idea of the uses, advantages, and limitations of particular hardware or software aids.

SIMPLE DEBUGGING TOOLS

The simplest debugging tools available are:

- **A single-step facility**
- **A breakpoint facility**
- **A Register Dump program (or utility)**
- **A Memory Dump program**

The single-step facility allows you to execute the program one step at a time. Most Z80-based microcomputers have this facility, since the circuitry is fairly simple. Of course, **the only things that you will be able to see when the computer executes a single-step are the states of the output lines that you are monitoring.** The most important lines are:

```
SINGLE-
STEP
```

- Data Bus
- Address Bus
- Control lines $\overline{MREQ}$ (Memory Request), $\overline{IORQ}$ (Input/Output Request), $\overline{RD}$ (Memory Read), and $\overline{WR}$ (Memory Write).

If you monitor these lines (either in hardware or in software), you will be able to see the progression of addresses, instructions, and data as the program executes. You will be able to tell what kind of operations the CPU is performing. This information will inform you of such errors as incorrect Jump instructions, omitted or incorrect addresses, erroneous operation codes, or incorrect data values. However, you cannot see the contents of registers and flags without some additional debugging facility or a special sequence of instructions. Many of the operations of the program cannot be checked in real time.

Table 14-1. Z80 Restart and Interrupt Addresses

Instruction or External Input (Mnemonic) (Pin)	Instruction Object Code (Hex)	Destination Address (Hex)
RST 00H	C7	0000
RST 08H	CF	0008
RST 10H	D7	0010
RST 18H	DF	0018
RST 20H	E7	0020
RST 28H	EF	0028
RST 30H	F7	0030
RST 38H or $\overline{INT}$ in Mode 1	FF	0038
$\overline{NMI}$		0066

There are many errors that a single-step mode cannot help you to find. These include timing errors and errors in the interrupt or DMA systems. Furthermore, the single-step mode is very slow, typically executing a program at less than one millionth

LIMITATIONS OF SINGLE-STEP MODE

of the speed of the processor itself. To single-step through one second of real processor time would take more than ten days. The single-step mode is useful only to check the logic of short instruction sequences.

A breakpoint is a place at which the program will automatically halt or wait so that the user can examine the current status of the system. The program will usually not start again until the operator requests a resumption of execution. Breakpoints allow you to check or pass through an

BREAKPOINT

entire section of a program. Thus, to see if an initialization routine is correct, you can place a breakpoint at the end of it and run the program. You can then check memory locations and registers to see if the entire section is correct. However, note that if the section is not correct, you'll still have to pin down the error, either with earlier breakpoints or with a single-step mode.

Breakpoints complement the single-step mode. You can use breakpoints either to localize the error or to pass through sections that you know are correct. You can then do the detailed debugging in the single-step mode. In some cases, breakpoints do not affect program timing; they can then be used to check input/output interrupts.

Breakpoints often use part or all of the microprocessor interrupt system. Some microprocessors have a special SOFTWARE INTERRUPT or TRAP facility that can act as a breakpoint. **On the Z80, if**

RST AS A BREAKPOINT

you are not already using all the RST vectors in your program, you can use the RST (Restart) instruction as a breakpoint. Table 14-1 gives the destination addresses for the various RST instructions. Chapter 12 describes the RST instruction in more detail. The breakpoint routine can print register and memory contents or just wait (e.g., execute HALT or a conditional jump dependent on a switch input) until you allow the computer to proceed. If you are not using the maskable interrupt ($\overline{INT}$) or the non-maskable interrupt ($\overline{NMI}$) in your system, you can use those vectors as externally controlled breakpoints. But remember that the interrupts (including $\overline{NMI}$) and RST use the Stack and Stack Pointer to store the return address. Figure 14-1 shows a routine where RST results in an endless loop. You would have to clear this breakpoint with a RESET or interrupt signal.

Figure 14-1. A Simple Breakpoint Routine

```
              ORG      18H
      RST18   EQU      18H
              JR       RST18     ;WAIT IN PLACE
```

The simplest method for inserting breakpoints is to replace the first byte of the in-struction with a RST instruction or to replace the instruction with a Jump or CALL instruction. Use of a RST instruction is preferred on the Z80. since it involves the replacement of only a single byte. whereas a JP or CALL involves three bytes. The JR instruction is not suitable for breakpointing because you cannot guarantee that the debug software is within -126 to +129 bytes of the instruction being breakpointed. Multiple-byte instructions used to implement breakpoints can cause problems on the Z80 due to the presence of single-byte instructions. To illustrate this program, examine the program segment shown below:

Memory Address (Hex)	Memory Contents (Hex)	Instruction (Mnemonic)	
100	7B		LD A.E
101	87	L1:	ADD A.A
102	87	L2:	ADD A.A

If you wish to set a breakpoint at location 100_{16} using a 3-byte CALL or JP. the code at locations 101_{16} and 102_{16} will also get overlaid by the CALL or JP instruction. This means that the debugger has to be aware that these locations have also been modified. Any transfers of control to L1 or L2 while the breakpoint is set will produce unexpected results unless the debugger is designed to catch this case. This added complexity can be avoided by using a RST instruction.

Many monitors have facilities for inserting and removing breakpoints implemented via some type of Jump instruction. Such breakpoints do not affect the timing of the program until

INSERTING BREAKPOINTS

the breakpoint is executed. However. note that this procedure will not work if part or all of the program is in ROM or PROM. Other monitors implement breakpoints by actually checking the address lines or the Program Counter in hardware or in software. This method allows breakpoints on addresses in ROM or PROM, but it may affect the timing if the address must be checked in software. A more powerful facility would allow the user to enter an address to which the processor would transfer control. Another possibility would be a return dependent on a switch:

```
              ORG    18H
      RST18   EQU    18H
              PUSH   AF          ;SAVE ACCUMULATOR. FLAGS
      WAITS:  IN     A.(PIODRA)  ;GET SWITCH DATA
              BIT    SW.A        ;IS SWITCH CLOSED?
              JR     NZ.WAITS    ;NO. WAIT UNTIL IT IS
              POP    AF          ;RESTORE ACCUMULATOR. FLAGS
              RET
```

Remember to re-enable the interrupts if the routine uses an external interrupt input.
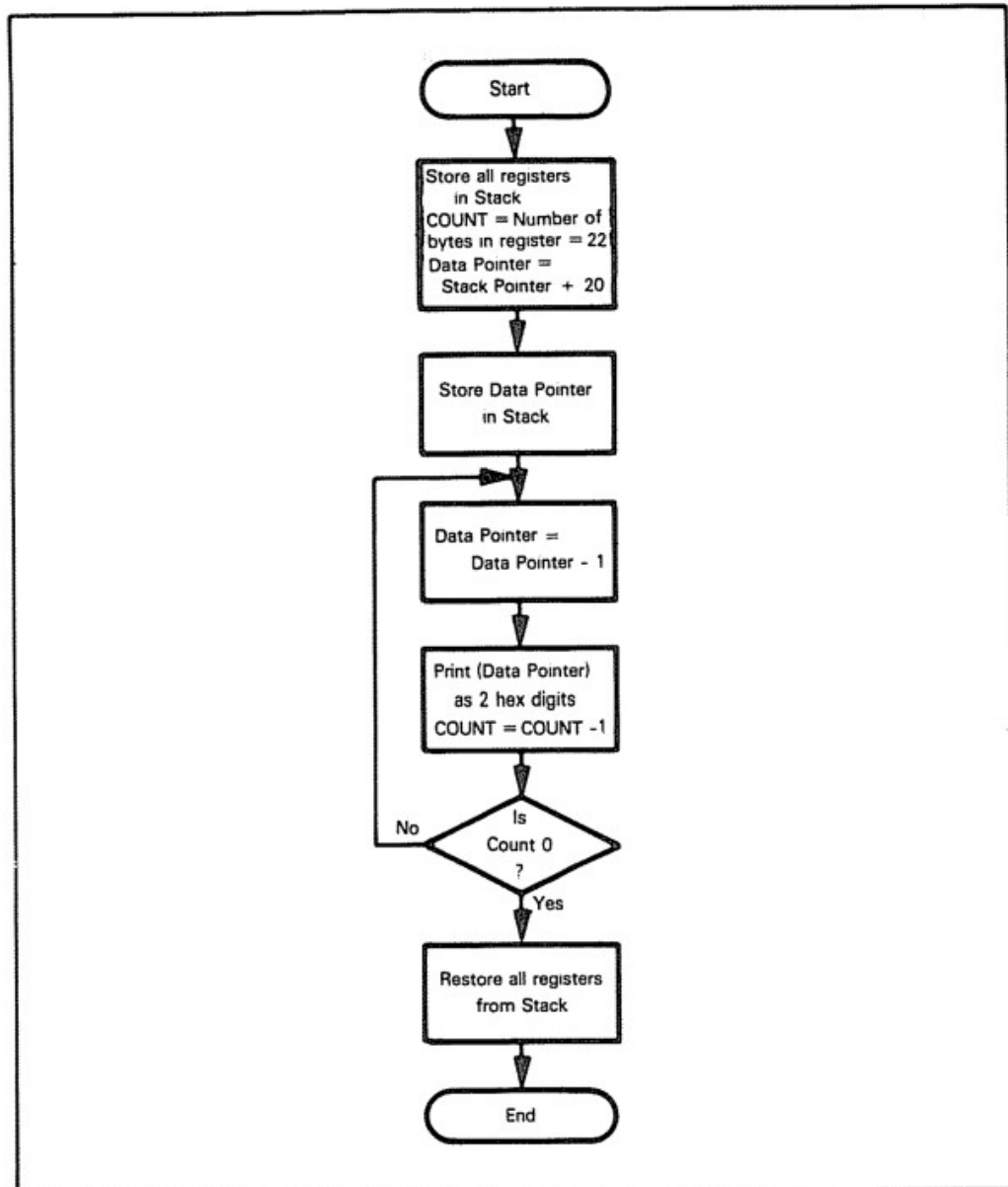
Figure 14-2. Flowchart of Register Dump Program

A Register Dump utility on a microcomputer is a program that lists the contents of all the CPU registers. This information is usually not directly obtainable. **The following routine will print the contents of all the registers on the system printer,** if we assume that PRTHEX prints the contents of the Accumulator as two hexadecimal digits. Figure 14-2 is a flowchart of the program and Figure 14-3 shows a typical result. We assume that the routine is entered with a CALL instruction that stores the old Program Counter at the top of the Stack.

REGISTER DUMPS

```
;
; PLACE ALL CPU REGISTER CONTENTS IN STACK (PC ALREADY ON STACK)
;
        PUSH    AF          ;SAVE REGULAR USER REGISTERS
        PUSH    BC
        PUSH    DE
        PUSH    HL
        PUSH    IX          ;SAVE INDEX REGISTERS
        PUSH    IY
        EX      AF,AF'      ;ACCESS AND SAVE PRIMED CPU REGISTERS
        EXX
        PUSH    AF
        PUSH    BC
        PUSH    DE
        PUSH    HL

;
; USE STACK POINTER AS STARTING ADDRESS
;
        LD      HL,0        ;GET STACK POINTER
        ADD     HL,SP
        LD      DE,20       ;COMPUTE ORIGINAL STACK POINTER
        ADD     HL,DE
        PUSH    HL          ;SAVE ORIGINAL STACK POINTER IN STACK

;
; PRINT CONTENTS OF REGISTERS
; ORDER IS PC(HIGH),PC(LOW),A,F,B,C,D,E,H,L,IX(HIGH),IX(LOW),IY(HIGH),
;     IY(LOW),A',F',B',C',D',E',H',L',SP(HIGH),SP(LOW)
;
        LD      B,22        ;NUMBER OF BYTES = 22
PRNT1:  DEC     HL
        LD      A,(HL)      ;GET A BYTE FROM STACK
        CALL    PRTHEX      ;AND PRINT IT
        DJNZ    PRNT1

;
; RESTORE REGISTERS FROM STACK
;
        POP     HL          ;POP AND DISCARD ORIGINAL STACK POINTER
        POP     HL          ;RESTORE PRIMED CPU REGISTERS
        POP     DE
        POP     BC
        POP     AF
        EX      AF,AF'
        EXX
        POP     IY          ;RESTORE INDEX REGISTERS
        POP     IX
        POP     HL          ;RESTORE REGULAR CPU REGISTERS
        POP     DE
        POP     BC
        POP     AF
        RET
```

```
1D      (A)
42      (F)
07      (B)
3E      (C)
23      (D)
01      (E)
17      (H)
01      (L)
D3      (IX)
58
E2      (IY)
A2
36      (A')
67      (F')
E8      (B')
11      (C')
EB      (D')
09      (E')
D7      (H')
66      (L')
68      (STACK POINTER)
E2
```

Figure 14-3 Results of a Typical Z80 Register Dump

**A Memory Dump is a program that lists the contents of memo-
ry on an output device (such as a printer).** This is a much more
efficient way to examine data arrays or entire programs than just

MEMORY
DUMP

looking at single locations. However, very large memory dumps are not useful (except
to supply scrap paper) because of the sheer mass of information that they produce.
They may also take a long time to execute on a slow printer. **Small dumps may,**
however, **provide the programmer with a reasonable amount of information that
can be examined as a unit. Relationships such as regular repetitions of data pat-
terns or offsets of entire arrays may become obvious.**

A general dump is often rather difficult to write. The programmer should be careful of
the following situations:

1) The size of the memory area exceeds 256 bytes, so that an 8-bit counter will not
   suffice.

2) The ending location is an address smaller than the starting location. This can be
   treated as an error, or simply cause no output, since the user would seldom want to
   print the entire memory contents in an unusual order.

Since the speed of the Memory Dump depends on the speed of the output device, the
efficiency of the routine seldom matters. **The following program will ignore cases
where the starting address is larger than the ending address, and will handle
blocks of any length.** We assume that the starting address is in Register Pair DE and
the ending address is in Register Pair HL.

```
; STOP IF ENDING ADDRESS BEFORE STARTING ADDRESS
;
        AND     A           ;CLEAR CARRY
        SBC     HL.DE       ;IS ENDING ADDRESS BEFORE STARTING?
        JR      C.DONE      ;YES, DO NOT DUMP ANYTHING
        XCHG                ;GET STARTING ADDRESS INTO HL
        INC     DE          ;COUNT = NUMBER OF LOCATIONS TO BE
                        ;           DUMPED
;
; PRINT CONTENTS OF LOCATIONS
;
DUMP:   LD      A.(HL)      ;GET CONTENTS OF A LOCATION
        CALL    PRTHEX      ;AND PRINT IT
        INC     HL
        DEC     DE
        LD      A.E         ;ALL LOCATIONS DUMPED?
        OR      D
        JR      NZ.DUMP     ;NO. CONTINUE DUMPING
DONE:   HALT
```

Note that the only 16-bit Subtract instruction is SBC, which subtracts the contents of a
register pair and the Carry from Register Pair HL. SBC. like other Subtract instructions.
sets the Carry if a borrow is required (contrary to what some Z80 manuals say).

Figure 14-4 shows the output from a dump of memory locations 1000 to 101F.

14-7

```
23    1F    60    54    37    28    3E    00
6E    42    38    17    59    44    98    37
47    36    23    81    E1    FF    FF    5A
34    ED    BC    AF    FE    FF    27    02
```

Figure 14-4. Results of a Typical Memory Dump

This routine correctly handles the case in which the starting and ending locations are the same (try it!). You will have to interpret the results carefully if the dump area includes the Stack, since the dump subroutine itself uses the Stack. PRTHEX may also change memory and Stack locations.

In a memory dump, the data can be displayed in a number of different ways. Common forms are ASCII characters or pairs of hexadecimal digits for 8-bit values and four hexadecimal digits for 16-bit values. The format should be chosen based on the intended use of the dump. It is almost always easier to interpret an object code dump if it is displayed in hexadecimal form rather than ASCII form.

A common and useful dump format is illustrated here:

1000 54 68 65 20 64 75 6D 70   The dump

Each line consists of three parts. The line starts with the hexadecimal address of the first byte displayed on the line. Following the address are eight or sixteen bytes displayed in hexadecimal form. Last is the ASCII representation of the same eight or sixteen bytes. Try rewriting the memory dump program so that it will print the address and the ASCII characters as well as the hexadecimal form of the memory contents.

MORE ADVANCED DEBUGGING TOOLS

The more advanced debugging tools that are most widely used are:

- **Similar programs to check software**
- **Logic analyzers to check signals and timing**

Many variations of both these tools exist, and we shall discuss only the standard features.

The simulator is the computerized equivalent of the pencil-and-paper computer. It **is a computer program that goes through the operating cycle of another computer, keeping track of the contents of all the registers, flags, and memory locations.** We could, of course, do this by hand, but it would require a large amount of effort and close attention to the exact effects of each instruction. The simulator program never gets tired or confused, forgets an instruction or register, or runs out of paper.

| SOFTWARE |
| SIMULATOR |

Most simulators are large FORTRAN programs. They can be purchased or used on the time-sharing services. The Z80 simulator is available in several versions from different sources.

Typical simulator features are:

1) A breakpoint facility. Usually, breakpoints can be set after a particular number of cycles have been executed, when a memory location or one of a set of memory locations is referenced, when the contents of a location or one of a set of locations are altered, or on other conditions.

2) Register and memory dump facilities that can display the values of memory locations, registers, and I/O ports.

3) A trace facility that will print the contents of particular registers or memory locations whenever the program changes or uses them.

4) A load facility that allows you to set values initially or change them during the simulation.

Some simulators can also simulate input/output, interrupts, and even DMA.

The simulator has many advantages:

1) It can provide a complete description of the status of the computer, since the simulator program is not restricted by pin limitations or other characteristics of the underlying circuitry.

2) It can provide breakpoints, dumps, traces, and other facilities, without using any of the processor's memory space or control system. These facilities will therefore not interfere with the user program.

3) Programs, starting points, and other conditions are easy to change.

4) All the facilities of a large computer, including peripherals and software, are available to the microprocessor designer.

On the other hand, the simulator is limited by its software base and its separation from the real microcomputer. The major limitations are:

1) The simulator cannot help with timing problems, since it operates far more slowly than real time and does not model actual hardware or interfaces.

2) The simulator cannot fully model the input/output section.

3) The simulator is usually quite slow. Reproducing one second of actual processor time may require hours of computer time. Using the simulator can be quite expensive.

The simulator represents the software side of debugging; it has the typical advantages and limitations of a wholly software-based approach. The simulator can provide insight into program logic and other software problems, but cannot help with timing, I/O, and other hardware problems.

The logic or microprocessor analyzer is the hardware solution to debugging. Basically, the analyzer is the parallel digital version of the standard oscilloscope. The analyzer displays information in binary, hexadecimal or mnemonic form on a CRT, and has a variety of triggering events, thresholds, and inputs. Most analyzers also have a memory so that they can display the past contents of the busses.

```
LOGIC
ANALYZER
```

The standard procedure is to set a triggering event, such as the occurrence of a particular address on the Address Bus or instruction on the Data Bus. For example, one might trigger the analyzer if the microcomputer tries to store data in a particular address or execute an input or output instruction. One may then look at the sequence of events that preceded the breakpoint. **Common problems you can find in this way include short noise spikes (or glitches), incorrect signal sequences, overlapping wave-forms, and other timing or signaling errors. Of course, a software simulator could not be used to diagnose those errors any more than a logic analyzer could conveniently be used to find errors in program logic.**

Logic analyzers vary in many respects. Some of these are:

1) Number of input lines. At least 24 are necessary to monitor an 8-bit Data Bus and a 16-bit Address Bus. Still more are necessary for control signals, clocks, and other important inputs.

2) Amount of memory. Each previous state that is saved will occupy several bytes.

3) Maximum frequency. It must be several MHz to handle the fastest processors.

4) Minimum signal width (important for catching glitches).

5) Type and number of triggering events allowed. Important features are pre- and post-trigger delays; these allow the user to display events occurring before or after the trigger event.

6) Methods of connecting to the microcomputer. This may require a rather complex interface.

7) Number of display channels.

8) Binary, hexadecimal or mnemonic displays.

9) Display formats.

10) Signal hold time requirements.

11) Probe capacitance.

12) Single or dual thresholds.

All of these factors are important in comparing different logic and microprocessor analyzers, since these instruments are new and unstandardized. A tremendous variety of products is already available and this variety will become even greater in the future.

Logic analyzers, of course, **are necessary only for systems with complex timing. Simple applications with low-speed peripherals have few hardware problems that a designer cannot handle with a standard oscilloscope.**

DEBUGGING WITH CHECKLISTS

The designer cannot possibly check an entire program by hand; however, there are certain trouble spots that the designer can easily check. **You can use systematic hand checking to find a large number of errors without resorting to any debugging tools.**

The question is where to place the effort. The answer is on points that can be handled with either a yes-no answer or with a simple arithmetic calculation. Do not try to do complex

arithmetic, follow all the flags, or try every conceivable case. Limit your hand checking to matters that can be settled easily. Leave the complex problems to be solved with the aid of debugging tools. But proceed systematically; build your checklist, and make sure that the program performs the basic operations correctly.

The first step is to compare the flowchart or other program documentation with the actual code. Make sure that everything that appears in one also appears in the other. A simple checklist will do the job. It is easy to completely omit a branch or a processing section.

Next concentrate on the program loops. Make sure that all registers and memory locations used inside the loops are initialized correctly. This is a common source of errors; once again, a simple checklist will suffice.

Now look at each conditional branch. Select a sample case that should produce a branch and one that should not; try both of them. Is the branch correct or reversed? If the branch involves checking whether a number is above or below a threshold, try the equality case. Does the correct branch occur? Make sure that your choice is consistent with the problem definition.

Look at the loops as a whole. Try the first and last iterations by hand: these are often troublesome special cases. What happens if the number of iterations is zero; i.e., there is no data or the table has no elements? Does the program fall through correctly? Programs often will perform one iteration unnecessarily, or, even worse, decrement counters past zero before checking them.

Check off everything down to the last statement. Don't assume (hopefully) that the first error is the only one in the program. Hand checking will allow you to get the maximum benefit from debugging runs, since you will get rid of many simple errors ahead of time.

A quick review of the hand checking questions:

```
HAND
CHECKING
QUESTIONS
```

1) Is every element of the program design in the program (and vice versa for documentation purposes)?

2) Are all registers and memory locations used inside loops initialized before they are used?

3) Are all conditional branches correct?

4) Do all loops start and end properly?

5) Are equality cases handled correctly?

6) Are trivial cases handled correctly?

LOOKING FOR ERRORS

Of course, despite all these precautions (or if you skip over some of them), programs often still don't work. The designer is left with the problem of how to find the mistakes. The hand

```
COMMON
ERRORS
```

checklist provides a starting place if you didn't use it earlier; some of the errors that you may not have eliminated are:

1) **Failure to initialize variables such as counters, pointers, sums, etc.** Do not assume that registers, memory locations, or flags necessarily contain zero before they are used.

2) **Inverting the logic of a conditional jump,** such as using Jump on Carry when you mean Jump on Not Carry. Remember the effects of a comparison or subtraction (A is the contents of the Accumulator, M the contents of the register or memory location):

$$\text{Zero flag} = 1 \text{ if } A = M$$
$$= 0 \text{ if } A \neq M$$
$$\text{Carry flag} = 1 \text{ if } A < M$$
$$= 0 \text{ if } A \geq M$$

Note particularly that Carry = 0 if A = M, (the equality case). So, Jump on Carry means jump if A < M, and Jump on Not Carry means jump if A ≥ M. If you want the equality case on the other side, try either reversing the roles of A and M or adding 1 to M. For example, if you want a jump if A ≥ 10, use:

```
CP      10
JR      NC,ADDR
```

If, on the other hand, you want a jump if A > 10, use:

```
CP      11
JR      NC,ADDR
```

3) **Updating the counters and pointers in the wrong place or not at all.** Be sure that there are no paths through a loop that either skip or repeat the updating instructions.

14-11

4) **Failure to fall through correctly in trivial cases** such as no data in a buffer, no tests to be run, or no entries in a transaction. Do not assume that such cases will never occur unless the program specifically eliminates them.

Other problems to watch for are:

5) **Reversing the order of operands.** Remember that the LD instruction moves the second operand into the first operand. For example, LD B,A moves the contents of A to B, not the other way around.

6) **Changing condition flags before you use them.**

Remember that INC and DEC, when applied to a single register or memory location, affect all the flags except Carry. Remember also that POP AF and EX AF.AF' affect all the flags, and that Logical instructions clear the Carry.

7) **Failing to change condition flags when you intend to.**

The Zero and Sign flags may not represent the current state of the Accumulator, since many instructions (particularly LD) do not change the flags. Note that incrementing or decrementing register pairs (for example, INC HL or DEC BC) and complementing the Accumulator (CPL) affect no flags at all.

8) **Confusing values and addresses.**

Remember that LD HL,1000H loads HL with the number 1000 (hex) while LD HL,(1000H) loads HL with the contents of memory locations 1000 and 1001. A similar distinction applies to LD A,COUNT and LD A,(COUNT).

9) **Accidentally reinitializing a register or memory location.**

Make sure that no Jump instructions transfer control back to initialization statements.

10) **Confusing numbers and characters.**

Remember that the ASCII and EBCDIC representations of digits differ from the digits themselves. For example, ASCII 7 is hex 37, whereas hex 07 is the ASCII BELL character.

11) **Confusing binary and decimal numbers.**

Remember that the BCD representation of a number differs from its binary representation. For example, BCD 36, when treated as a simple hexadecimal constant, is equivalent to 54 decimal (try it).

12) **Reversing the order in subtraction. Be careful also with other operations (like division) that do not commute.** Remember that SUB and CP produce A-M, not M-A.

13) **Ignoring the effects of subroutines and macros.**

Don't assume that calls to subroutines or invocations of macros will not change flags, registers, or memory locations. Be sure of exactly what effects subroutines or macros have. Note that it is very important to document these effects so that the user can determine them without going through the entire listing.

14) **Using the Shift instructions improperly.**

Remember the precise effects of RLC, RL, RRC, RR, SLA, SRA, and SRL. They are all 1-bit shifts. SLA and SRL both clear the empty bit. SRA preserves the sign (most significant bit) by extending it to the right. RLC and RRC are circular shifts that do not include the Carry in the circular register; RL and RR are circular shifts that include the Carry. Remember that these instructions affect all the flags, even if they are applied to the data in a memory location. Note, however, that the one-word shifts RLCA, RLA, RRCA, and RRA affect only the Carry.

15) **Counting the length of an array incorrectly.**

Remember that there are five (not four) memory locations included in addresses 0100 through 0104, inclusive.

16) **Confusing registers and register pairs.**

Remember that the CPU registers and register pairs are physically the same. You can use them singly for 8-bit data or in pairs for addresses or 16-bit data, but not both at the same time. Note that INC HL actually increments L, affecting H only if L is incremented to zero.

17) **Confusing 8- and 16-bit registers.**

The Accumulator and other CPU registers are eight bits long, while the index registers, Program Counter, Stack Pointer, and register pairs are 16 bits long. You cannot transfer the contents of a 16-bit register to an 8-bit register or vice versa.

18) **Forgetting that 16-bit numbers or addresses occupy two memory locations.**

LD HL,(40H) loads Register Pair HL with the contents of memory locations 0040 and 0041. Similarly, PUSH DE stores Register Pair DE in two Stack locations. Also remember that the Z80 stores all 2-byte quantities in low-order/high-order format. For example, LD (40H),HL will store the contents of Register L in location 0040 and the contents of Register H in location 0041.

19) **Confusing the Stack and the Stack Pointer.**

DEC, INC, and LD affect the Stack Pointer, not the contents of the Stack. PUSH and POP transfer data to or from the Stack. Remember that CALL, RET, RETI, RETN, and RST also use the Stack to save or restore the Program Counter. The response to an interrupt always involves saving the old Program Counter in the Stack even if no explicit instruction is obtained externally (as in responding to $\overline{\text{NMI}}$ or to $\overline{\text{INT}}$ in interrupt modes 1 or 2). Note that such instructions as EX (SP),HL do not affect the Stack Pointer; they exchange the top two memory locations in the Stack with the contents of a register pair or Index register, but leave the Stack length unchanged.

20) **Forgetting to initialize the Stack Pointer.**

Remember that you must place the proper memory address into the Stack Pointer before calling any subroutines or performing any Stack operations.

21) **Changing a register or memory location before using it.**

Remember that LD changes the contents of the destination (but not the source). Be careful of instructions that implicitly use certain registers — for example, DJNZ decrements Register B; LDI, LDIR, LDD, LDDR, CPI, CPIR, CPD, and CPDR all decrement the Byte Counter in Register Pair BC and increment or decrement Register Pair HL. LDI, LDIR, LDD, and LDDR also increment or decrement Register Pair DE. INI, INIR, IND, INDR, OUTI, OUTIR, OUTD, and OTDR all decrement Register B and increment or decrement Register Pair HL.

22) **Forgetting to transfer control past sections of the program that should not be executed in particular situations.**

Remember that the computer will proceed sequentially through the program memory unless specifically ordered to do otherwise.

Interrupt-driven programs are particularly difficult to debug, since errors may occur randomly. If, for example, the program enables the interrupts a few instructions too early, an error will occur only if an interrupt is received while the program is executing those few instructions. **In fact you can usually assume that randomly occurring errors are caused by the interrupt system.**[2] **Typical errors in interrupt-driven programs are:**

1) **Forgetting to re-enable interrupts after accepting one and servicing it.**

   The processor disables the interrupt system automatically on RESET or on accepting an interrupt. Be sure that no possible sequences fail to re-enable the interrupt system. Remember that, in addition to re-enabling interrupts, the program often has to perform some action to cause the interrupting signal to be reset. If this is not done, it will appear as if the interrupting device is constantly requesting service.

2) **Using the Accumulator before saving it;** i.e., PUSH AF must precede any input or output operations that involve the Accumulator.

3) **Forgetting to save and restore the Accumulator and flags** (Register Pair AF).

4) **Restoring registers in the wrong order.**

   If the order in which they were saved was:

   > PUSH AF
   > PUSH BC
   > PUSH DE
   > PUSH HL

   the order of restoration should be:

   > POP HL
   > POP DE
   > POP BC
   > POP AF

5) **Enabling interrupts before establishing all the necessary conditions** such as priority, flags, PIO and SIO configurations, pointers, counters, etc.

   A checklist can aid here.

6) **Leaving results in registers and destroying them in the restoration process.**

   As noted earlier, registers should not be used to pass information between the regular program and the interrupt service routines.

7) **Forgetting that RST (and $\overline{\text{NMI}}$) leaves an address in the Stack whether you use it or not.**

   You may have to re-initialize or update the Stack Pointer.

8) **Not disabling the interrupt during multi-word transfers or instruction sequences.**

   Watch particularly for situations where the interrupt service routine may use the same memory locations that the program is using.

Hopefully, these lists will at least give you some ideas as to where to look for errors. Unfortunately, even the most systematic debugging can still leave some truly puzzling problems, particularly when interrupts are involved.[3]
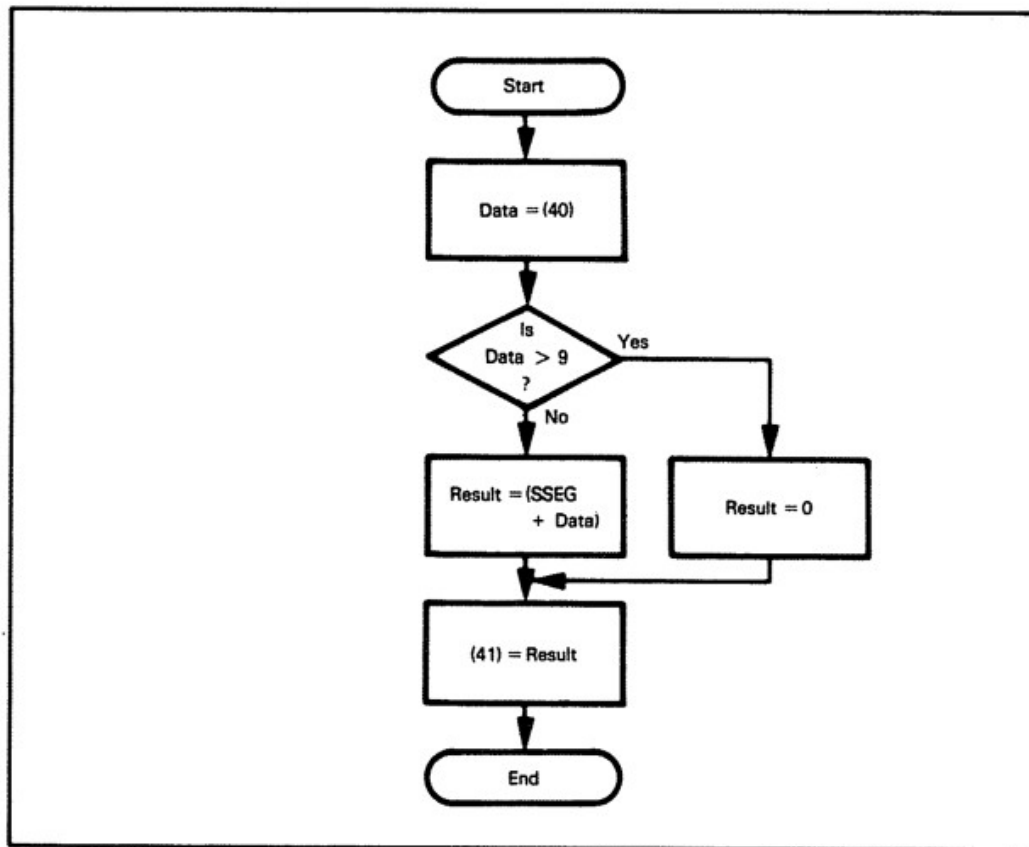
Figure 14-5. Flowchart of Decimal to Seven-Segment Conversion

Debugging Example 1: Decimal to Seven-Segment Conversion

The program converts a decimal number in memory location 0040 to a seven-segment code in memory location 0041. It blanks the display if memory location 0040 does not contain a decimal number.

```
DEBUGGING
A CODE
CONVERSION
PROGRAM
```

Initial Program (from flowchart in Figure 14-5):

```
        LD      A,40H       ;GET DATA
        CP      9           ;IS DATA A DECIMAL DIGIT?
        JR      C,DONE      ;NO, KEEP ERROR CODE
        LD      HL,(SSEG)   ;GET BASE ADDRESS OF 7-SEGMENT TABLE
        LD      D,A
        ADD     HL,DE       ;FIND ELEMENT BY INDEXING
        LD      A,(HL)      ;GET 7-SEGMENT CODE FROM TABLE
DONE.   LD      (41H),A     ;SAVE 7-SEGMENT CODE OR ERROR CODE
        HALT
SSEG.   DEFB    3FH
        DEFB    06H
        DEFB    5BH
        DEFB    4FH
        DEFB    66H
        DEFB    6DH
        DEFB    7DH
        DEFB    07H
        DEFB    7DH
        DEFB    6FH
```

Using the checklist procedure, we were able to find the following errors:

1) The block that cleared Result had been omitted.

2) The conditional branch was incorrect.

For example, if the data is zero, CP 9 sets the Carry, since $0 < 9$. However, the jump on the opposite condition (i.e., JR NC,DONE) still did not produce the correct result. Now the program handles the equality case incorrectly since, if the data is 9, CP 9 clears the Carry and causes a jump. The correct version is:

```
        CP      10          ;IS DATA A DECIMAL DIGIT?
        JR      NC,DONE     ;NO, KEEP ERROR CODE
```

Second Program:

```
        LD      B.0             ;GET ERROR CODE TO BLANK DISPLAY
        LD      A.40H           ;GET DATA
        CP      10              ;IS DATA A DECIMAL DIGIT?
        JR      NC.DONE         ;NO. KEEP ERROR CODE
        LD      HL.(SSEG)       ;GET BASE ADDRESS OF 7-SEGMENT TABLE
        LD      D.A
        ADD     HL.DE           ;FIND ELEMENT BY INDEXING
        LD      A.(HL)          ;GET 7-SEGMENT CODE FROM TABLE
DONE:   LD      (41H).A         ;SAVE 7-SEGMENT CODE OR ERROR CODE
        HALT
SSEG.   DEFB    3FH
        DEFB    06H
        DEFB    5BH
        DEFB    4FH
        DEFB    66H
        DEFB    6DH
        DEFB    7DH
        DEFB    07H
        DEFB    7DH
        DEFB    6FH
```

This version was hand checked successfully.

Since the program was simple. the next stage was to single-step through it with read data. The data selected for the trials was:

> 0   (the smallest number)
> 9   (the largest number)
> 10   (a border case)
> 6B (hex) (random)

The first trial was with zero in location 0040 (hex). The first error was obvious — LD A.40H loaded the number 40 into A. not the contents of memory location 0040. The correct instruction was LD A.(40H). After this correction was made. the program moved along with no apparent errors until it tried to execute the LD A.(HL) instruction.

The contents of the Address Bus during the data fetch was 0647. an address that did not even exist in the microcomputer. Clearly. something had gone wrong.

It was now time for some more hand-checking Since we knew that JR NC.DONE was correct. the error was beyond that instruction but before LD A.(HL). A hand check showed:

1) LD HL.(SSEG) places 3F (hex) into L and 06 (hex) into H.

   This is clearly wrong. We want LD. HL.SSEG. not LD HL.(SSEG). That is. we want the address SSEG. not the contents of that address. to be loaded into Register Pair HL.

2) LD D.A places 0 into Register D.

   This is wrong — the data should be placed into E. since we want to add it to the least significant bits of the table address. In fact. an instruction should clear Register D. since the erroneous program was not initializing or changing the other half of Register Pair DE at all.

Third Program:

```
          LD      B,0          ;GET ERROR CODE TO BLANK DISPLAY
          LD      A,(40H)      ;GET DATA
          CP      10           ;IS DATA A DECIMAL DIGIT?
          JR      NC,DONE      ;NO, KEEP ERROR CODE
          LD      HL,SSEG      ;GET BASE ADDRESS OF 7-SEGMENT TABLE
          LD      E,A
          LD      D,0          ;USE DATA AS 16-BIT INDEX
          ADD     HL,DE        ;FIND ELEMENT BY INDEXING
          LD      A,(HL)       ;GET 7-SEGMENT CODE FROM TABLE
DONE:     LD      (41H),A      ;SAVE 7-SEGMENT CODE OR ERROR CODE
          HALT
SSEG:     DEFB    3FH
          DEFB    06H
          DEFB    5BH
          DEFB    4FH
          DEFB    66H
          DEFB    6DH
          DEFB    7DH
          DEFB    07H
          DEFB    7DH
          DEFB    6FH
```

This program produced the following results:

Data	Result
00	3F
09	6F
0A	0A
6B	6B

The program was not clearing the result if the data was invalid, i.e., greater than 9. The program never used the blank code in Register B. Since the program was simple, it could be tested for all the decimal digits. The results were:

Data	Result
0	3F
1	06
2	5B
3	4F
4	69
5	6D
6	7D
7	07
8	7D
9	6F

Note that the result for number 8 is wrong — it should be 7F Since everything else is correct, the error is almost surely in the table. In fact, entry 8 in the table had been miscopied.

The final program is:

```
; DECIMAL TO 7-SEGMENT CONVERSION
;
          LD      B,0          ;GET ERROR CODE TO BLANK DISPLAY
          LD      A,(40H)      ;GET DATA
          CP      10           ;IS DATA A DECIMAL DIGIT?
          JR      NC,DONE      ;NO. KEEP ERROR CODE
          LD      HL,SSEG      ;GET BASE ADDRESS OF 7-SEGMENT TABLE
          LD      E,A
          LD      D,0          ;USE DATA AS 16-BIT INDEX
          ADD     HL,DE        ;FIND ELEMENT BY INDEXING
          LD      B,(HL)       ;GET 7-SEGMENT CODE FROM TABLE
DONE:     LD      A,B
          LD      (41H),A      ;SAVE 7-SEGMENT CODE OR ERROR CODE
          HALT
SSEG.     DEFB    3FH
          DEFB    06H
          DEFB    5BH
          DEFB    4FH
          DEFB    66H
          DEFB    6DH
          DEFB    7DH
          DEFB    07H
          DEFB    7FH
          DEFB    6FH
```

The errors encountered in this program are typical of the ones that Z80 assembly language programmers should anticipate. They include:

1) Failing to initialize registers or memory locations.

2) Inverting the logic on conditional branches.

3) Branching incorrectly in the case in which the operands are equal.

4) Confusing immediate and direct addressing. i.e., data and addresses.

5) Failing to distinguish between 8-bit data and 16-bit addresses.

6) Branching to the wrong place so that one path through the program is incorrect.

7) Copying lists of numbers (or instructions) incorrectly.

Note that straightforward instructions like ADD. SUB. AND. etc., seldom produce any problems. One particularly annoying error that you should watch for is reversing the operands on LD instructions. Many of these errors can be eliminated through the use of a low-level system programming language like PLZ/ASM.[4]

14-19
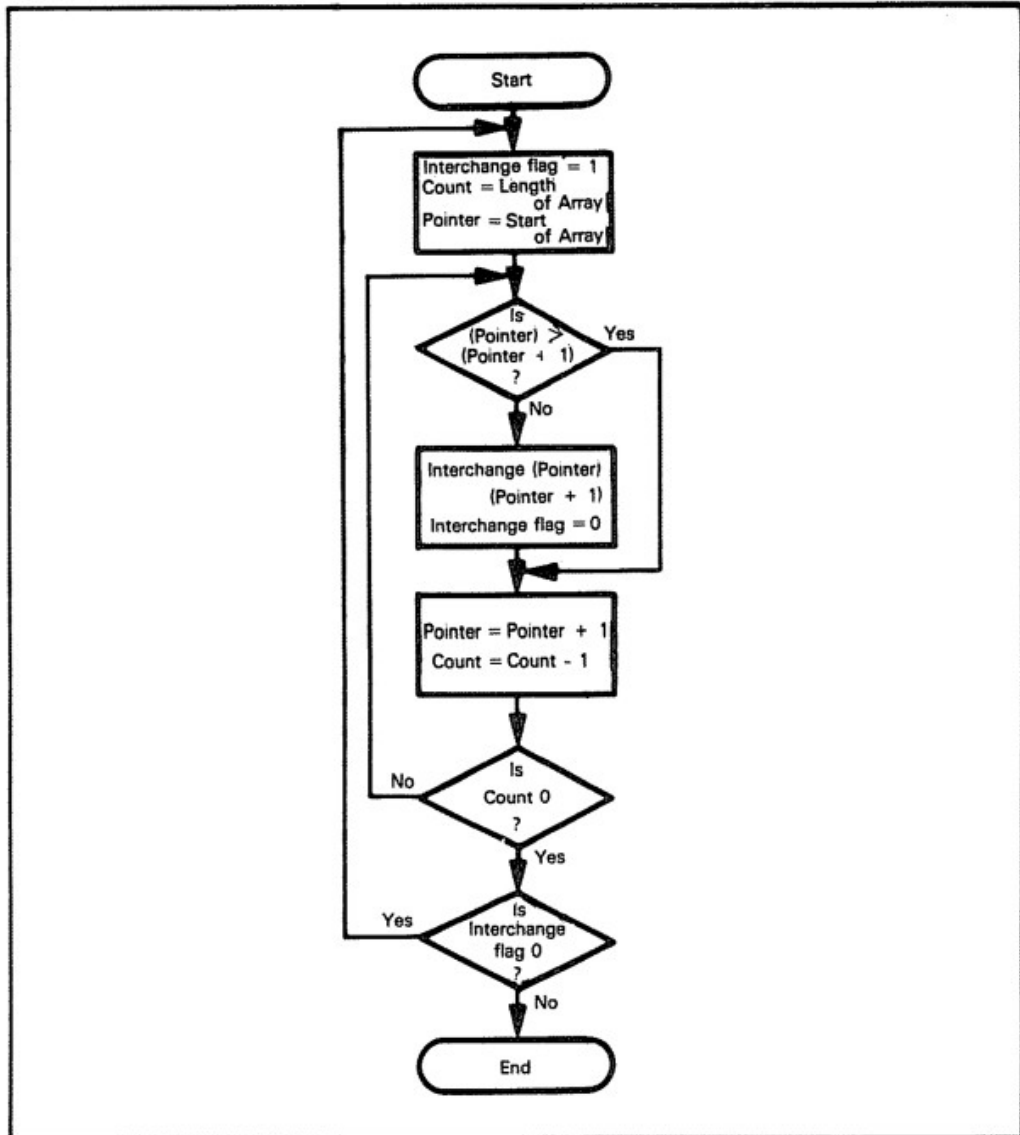
Figure 14-6. Flowchart of Sort Program

14-20

Debugging Example 2: Sort into Decreasing Order

The program sorts an array of unsigned 8-bit binary numbers into decreasing order. The array begins in memory location 0041 and its length is in memory location 0040.

Initial Program (from flowchart in Figure 14-6):

```
        LD      C.0         ;CLEAR INTERCHANGE FLAG
        LD      A,(40H)     ;COUNT = LENGTH OF ARRAY
        LD      C.A
        LD      HL.41H      ;POINT TO START OF ARRAY
PASS1:  LD      A.(HL)      ;GET ELEMENT FROM ARRAY
        INC     HL
        CP      (HL)        ;IS IT LESS THAN NEXT ELEMENT?
        JR      C.CNT       ;NO, NO INTERCHANGE NECESSARY
        LD      (HL).A      ;YES, INTERCHANGE ELEMENTS
        INC     HL
CNT:    DJNZ    PASS1
        DEC     C           ;WAS INTERCHANGE FLAG SET?
        JR      NZ.PASS1    ;YES. DO ANOTHER PASS
        HALT
```

The hand check shows that all the blocks in the flowchart have been implemented in the program and that all the registers have been initialized. The conditional branches must be examined carefully. The instruction JR C.CNT must force a branch if the new value is less than or equal to the old value. Note that the equality case must not result in an interchange. since this will create an endless loop with the two equal elements being switched back and forth.

Try an example:

$$(0040) = 30$$
$$(0041) = 37$$

CP (HL) results in the calculation of 30−37. The Carry is set to one. This example should result in an interchange but does not.

JR NC.CNT will provide the proper branch in this case. If the two numbers are equal, the comparison will clear the Carry and JR NC.CNT is again correct.

How about JR NZ.SORT at the end of the program? If there are any elements out of order. the interchange flag will be one, so the branch is wrong. It should be JR Z.SORT.

Now let's hand check the first iteration of the program. The initialization results in the following values:

$$A = COUNT$$
$$B = COUNT$$
$$C = 0$$
$$HL = 0041$$

The effects of the loop instructions are:

```
        LD      A,(HL)      ;A = (0041)
        INC     HL          ;HL = 0042
        CP      (HL)        ;(0041)-(0042)
        JR      NC.CNT
        LD      (HL).A      ;(0042) = (0041)
        INC     HL          ;HL = 0043
CNT:    DJNZ    PASS1       ;B = COUNT-1
```

14-21

Note that we have already checked the Conditional Jump instructions. Clearly the logic is incorrect. If the first two numbers are out of order, the results after the first iteration should be:

$$(0041) = \text{OLD } (0042)$$
$$(0042) = \text{OLD } (0041)$$
$$\text{HL} = 0042$$
$$\text{B} = \text{COUNT-1}$$

Instead, they are:

$$(0041) = \text{UNCHANGED}$$
$$(0042) = \text{OLD } (0041)$$
$$\text{HL} = 0043$$
$$\text{B} = \text{COUNT-1}$$

The error in HL is easy to correct. The second INC HL is unnecessary and should be omitted. The interchange requires a bit more care and a temporary register, i.e.,

```
LD      D,(HL)
LD      (HL),A
DEC     HL
LD      (HL),D
INC     HL
```

An interchange always requires a temporary storage place in which one number can be saved while the other one is being transferred.

All of these changes require a new copy of the program, i.e.,

```
            LD      C,0         ;CLEAR INTERCHANGE FLAG
            LD      A,(40H)     ;COUNT = LENGTH OF ARRAY
            LD      C,A
            LD      HL,41H      ;POINT TO START OF ARRAY
PASS1:      LD      A,(HL)      ;GET ELEMENT FROM ARRAY
            INC     HL
            CP      (HL)        ;IS IT LESS THAN NEXT ELEMENT?
            JR      NC,CNT      ;NO, NO INTERCHANGE NECESSARY
            LD      D,(HL)      ;YES, INTERCHANGE ELEMENTS
            LD      (HL),A
            DEC     HL              ·
            LD      (HL),D
            INC     HL
CNT:        DJNZ    PASS1
            DEC     C           ;WAS INTERCHANGE FLAG SET?
            JR      NZ,PASS1    ;YES, DO ANOTHER PASS
            HALT
```

How about the last iteration? Let's say that there are three elements:

$$
\begin{aligned}
(0040) &= 03 \\
(0041) &= 02 \\
(0042) &= 04 \\
(0043) &= 06
\end{aligned}
$$

Each time through, the program increments Register Pair HL by one. So, at the start of the third iteration,

$$(HL) = 0041 + 2 = 0043$$

The effects of the loop instructions are:

```
            LD      A,(HL)      ;A = (0043)
            INC     HL          ;HL = 0044
            CP      (HL)        ;(0043)-(0044)
```

This is incorrect; the program has tried to move beyond the end of the data. The previous iteration should, in fact, have been the last one, since the number of pairs is one less than the number of elements. The correction is to reduce the number of iterations by one; this can be accomplished by placing DEC B after LD A,(40H).

How about the trivial cases? What happens if the array contains no elements at all, or only one element? The answer is that the program does not work correctly and may change a whole block of data improperly and without any warning (try it!). The corrections to handle the trivial cases are simple but essential; the cost is only a few bytes of memory to avoid problems that could be very difficult to solve later.

14-23

The new program is:

```
        LD    C,0        ;CLEAR INTERCHANGE FLAG
        LD    A,(40H)    ;COUNT = LENGTH OF ARRAY
        CP    2          ;DOES ARRAY HAVE 2 OR MORE ELEMENTS?
        JR    C,DONE     ;NO, NO ACTION NECESSARY
        LD    B,A
        DEC   B          ;NUMBER OF PAIRS = COUNT-1
        LD    HL,41H     ;POINT TO START OF ARRAY
PASS1:  LD    A,(HL)     ;GET ELEMENT FROM ARRAY
        INC   HL
        CP    (HL)       ;IS IT LESS THAN NEXT ELEMENT?
        JR    NC,CNT     ;NO, NO INTERCHANGE NECESSARY
        LD    D,(HL)     ;YES, INTERCHANGE ELEMENTS
        LD    (HL),A
        DEC   HL
        LD    (HL),D
        INC   HL
CNT:    DJNZ  PASS1
        DEC   C          ;WAS INTERCHANGE FLAG SET?
        JR    NZ,PASS1   ;YES, DO ANOTHER PASS
        HALT
```

Now it's time to check the program on the computer or on the simulator. A simple set of data is:

$$(0040) = 02$$
$$(0041) = 00$$
$$(0042) = 01$$

This set consists of two elements in the wrong order. The program should take two passes. The first pass should rearrange the elements, producing:

$$(0041) = 01$$
$$(0042) = 00$$
$$C = 01$$

The second pass should complete the operation and produce:

$$C = 00$$

This program is rather long for single stepping, so we'll use breakpoints instead. Each breakpoint will halt the computer and print the contents of all the registers. The breakpoints will come:

1) After LD HL,41H to check the initialization.
2) After CP (HL) to check the comparison.
3) After the second INC HL (i.e., just before the label CNT) to check the interchange.
4) After DEC C to check the completion of a pass through the array. The contents of the registers after the first breakpoint were:

Register	Contents
A	02
B	01
C	00
H	00
L	41

These are all correct, so the program is performing the initialization correctly in this case.

The results at the second breakpoint were:

Register	Contents
A	00
B	01
C	00
H	00
L	42
CARRY	1

These results are also correct. The results at the third breakpoint were:

Register	Contents
A	00
B	01
C	00
D	01
H	00
L	42

Checking memory showed:

(0041) = 01
(0042) = 00

The results at the fourth breakpoint were:

Register	Contents
A	00
B	01
C	00
D	01
H	00
L	42

Here, Register C does not contain the correct value — it should have been set to one to indicate that an interchange had occurred. In fact, a look at the program shows that no instruction ever changes C to mark the interchange. The correction is to place the instruction LD C,1 after JR NC,CNT.

Now the procedure is to load Register C with the correct value and continue. The second iteration of the second breakpoint gives:

Register	Contents
A	00
B	00
C	00
H	00
L	43
CARRY	1

Clearly the program has proceeded incorrectly without reinitializing the registers (particularly HL). The conditional jump that depends on the interchange flag should transfer control all the way back to the start of the program, not to the label PASS1.

The final version of the program is:

```
SORT    LD      C.0         :CLEAR INTERCHANGE FLAG
        LD      A.(40H)     :COUNT = LENGTH OF ARRAY
        CP      2           :DOES ARRAY HAVE 2 OR MORE ELEMENTS?
        JR      C.DONE      :NO. NO ACTION NECESSARY
        LD      B.A
        DEC     B           :NUMBER OF PAIRS = COUNT-1
        LD      HL.41H      :POINT TO START OF ARRAY
PASS1   LD      A.(HL)      :GET ELEMENT FROM ARRAY
        INC     HL
        CP      (HL)        :IS IT LESS THAN NEXT ELEMENT?
        JR      NC.CNT      :NO. NO INTERCHANGE NECESSARY
        LD      C.1         :YES. SET INTERCHANGE FLAG
        LD      D.(HL)      :INTERCHANGE ELEMENTS
        LD      (HL).A
        DEC     HL
        LD      (HL).D
        INC     HL
CNT     DJNZ    PASS1
        DEC     C           :WAS INTERCHANGE FLAG SET?
        JR      NZ.SORT     :YES. DO ANOTHER PASS
        HALT
```

Clearly we cannot check all the possible input values for this program. Two other simple sets of data for debugging purposes are:

1) Two equal elements

$$(0040) = 02$$
$$(0041) = 00$$
$$(0042) = 00$$

2) Two elements already in decreasing order

$$(0040) = 02$$
$$(0041) = 01$$
$$(0042) = 00$$

14-26

INTRODUCTION TO TESTING

Program testing is closely related to program debugging. Surely some of the test cases will be the same as the test data used for debugging, such as:

- Trivial cases such as no data or a single element
- Special cases that the program singles out for some reason
- Simple examples that exercise particular parts of the program

In the case of the decimal to seven-segment conversion program, these cases cover all the possible situations. The test data consists of:

- The numbers 0 through 9
- The boundary case 10
- The random case 6B

The program does not distinguish any other cases. **Here debugging and testing are virtually the same.**

In the sorting program, the problem is more difficult. The number of elements could range from 0 to 255, and each of the elements could lie anywhere in that range. The number of possible cases is therefore enormous. Furthermore, the program is moderately complex. How do we select test data that will give us a degree of confidence in that program? **Here testing requires some design decisions.** The testing problem is particularly difficult if the program depends on sequences of real-time data. How do we select the data, generate it, and present it to the microcomputer in a realistic manner?

Most of the tools mentioned earlier for debugging are helpful in testing also. Logic or microprocessor analyzers can help check the hardware; simulators can help check the software. Other tools can also be of assistance, e.g.,

1) **I/O simulations** that can simulate a variety of devices from a single input and a single output device.
2) **In-circuit emulators** that allow you to attach the prototype to a development system or control panel and test it.
3) **ROM simulators** that have the flexibility of a RAM but the timing of the particular ROM or PROM that will be used in the final system.
4) **Real-time operating systems** that can provide inputs or interrupts at specific times (or perhaps randomly) and mark the occurrence of outputs. Real-time breakpoints and traces may also be included.
5) **Emulations** (often on micro programmable computers) that may provide real-time execution speed and programmable I/O.[5]
6) **Interfaces** that allow another computer to control the I/O system and test the microcomputer program.
7) **Testing programs** that check each branch in a program for logical errors.
8) **Test generation programs** that can generate random data or other distributions.

Formal testing theorems exist, but they are usually applicable only to very short programs.

You must be careful that the test equipment does not invalidate the test by modifying the environment. Often, test equipment may buffer, latch, or condition input and output signals. The actual system may not do this, and may therefore behave quite differently.

Furthermore, extra software in the test environment may use some of the memory space or part of the interrupt system. It may also provide error recovery and other features that will not exist in the final system. A software test bed must be just as realistic as a hardware test bed, since software failure can be just as critical as hardware failure.

Emulations and simulations are, of course, never precise. They are usually adequate for checking logic, but can seldom help test the interface or the timing. On the other hand, real-time test equipment does not provide much of an overview of the program logic and may affect the interfacing and timing.

SELECTING TEST DATA

Very few real programs can be checked for all cases. The designer must choose a sample set that in some sense describes the entire range of possibilities.

Testing should, of course, be part of the total development procedure. Top-down design and structured programming provide for testing as part of the design. This is called structured testing.[6]

STRUCTURED TESTING

Each module within a structured program should be checked separately. **Testing, as well as design, should be modular, structured, and top-down.**

But that leaves the question of selecting test data for a module. The designer must first list all special cases that a program recognizes. These may include:

TESTING SPECIAL CASES

- Trivial cases
- Equality cases
- Special situations

The test data should include all of these.

You must next identify each class of data that statements within the program may distinguish. These may include:

FORMING CLASSES OF DATA

- Positive or negative numbers
- Numbers above or below a particular threshold
- Data that does or does not include a particular sequence or character
- Data that is or is not present at a particular time

If the modules are short, the total number of classes should still be small even though each division is multiplicative; i.e., two two-way divisions result in four data classes.

You must now separate the classes according to whether the program produces a different result for each entry in the class (as in a table) or produces the same result for each entry (such as a warning that a parameter is above a threshold). In the discrete case, one may include each element if the total number is small or sample if the number is large. The sample should include all boundary cases and at least one case selected randomly. Random number tables are available in books, and random number generators are part of most computer facilities.

SELECTING DATA FROM CLASSES

You must be careful of distinctions that may not be obvious. For example, an 8-bit microprocessor will regard an 8-bit unsigned number greater than 127 as negative; the programmer must consider this when using conditional branches that depend on the Sign flag. You must also watch for instructions that do not affect flags, overflow in signed arithmetic, and the distinctions between address-length (16-bit) quantities and data-length (8-bit) quantities.

Testing Example 1: Sort Program

The special cases here are obvious:

- No elements in the array
- One element, magnitude may be selected randomly

The other special case to be considered is one in which elements are equal.

There may be some problem here with signs and data length. Note that the array itself must contain fewer than 256 elements. The use of the instruction LD C,1 or SET 1,C rather than DEC C to clear the interchange flag means that there will be no difficulty if the number of elements or interchanges exceeds 128.

We could check the effects of sign by picking half the regular test cases with numbers of elements between 128 and 255 and half between 2 and 127. All magnitudes should be chosen randomly so as to avoid unconscious bias as much as possible.

Testing Example 2: Self-Checking Numbers (see Chapter 8)

Here we will presume that a prior validity check has ensured that the number has the right length and consists of valid digits. Since the program makes no other distinctions, test data should be selected randomly. Here a random number table or random number generator will prove ideal; the range of the random numbers is 0 to 9.

TESTING PRECAUTIONS

The designer can simplify the testing stage by designing programs sensibly. You should use the following rules:

1) Try to eliminate trivial cases as early as possible without introducing unnecessary distinctions.
2) Minimize the number of special cases. Each special case means additional testing and debugging time.
3) Consider performing validity or error checks on the data prior to processing.
4) Be careful of inadvertent and unnecessary distinctions, particularly in handling signed numbers or using operations that refer to signed numbers.
5) Check boundary cases by hand. These are often a source of errors. Be sure that the problem definition specifies what is to happen in these cases.
6) Make the program as general as reasonably possible. Each distinction and separate routine increases the required testing.
7) Divide the program and design the modules so that the testing can proceed in steps in conjunction with the other stages of software development.[7]

CONCLUSIONS

Debugging and testing are the stepchildren of the software development process. Most projects leave far too little time for them and most textbooks neglect them. But designers and managers often find that these stages are the most expensive and time-consuming. Progress may be very difficult to measure or produce. Debugging and testing microprocessor software is particularly difficult because the powerful hardware and software tools that can be used on larger computers are seldom available for microcomputers.

The designer should plan debugging and testing carefully. We recommend the following procedure:

1) Try to write programs that can easily be debugged and tested. Modular programming, structured programming, and top-down design are useful techniques.

2) Prepare a debugging and testing plan as part of the program design. Decide early what data you must generate and what equipment you will need.

3) Debug and test each module as part of the top-down design process.

4) Debug each module's logic systematically. Use checklists, breakpoints, and the single-step mode. If the program logic is complex, consider using the software simulator.

5) Check each module's timing systematically if this is a problem. An oscilloscope can solve many problems if you plan the test properly. If the timing is complex, consider using a logic or microprocessor analyzer.

6) Be sure that the test data is a representative sample. Watch for any classes of data that the program may distinguish. Include all special and trivial cases.

7) If the program handles each element differently or the number of cases is large, select the test data randomly.[8]

8) Record all test results as part of the documentation. If problems occur, you will not have to repeat test cases that have already been checked.

REFERENCES

1. For more information about logic analyzers, see:

   R. L. Down, "Understanding Logic Analyzers," Computer Design, June 1977, pp. 188-191.

   W. A. Farnbach, "Bring up Your μP," Electronic Design, July 10, 1976, pp. 80-85.

   B. Farly, "Logic Analyzers Aren't All Alike," Electronic Design, Feb. 1, 1978, pp. 70-76.

   K. Pines, "What Do Logic Analyzers Do?," Digital Design, September 1977, pp. 55-77.

   N. A. Robin, "The Logic Analyzer: A Computer Troubleshooting Tool," Computer Design, March 1976, pp. 89-96.

   S. Runyon, "Focus on Logic and μP Analyzers," Electronic Design, February 1, 1977, pp. 40-50.

   A. Santoni, "The Latest Logic Analyzers Offer More Functions and Less Cost," Electronic Design, Feb. 1, 1978, pp. 26-32.

2. See W. J. Weller, Assembly Level Programming for Small Computers, Lexington Books, Lexington, Mass., 1975.

3. Some guidelines for debugging interrupt problems are given in R. L. Baldrige, "Interrupts Add Power, Complexity to Microcomputer System Design," EDN, August 5, 1977, pp. 67-73.

4. See C. Bass, "PLZ: A Family of System Programming Languages for Microprocessors," Computer, March 1978, pp. 34-39.

5. See, for example, H. R. Burris, "Time-Scaled Emulations of the 8080 Microprocessor," Proceedings of the 1977 National Computer Conference, pp. 937-946.

6. See D. A. Walsh, "Structured Testing," Datamation, July 1977, pp. 111-118.

7. Testing (and debugging) are also discussed in R. A. DeMillo et al., "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, April 1978, pp. 34-41 and in W. F. Dalton, "Design Microcomputer Software," Electronics, January 19, 1978, pp. 97-101.

8. Random numbers and their generation are discussed in T. G. Lewis, Distribution Sampling for Computer Simulation, Lexington Books, Lexington, Mass., 1975 and in R. A. Mueller, et al., "A Random Number Generator for Microprocessors," Simulation, April 1977, pp. 123-127.

Chapter 15
DOCUMENTATION AND REDESIGN

The working program is not the only requirement of software development. Adequate documentation is also an important part of a software product. Not only does documentation help the designer in the testing and debugging stages, it is also essential for later use and extension of the program. A poorly documented program will be difficult to maintain, use, or extend.

Occasionally, a program uses too much memory or executes too slowly. The designer must then improve it. This stage is called redesign, and requires that you concentrate on the parts of the program that can yield the most improvement.

SELF-DOCUMENTING PROGRAMS

Although no program is ever completely self-document- | **RULES FOR**
ing, some of the rules that we mentioned earlier can help. | **SELF-DOCUMENTING**
These include: | **PROGRAMS**

- Clear, simple structure with as few transfers of control (jumps) as possible
- Use of meaningful names and labels
- Use of names for I/O devices, parameters, numerical factors, etc.
- Emphasis on simplicity rather than on minor savings in memory usage, execution time, or typing

For example, the following program sends a string of characters to a teletypewriter:

```
        LD      A,(2000H)
        LD      B,A
        LD      HL,1000H
W:      LD      A,(HL)
        OUT     (6),A
        CALL    XXX
        INC     HL
        DJNZ    W
        HALT
```

Even without comments we can improve the program, as follows:

```
MESSG   EQU     1000H
COUNT   EQU     2000H
TTYSIO  EQU     6
        LD      A,(COUNT)
        LD      B,A
        LD      HL,MESSG
OUTCH:  LD      A,(HL)
        OUT     (TTYSIO),A
        CALL    BITDLY
        INC     HL
        DJNZ    OUTCH
        HALT
```

Surely this program is easier to understand than the earlier version. Even without further documentation, you could probably guess at the function of the program and the meanings of most of the variables. **Other documentation techniques cannot substitute for self-documentation.**

Some further notes on choosing names:

1) **Use the obvious name** when it is available, like TTY or CRT for output devices, START or RESET for addresses, DELAY or SORT for subroutines, COUNT or LENGTH for data.

2) **Avoid acronyms** like S16BA for SORT 16-BIT ARRAY. These seldom mean anything to anybody.

3) **Use full words** or close to full words when possible, like DONE, PRINT, SEND, etc.

4) **Keep the names** as **distinct** as possible.

COMMENTS

The most obvious form of additional documentation is the comment. However, few programs (even those used as examples in books), have effective comments. You should consider the following guidelines for good comments.

1) **Don't repeat the meaning of the instruction code.** Rather, explain the purpose of the instruction in the program. Comments like

```
        DEC B     ;B = B-1
```

add nothing to documentation. Rather, use

```
        DEC B     ;LINE NUMBER = LINE NUMBER-1
```

Remember that you know what the operation codes mean and anyone else can look them up in the manual. **The important point is to explain what task the program is performing.**

2) **Make the comments as clear as possible.** Do not use abbreviations or acronyms unless they are well-known (like ASCII, PIO, or UART) or standard (like no for number, ms for millisecond, etc.). Avoid comments like

```
        DEC B         ;LN = LN-1
                or
        DEC B         ;DEC LN BY 1
```

The extra typing simply is not all that expensive.

3) **Comment every important or obscure point.** Be particularly careful to mark operations that may not have obvious functions, such as

```
        AND 11011111B    ;TURN TAPE READER BIT OFF
    or
        ADD HL,DE        ;INDEX GRAY CODE TABLE
```

Clearly, I/O operations often require extensive comments. If you're not exactly sure of what an instruction does, or if you have to think about it, add a clarifying comment. The comment will save you time later and will be helpful in documentation.

4) **Don't comment the obvious.** A comment on each line simply makes it difficult to find the important points. Standard sequences like

```
INC      HL
DJNZ     SEARCH
```

need not be marked unless you're doing something special. One comment will often suffice for several lines, as in

```
RRCA                    ;SWAP DIGITS
RRCA
RRCA
RRCA

LD       A,C            ;EXCHANGE MOST SIGNIFICANT, LEAST
LD       C,B            .  SIGNIFICANT BYTES
LD       B,A
```

5) **Place comments on the lines to which they refer or at the start of a sequence.**

6) **Keep your comments up-to-date.** If you change the program. change the comments.

7) **Use standard forms and terms** in commenting. Don't worry about repetitiveness. Varied names for the same things are confusing, even if the variations are just COUNT and COUNTER. START and BEGIN, DISPLAY and LEDS, or PANEL and SWITCHES.

There's no real gain in not being consistent. The variations may seem obvious to you now, but may not be clear later; others will get confused from the very beginning.

8) **Make comments mingled with instructions brief.** Leave a complete explanation to header comments and other documentation. Otherwise, the program gets lost in the comments and you may have a hard time even finding it.

9) **Keep improving your comments.** If you come to one that you can't read or understand, take the time to change it. If you find that the listing is getting crowded, add some blank lines. The comments won't improve themselves; in fact, they will just become worse as you leave the task behind and forget exactly what you did.

10) **Before every major section, subsection, or subroutine, insert a number of comments describing the functions of the code that follows.** Care should be taken to describe all inputs, outputs, and side effects, as well as the algorithm employed.

11) It is good practice **when modifying working programs** to **use comments to indicate the date, author, and type of modification made.**

Remember, comments are important. Good ones will save you time and effort. Put some work into comments and try to make them as effective as possible.

Commenting Example 1: Multiple-Precision Addition

The basic program is:

```
            LD      A,(30H)
            LD      B,A
            LD      HL,41H
            LD      DE,51H
            AND     A
ADDWD:      LD      A,(DE)
            ADC     A,(HL)
            LD      (HL),A
            INC     DE
            INC     HL
            DJNZ    ADDWD
            HALT
```

First, comment the important points. These are typically initializations, data fetches, and processing operations. Don't bother with standard sequences like updating pointers and counters. Remember that names are clearer than numbers, so use them freely.

The new version of the program is:

```
;MULTIPRECISION ADDITION
;
;THIS PROGRAM PERFORMS MULTI-BYTE ADDITION
;
;INPUTS:     LOCATION 30H = LENGTH OF NUMBERS (IN BYTES)
            LOCATIONS 41H-50H = FIRST ADDEND IN LSB→MSB ORDER
;           LOCATIONS 51H-60H = SECOND ADDEND
;OUTPUTS:   LOCATIONS 41H-51H = SUM
;

LENGTH      EQU     30H
NUMB1       EQU     41H
NUMB2       EQU     51H
            LDA     LENGTH      ;COUNT = LENGTH OF NUMBERS (IN BYTES)
            LD      B,A
            LD      HL,NUMB1    ;START AT LSB'S OF 1ST NUMBER
            LD      DE,NUMB2    ;START AT LSB'S OF 2ND NUMBER
            AND     A
ADDWD:      LD      A,(DE)      ;GET 8 BITS OF 2ND NUMBER
            ADC     A,(HL)      ;ADD 8 BITS OF 1ST NUMBER
            LD      (HL),A      ;STORE RESULT IN 1ST NUMBER
            INC     DE
            INC     HL
            DJNZ    ADDWD
            HALT
```

Second, look for any instructions that might not have obvious functions and mark them. Here, the purpose of AND A is to clear the Carry the first time through.

Third, ask yourself whether the comments tell you what you would need to know if you wanted to use the program, e.g.:

1) Where is the program entered? Are there alternative entry points?

2) What parameters are necessary? How and in what form must they be supplied?

3) What operations does the program perform?

4) From where does it get the data?

5) Where does it store the results?

6) What special cases does it consider?

7) What does the program do about errors?

8) How does it exit?

Some of the questions may not be relevant to a particular program and some of the answers may be obvious. Make sure that you won't have to sit down and dissect the program to figure out what the answers are. Remember that too much explanation is just dead wood that you will have to clear out of the way. Is there anything that you would add to or subtract from this listing? If so, go ahead — you are the one who has to feel that the commenting is adequate and reasonable.

```
; MULTIPRECISION ADDITION
;
;THIS PROGRAM PERFORMS MULTI-BYTE ADDITION
;
;  INPUTS:  LOCATION 30H = LENGTH OF NUMBERS (IN BYTES)
;           LOCATIONS 41H-50H = FIRST ADDEND IN LSB→MSB ORDER
;           LOCATIONS 51H-60H = SECOND ADDEND
;  OUTPUTS:LOCATIONS 41H-51H = SUM
;
LENGTH    EQU     30H              ;LENGTH OF NUMBERS
NUMB1     EQU     41H              ;LSB'S OF 1ST NUMBER AND RESULT
NUMB2     EQU     51H              ;LSB'S OF 2ND NUMBER
          LDA     LENGTH           ;COUNT = LENGTH OF NUMBERS (IN BYTES)
          LD      B,A
          LD      HL,NUMB1         ;START AT LSB'S OF 1ST NUMBER
          LD      DE,NUMB2         ;START AT LSB'S OF 2ND NUMBER
          AND     A                ;CLEAR CARRY TO START
ADDWD:    LD      A,(DE)           ;GET 8 BITS OF 2ND NUMBER
          ADC     A,(HL)           ;ADD 8 BITS OF 1ST NUMBER
          LD      (HL),A           ;STORE RESULT IN 1ST NUMBER
          INC     DE
          INC     HL
          DJNZ    ADDWD
          HALT
```

Commenting Example 2: Teletypewriter Output

The basic program is:

```
          LD      A,(60H)
          ADD     A,A
          LD      B,11
TBIT:     OUT     (PIODRB),A
          RRA
          SCF
          CALL    BITDLY
          DJNZ    TBIT
          HALT
```

Commenting the important points and adding names gives:

```
;TELETYPEWRITER OUTPUT PROGRAM
;
;THIS PROGRAM PRINTS THE CONTENTS OF MEMORY LOCATION 60H TO THE
;   TELETYPEWRITER
;
;   INPUTS:   LOCATION 60H  = CHARACTER CODE
;   OUTPUTS:  NONE
;

TTYPIO    EQU      PIODRB
NBITS     EQU      11           ;NUMBER OF BITS PER CHARACTER
TDATA     EQU      60H          ;ADDRESS OF CHARACTER TO BE
                                ;   TRANSMITTED
          LD       A,(TDATA)    ;GET DATA
          ADD      A,A          ;SHIFT LEFT AND FORM START BIT
          LD       B,NBITS      ;COUNT = NUMBER OF BITS PER CHARACTER
TBIT:     OUT      (TTYPIO),A   ;SEND BIT TO TTY
          RRA                   ;UPDATE FOR NEXT BIT
          SCF                   ;FORM STOP BIT (LOGIC ONE)
          CALL     BITDLY       ;DELAY 1 BIT TIME
          DJNZ     TBIT
          HALT
```

Note how easily we could change this program so that it would transfer a whole string of data, starting at the address in locations DPTR and DPTR + 1 and ending with an "03" character (ASCII ETX). Furthermore, let us make the terminal a 30 character per second device with one stop bit (we will have to change subroutine BITDLY). Try making the changes before looking at the listing.

```
;STRING OUTPUT PROGRAM
;
;THIS PROGRAM OUTPUTS A STRING TO THE TERMINAL. TRANSMISSION CEASES
;   WHEN AN ASCII ETX (30H) IS ENCOUNTERED
;
;   INPUTS:   LOCATIONS 60H-61H CONTAIN ADDRESS OF
;             STRING TO OUTPUT
;   OUTPUTS:  NONE
;

DPTR      EQU      60H          ;LOCATION OF OUTPUT BUFFER START
                                ;   ADDRESS
ENDCH     EQU      03           ;ENDING CHARACTER = ASCII ETX
NBITS     EQU      11           ;NUMBER OF BITS PER CHARACTER
TTYPIO    EQU      PIODRB
          LD       HL,(DPTR)    ;GET STARTING ADDRESS OF STRING
TCHAR:    LD       A,(HL)       ;GET A CHARACTER
          CP       ENDCH        ;IS IT ENDING CHARACTER?
          JR       Z,DONE       ;YES, DONE
          ADD      A,A          ;SHIFT DATA LEFT AND FORM START BIT
          LD       B,NBITS      ;COUNT = NUMBER OF BITS PER CHARACTER
TBIT:     OUT      (TTYPIO),A   ;SEND BIT TO TTY
          RRA                   ;UPDATE FOR NEXT BIT
          SCF                   ;FORM STOP BIT (LOGIC ONE)
          CALL     BITDLY       ;DELAY 1 BIT TIME
          DJNZ     TBIT
          INC      HL
          JR       TCHAR
DONE:     HALT
```

Good comments can make it easy for you to change a program to meet new requirements. For example, try changing the last program so that it:

- Starts each message with ASCII STX (02 hex) followed by a three-digit identification code stored in memory locations 0030 through 0032
- Adds no start or stop bits
- Waits 1 ms between bits
- Transmits 40 characters, starting with the one located at the address in DPTR and DPTR+1
- Ends each message with two consecutive ASCII ETXs (03 hex)

FLOWCHARTS AS DOCUMENTATION

We have already described the use of flowcharts as a design tool in Chapter 13. Flowcharts are also useful in documentation, particularly if:

> **HINTS FOR USING FLOWCHARTS**

- They are not so detailed as to be unreadable
- Their decision points are clearly explained and marked
- They include all branches
- They correspond to the actual program listings

Flowcharts are helpful if they give you an overall picture of the program. They are not helpful if they are just as difficult to read as an ordinary listing.

STRUCTURED PROGRAMS AS DOCUMENTATION

A structured program can serve as documentation for an assembly language program if:

- You describe the purpose of each section in the comments
- You make it clear which statements are included in each conditional or loop structure by using indentation and ending markers
- You make the total structure as simple as possible
- You use a consistent, well-defined language

The structured program can help you to check the logic or improve it. Furthermore, since the structured program is machine-independent, it can also aid you in implementing the same task on another computer.

MEMORY MAPS

A memory map is simply a list of all the memory assignments in a program. The map allows you to determine the amount of memory needed, the locations of data or subroutines, and the parts of memory not allocated. The map is a handy reference for finding storage locations and entry points and for dividing memory between different routines or programmers. The map will also give you easy access to data and subroutines if you need them in later extensions or in maintenance. Sometimes a graphical map is more helpful than a listing.

A typical map would be:

Program Memory		
Address	Routine	Purpose
0000-0002	RESET	TRANSFERS CONTROL TO MAIN PROGRAM IN LOCATION 40 HEX
0038-003A	INTRPT	TRANSFERS CONTROL TO INTERRUPT SERVICE IN LOCATION 300 HEX
0040-0265	MAIN	MAIN PROGRAM
0270-027F	DELAY	DELAY PROGRAM
0280-0290	DSPLY	DISPLAY CONTROL PROGRAM
0300-0340	KEYIN	INTERRUPT CONTROL PROGRAM FOR KEYBOARD
Data Memory		
1000	NKEYS	NUMBER OF KEYS
1001-1002	KPTR	KEYBOARD BUFFER POINTER
1003-1041	KBFR	KEYBOARD BUFFER
1042-1051	DBFR	DISPLAY BUFFER
1052-105F	TEMP	TEMPORARY STORAGE
10E0-10FF	STACK	RAM STACK

The map may also list additional entry points and include a specific description of the unused parts of memory.

PARAMETER AND DEFINITION LISTS

Parameter and definition lists at the start of the program and each subroutine make understanding and changing the program far simpler. The following rules can help:

1) **Separate RAM locations, I/O units, parameters, definitions, and memory system constants.**

2) **Arrange lists alphabetically when possible,** with a description of each entry.

3) **Give each parameter that might change a name and include it in the lists.** Such parameters may include timing constants, inputs or codes corresponding to particular keys or functions, control or masking patterns, starting or ending characters, thresholds, etc.

4) **Make the memory system constants into a separate list.** These constants will include Reset and interrupt service addresses, the starting address of the program, RAM areas, Stack areas, etc.

5) **Give each port used by an I/O device a name,** even though devices may share ports in the current system. The separation will make expansion or reconfiguration much simpler.

A typical list of definitions will be:

```
:
:MEMORY SYSTEM CONSTANTS
.
RESET       EQU    0          :RESET ADDRESS
INTRP       EQU    38H        :INTERRUPT ENTRY
START       EQU    40H        :START OF MAIN PROGRAM
KEYIN       EQU    300H       :KEYBOARD INTERRUPT PROGRAM
RAMST       EQU    1000H      :START OF DATA STORAGE
STKPTR      EQU    1100H      :START OF STACK

:
. I/O UNITS
.
DSPLY       EQU    0E0H       :OUTPUT PIO FOR DISPLAYS
KBDIN       EQU    0E1H       :INPUT PIO FOR KEYBOARD
KBDOUT      EQU    0E0H       :OUTPUT PIO FOR KEYBOARD
TTYPIO      EQU    0F0H       ;TTY DATA PORT

.
;RAM LOCATIONS
;
            ORG    RAMST
NKEYS       DEFS   1          :NUMBER OF KEYS
KBDPTR      DEFS   2          :KEYBOARD BUFFER POINTER
KBDBFR      DEFS   40H        :KEYBOARD INPUT BUFFER
DSPBFR      DEFS   10H        :DISPLAY DATA BUFFER
TEMP        DEFS   14H        ;TEMPORARY STORAGE

:
;PARAMETERS
.
BOUNCE      EQU    2          :DEBOUNCING TIME IN MS
GOKEY       EQU    10         :IDENTIFICATION OF 'GO' KEY
MSCNT       EQU    133        :COUNT FOR 1 MS DELAY
OPEN        EQU    0FH        :PATTERN FOR OPEN KEYS
TPULS       EQU    1          :PULSE LENGTH FOR DISPLAYS IN MS

:
;DEFINITIONS
.
ALL1        EQU    0FFH       :ALL ONES PATTERN
STCON       EQU    80H        :START CONVERSION PULSE
```

Of course, the RAM entries will usually not be in alphabetical order, since the designer must order these so as to minimize the number of address changes required in the program.

LIBRARY ROUTINES

Standard documentation of subroutines will allow you to build up a library of useful programs. The idea is to make these programs easily accessible. A standard format will allow you or anyone else to see at a glance what the program does. The best procedure is to make up a standard form and use it consistently. Save these programs in a well-organized manner (for example, according to processor, language, and type of program), and you will soon have a useful set. But **remember that without organization and proper documentation, using the library may be more difficult than rewriting the program from scratch.** Debugging a system requires a precise understanding of all the effects of each subroutine.

Among the information that you will need in the standard form is:

- Purpose of the program
- Processor used
- Language used
- Parameters required and how they are passed to the subroutine
- Results produced and how they are passed to the main program
- Number of bytes of memory used
- Number of clock cycles required. This number may be an average or a typical figure, or it may vary widely. Actual execution time will, of course, depend on the processor clock rate
- Registers affected
- Flags affected
- A typical example
- Error handling
- Special cases
- Documented program listing

If the program is complex, the standard library form should also include a general flowchart or a structured program. As we have mentioned before, a library program is most likely to be useful if it performs a single distinct function in a reasonably general manner.

LIBRARY EXAMPLES

Library Example 1: Sum of Data

Purpose: The program SUM8 computes the sum of a set of 8-bit unsigned binary numbers.

Language: Z80 assembler.

Initial Conditions: Starting address of set of numbers in Register Pair HL, length of set in Accumulator.

Final Conditions: Sum in Accumulator.

Requirements:

Memory - 7 bytes.
Time - 13 + 26N clock cycles, where N is the length of the set of numbers.
Registers - A, B, H, L.
All flags affected.

Typical Case: (all data in hexadecimal)

```
Start:
     HL  =  0050
      A  =  03
   (0050) =  27
   (0051) =  3E
   (0052) =  26
End:
      A  =  8B
```

Error Handling: Program ignores all carries. Carry bit reflects only the last operation. Initial contents of Accumulator must be 1 or more.

Listing:

```
;
;SUM OF 8-BIT DATA
;
SUM8:   LD      B,A         ;COUNT = LENGTH OF DATA BLOCK
        SUB     A           ;SUM = ZERO
ADD8:   ADD     A,(HL)      ;SUM = SUM + DATA ENTRY
        INC     HL
        DJNZ    ADD8
        RET
```

Library Example 2: Decimal-to-Seven-Segment Conversion

Purpose: The program SEVEN converts a decimal number to a seven-segment display code.

Language: Z80 assembler.

Initial Conditions: Data in Accumulator.

Final Conditions: Seven-segment code in Accumulator.

Requirements:

Memory   - 26 bytes, including the seven-segment table (10 entries).

Time   - 74 clock cycles if the data is valid, 40 if it is not.

Registers - A, B, D, E, H, L.

All flags affected.

Input data in Accumulator is destroyed.

Typical Case: (data in hexadecimal)

```
Start:
      A  =  05
End:
      A  =  66
```

Error Handling: Program returns zero in the Accumulator if data is not a decimal digit.

Listing:

```
;
;DECIMAL TO SEVEN-SEGMENT CONVERSION
;
SEVEN:    LD      B,0          ;GET ERROR CODE TO BLANK DISPLAY
          CP      10           ;IS DATA A DECIMAL DIGIT?
          JR      NC,DONE      ;NO, KEEP ERROR CODE
          LD      L,A          ;YES, MAKE DATA INTO A 16-BIT INDEX
          LD      H,0
          LD      DE,SSEG      ;GET BASE ADDRESS OF 7-SEGMENT TABLE
          ADD     HL,DE        ;FIND ELEMENT BY INDEXING
          LD      B,(HL)       ;GET 7-SEGMENT CODE FROM TABLE
DONE:     LD      A,B          ;SAVE 7-SEGMENT CODE OR ERROR CODE
          RET
SSEG:     DEFB    3FH
          DEFB    06H
          DEFB    5BH
          DEFB    4FH
          DEFB    66H
          DEFB    6DH
          DEFB    7DH
          DEFB    07H
          DEFB    7FH
          DEFB    6FH
```

Library Example 3: Decimal Sum

Purpose: The program DECSUM adds two multi-word decimal numbers.

Language: Z80 assembler.

Initial Conditions: Address of LSBs of one number in Register Pair HL, address of LSBs of other number in Register Pair DE, length of numbers (in bytes) in A. Numbers arranged starting with LSBs at lowest address.

Final Conditions: Sum replaces number with starting address in Register Pair HL.

Requirements:

```
Memory    -   11 bytes.
Time      -   13 + 50N clock cycles, where N is the number of
              bytes involved.
Registers -   A, B, D, E, H, L.
All flags affected. Carry shows if sum produced a carry.
```

Typical Case: (data in hexadecimal)

```
Start:
       HL  =  0060
       DE  =  0050
        A  =  2
    (0060) =  34
    (0061) =  55
    (0050) =  88
    (0051) =  15
End:
    (0060) =  22
    (0061) =  71
    CARRY  =  0
```

Error Handling: Program does not check the validity of decimal inputs. Accumulator must be 1 or greater.

Listing:

```
DECSUM:   LD    B,A        ;COUNT = LENGTH OF NUMBERS (IN BYTES)
          AND   A          ;CLEAR CARRY TO START
DECADD:   LD    A,(DE)     ;GET 2 DECIMAL DIGITS FROM STRING 2
          ADC   A,(HL)     ;ADD PAIR OF DIGITS FROM STRING 1
          DAA              ;MAKE ADDITION DECIMAL
          LD    (HL),A     ;STORE RESULT IN STRING 1
          INC   DE
          INC   HL
          DJNZ  DECADD
          RET
```

TOTAL DOCUMENTATION

Complete documentation of microprocessor software will include all or most of the elements that we have mentioned. So, the total documentation package may involve:

> **DOCUMENTATION PACKAGE**

- **General flowcharts**
- **A written description of the program**
- **A list of all parameters and definitions**
- **A memory map**
- **A documented listing of the program**
- **A description of the test plan and test results**

The documentation may also include:

- **Programmers' flowcharts**
- **Data flowcharts**
- **Structured programs**

The documentation procedures outlined above are the minimal acceptable set of documents for non-production software. Production software demands even greater documentation efforts. The following documents should also be produced:

- Program Logic Manual
- User Guide
- Maintenance Manual

The program logic manual expands on the written explanation produced with the software. It should be written for a technically competent individual who may not possess the detailed knowledge assumed in the written explanation in the software. The program logic manual should explain what the design goals of the system were, what algorithms were chosen to implement these goals, and what tradeoffs had to be made in achieving them.

It should then explain in great detail what data structures were employed and how they are manipulated. It should provide a step-by-step guide to the inner workings of the code. Finally, it should contain any special tables or graphs that help explain any of the concepts embodied in the code. Code conversion charts, state diagrams, translation matrices, and flowcharts should be included.

The user guide is probably the most important and most overlooked piece of documentation. No matter how well a system is designed, it is useless if no one can use it effectively. The user guide should provide all users, sophisticated and unsophisticated, with an introduction to the system. It should then provide detailed ex-

planations of system features and their use. Use plenty of examples because a good example can crystallize the information contained in many pages of text. Step-by-step directions should be given. Test the user guide, i.e., try out the step-by-step usage procedures as you have documented them. Programmers with detailed knowledge of a system's design often take shortcuts that are not at all apparent to the general reader. An entire book could be written about the writing of user guides, and further discussion is beyond the scope of this book. However, remember that you can never spend too much effort in preparing a user guide, because it will be the most used of all system documents.

The maintenance manual is designed for the programmer who has to modify the system. It should outline step-by-step procedures for those reconfigurations designed into the system. In addition, it should outline any provisions placed into code for future expansion.

Documentation should not be taken lightly or postponed until the end of the software development. Proper documentation, combined with proper programming practices, is not only an important part of the final product but can also make development simpler, faster, and more productive. The designer should make consistent and thorough documentation part of every stage of software development.

REDESIGN

Sometimes the designer may have to squeeze the last microsecond of speed or the last byte of extra memory out of a program. As larger single-chip memories have become available, the memory problem has become less serious. The time problem, of course, is serious only if the application is time-critical; in many applications the microprocessor spends most of its time waiting for external devices, and program speed is not a major factor.

Squeezing the last bit of performance out of a program is seldom as important as some writers would have you believe. In the first place, the practice is expensive for the following reasons:

| COST OF |
| REDESIGN |

1) It requires extra programmer time, which is often the single largest cost in software development.
2) It sacrifices structure and simplicity with a resulting increase in debugging and testing time.
3) The programs require extra documentation.
4) The resulting programs will be difficult to extend, maintain, or re-use.

In the second place, the lower per-unit cost and higher performance may not really be important. Will the lower cost and higher performance really sell more units? Or would you do better with more user-oriented features? **The only applications that would seem to justify the extra effort and time are very high-volume, low-cost and low-performance applications where the cost of an extra memory chip will far outweigh the cost of the extra software development.** For other applications, you will find that you are playing an expensive game for no reason.

However, if you must redesign a program, the following hints will help. First, determine how much more performance or how much less memory usage is necessary. If the required improvement is 25% or less, you may be

| MAJOR OR |
| MINOR |
| REORGANIZATION |

able to achieve it by reorganizing the program. If it is more than 25%, you have made a basic design error; you will need to consider drastic changes in hardware or software. We will deal first with reorganization and later with drastic changes. You should also look at Chapter 5 of Z80 Programming for Logic Design for some examples.

Note particularly that saving memory can be critical if it allows a program to fit into the limited amount of ROM and RAM available in a simple one-chip or two-chip microcomputer. The hardware cost for small systems can thus be substantially reduced, if their requirements can be limited to the memory size and I/O limitations of that particular one-chip or two-chip system.

REORGANIZING TO USE LESS MEMORY

The following procedures will reduce memory usage for Z80 assembly language programs:

<div style="border:1px solid">SAVING MEMORY</div>

1) **Replace repetitious in-line code with subroutines.** Be sure, however, that the CALL and RETURN instructions do not offset most of the gain. Note that this replacement usually results in slower programs because of the time spent in transferring control back and forth.

2) **Use register operations when possible.** But remember the cost of the extra initialization.

3) **Use the Stack when possible.** The Stack Pointer is automatically updated after each use so that no explicit updating instructions are necessary.

4) **Eliminate Jump instructions.** Try to reorganize the program or use indirect jumps (JP (HL) or JP (IX or IY)), RST, or RETURN instructions.

5) **Take advantage of addresses that you can manipulate as 8-bit quantities.** These include page zero and addresses that are multiples of 100 hexadecimal. For example, you might try to place all ROM tables in one 100_{16}-byte section of memory, and all RAM variables into another 100_{16}-byte section.

6) **Organize data and tables so that you can address them without worrying about address calculation carries or without any actual indexing.** This will again allow you to manipulate 16-bit addresses as 8-bit quantities. See pages 5-1 to 5-6 of Z80 Programming for Logic Design for an example.

7) **Use the 16-bit instructions to replace two separate 8-bit operations.** This may be particularly useful in initialization or storing results.

8) **Use leftover results from previous sections of the program.**

9) **Take advantage of such instructions** as INC (HL), DCR (HL), LD (HL), RL (HL), and RR (HL), **which operate directly on memory locations without using registers.**

10) **Use INC or DEC to set or reset flag bits.**

11) **Use relative jumps rather than jumps with direct addressing.**

12) **Take advantage of the Block Move, Block Search, and Block I/O instructions** whenever you are handling blocks of data.

13) **Watch for special short forms of instructions** such as the Accumulator shifts (RLCA, RLA, RRCA, and RRA) and DJNZ.

14) **Use algorithms rather than tables** to calculate arithmetic or logical expressions and to perform code conversions. Note that this replacement may result in slower programs.

15) **Reduce the size of mathematical tables by interpolating** between entries. Here again, we are saving memory at the cost of execution time.

16) **Take advantage of the alternate register set to cut down on the use of storage.** This can save time as well.

Although some of the methods that reduce memory usage also save time, you can generally save an appreciable amount of time only by concentrating on frequently executed loops. Even

<div style="border:1px solid">SAVING EXECUTION TIME</div>

completely eliminating an instruction that is executed only once can save at most a few microseconds. But a savings in a loop that is executed frequently will be multiplied many times over.

So, **if you must reduce execution time, proceed as follows:**

1) **Determine how frequently each program loop is executed.** You can do this by hand or by using the software simulator or another testing method.

2) **Examine the loops in the order determined by their frequency of execution,** starting with the most frequent. Continue through the list until you achieve the required reduction.

3) **First, see if there are any operations that can be moved outside the loop,** i.e., repetitive calculations, data that can be placed into a register or the Stack, addresses that can be placed into register pairs or index registers, special cases or errors that can be handled elsewhere, etc. Note that this will require extra initialization and memory but will save time.

4) **Try to eliminate Jump statements.** These are very time-consuming. Or, use jumps with direct addressing that require more memory but less time than jumps with relative addressing.

5) **Replace subroutines with in-line code.** This will save at least a CALL and a RETURN instruction.

6) **Use the Stack for temporary data storage.**

7) **Use any of the hints mentioned in saving memory that also decrease execution time.** These include the use of block handling instructions, 8-bit addresses, 16-bit instructions, RST, special short forms of instructions, etc.

8) **Do not even look at instructions that are executed only once.** Any changes that you make in such instructions only invite errors for no appreciable gain.

9) **Avoid indexed and relative addressing whenever possible** because they take extra time.

10) **Use tables rather than algorithms;** make the tables handle as much of the tasks as possible even if many entries must be repeated.

MAJOR REORGANIZATIONS

If you need more than a 25% increase in speed or decrease in memory usage, do not try reorganizing the code. Your chances of getting that much of an improvement are small unless you call in an outside expert. You are generally better off making a major change.

The most obvious change is a better algorithm. Particularly if you are doing sorts, searches, or mathematical calculations, you may be able to find a faster or shorter method in the literature. Libraries of algorithms are available in some journals and from professional groups. See, for example, References 1 through 10 at the end of this chapter.

```
BETTER
ALGORITHMS
```

More hardware can replace some of the software. Counters, shift registers, arithmetic units, hardware multipliers, and other fast add-ons can save both time and memory. Calculators, UARTs, keyboards, encoders, and other slower add-ons may save memory even though they operate slowly. Compatible parallel and serial interfaces, and other devices specially designed for use with the Z80 may save time by taking some of the burden off the CPU.

Other changes may help as well:

1) **A CPU with a longer word will be faster** if the data is long enough. Such a CPU will use less total memory. 16-bit processors, for example, use memory more efficiently than 8-bit processors, since more of their instructions are one word long.

2) **Versions of the CPU may exist that operate at higher clock rates.** But remember that you will need faster memory and I/O ports, and you will have to adjust any delay loops.

3) **Two CPUs may be able to do the job in parallel** or separately if you can divide the job and solve the communications problem.

4) **A specially microprogrammed processor may be able to execute the same program much faster.** The cost, however, will be much higher even if you use an off-the-shelf emulation.

5) **You can make tradeoffs between time and memory.** Lookup tables and function ROMs will be faster than algorithms, but will occupy more memory.

This kind of problem, in which a large improvement is necessary, usually results from lack of adequate planning in the definition and design stages. In the problem definition stage you should determine which processor and methods will be adequate to handle the problem. If you misjudge, the cost later will be high. A cheap solution may result in an unwarranted expenditure of expensive development time. Do not try to just get by; the best solution is usually to do the proper design and chalk a failure up to experience. If you have followed such methods as flowcharting, modular programming, structured programming, top-down design, and proper documentation, you will be able to salvage a lot of your effort even if you have to make a major change.

REFERENCES

1. Collected Algorithms from ACM. ACM, Inc., P. O. Box 12105, Church Street Station, New York 10249.

2. Chen, T. C., "Automatic Computation of Exponentials, Logarithms, Ratios, and Square Roots," IBM Journal of Research and Development, Volume 18, pp. 380-388, July, 1972.

3. H. Schmid, Decimal Computation, Wiley-Interscience, New York, 1974.

4. Knuth, D. E., The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1967.

5. Knuth, D. E., The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Addison-Wesley, Reading, Mass., 1969.

6. Knuth, D. E., The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

7. Carnahan, B. et al., Applied Numerical Methods, Wiley, New York, 1969.

8. Despain, A. M. "Fourier Transform Computers Using CORDIC Iterations," IEEE Transactions on Computers, October 1974, pp. 993-1001.

9. Luke, Y. L., Algorithms for the Computation of Mathematical Functions, Academic Press, New York, 1977.

10. Hwang, K., Computer Arithmetic, Wiley, New York, 1978.

11. Dollhoff, T., "Microprocessor Software: How to Optimize Timing and Memory Usage. Part Four: Techniques for the Zilog Z80," Digital Design, February 1977, pp. 44-51.

Chapter 16
SAMPLE PROJECTS

PROJECT #1: A Digital Stopwatch

Purpose: This project is a digital stopwatch. The operator enters two digits (minutes and tenths of minutes) from a calculator-like keyboard and then presses the GO key. The system counts down the remaining time on two seven-segment LED displays (see Chapter 11 for a description of unencoded keyboards and LED displays).

<table><tr><td>STOPWATCH
INPUT
PROCEDURE</td></tr></table>

Hardware: The project uses one input port and one output port (one Z80 Parallel Input/Output Device or PIO), two seven-segment displays, a 12-key keyboard, a 7404 inverter, and either a 7400 NAND gate or a 7408 AND gate, depending on the polarity of the seven-segment displays. The displays may require drivers, inverters, and resistors, depending on their polarity and configuration.

The hardware is organized as shown in Figure 16-1. Output lines 0, 1, and 2 are used to scan the keyboard. Input lines 0, 1, 2, and 3 are used to determine whether any keys have been pressed. Output lines 0, 1, 2, and 3 are used to send BCD digits to the seven-segment decoder/drivers. Output line 4 is used to activate the LED displays (if line 4 is '1', the displays are lit). Output line 5 is used to select the left or right display; output line 5 is '1' if the left display is being used, '0' if the right display is being used. Thus, the common line on the left display should be active if line 4 is '1' and line 5 is '1', while the common line on the right display should be active if line 4 is '1' and line 5 is '0'. Output line 6 controls the right-hand decimal point on the left display. It may be driven with an inverter or simply left on.

Keyboard Connections: The keyboard is a simple calculator keyboard available for 50¢ from a local source. It consists of 12 unencoded key-switches arranged in four rows of three columns each. Since the wiring of the keyboard does not coincide with the observed rows and columns, the program uses a table to identify the keys. Tables 16-1 and 16-2 contain the input and output connections for the keyboard. The decimal point key is present for operator convenience and for future expansion; the current program does not actually use the key.

In an actual application, the keyboard would require pullup resistors to ensure that the inputs would actually be read as logic '1's when the keys were not being pressed. It would also require current-limiting resistors or diodes on the output port to avoid damaging the drivers in the case where two outputs were driving against each other. This could occur if two keys in the same row were pressed at the same time, thus connecting two different column outputs.
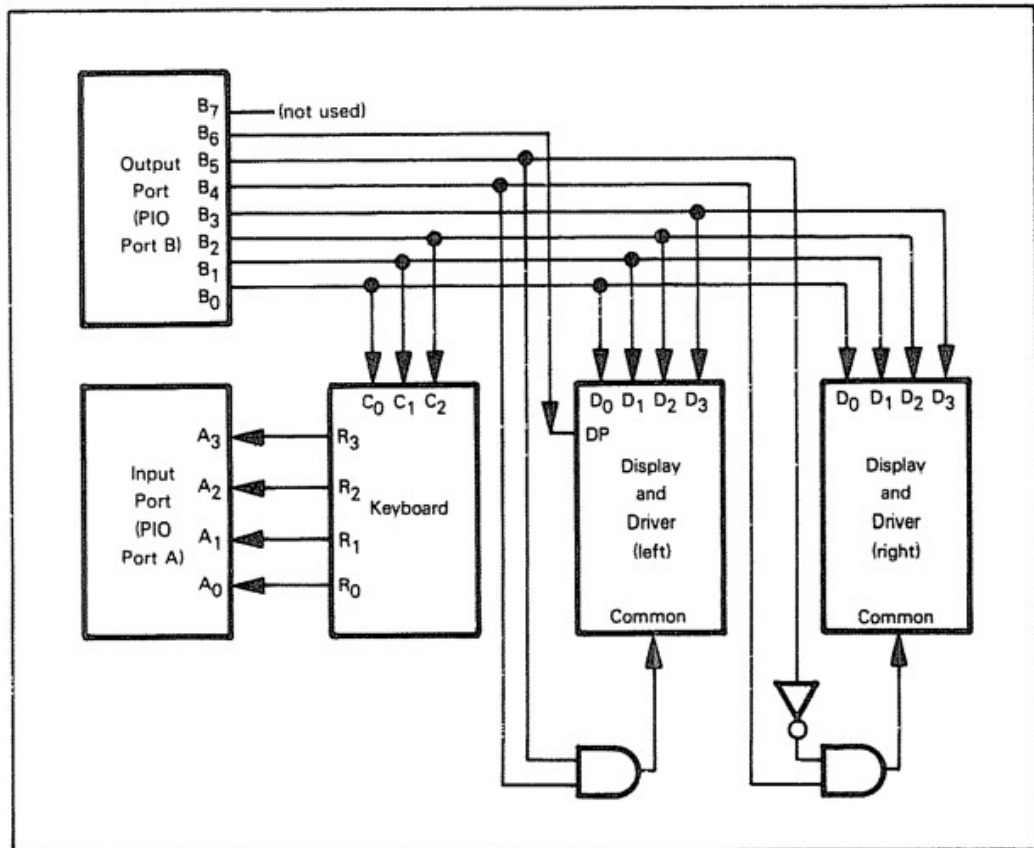
Figure 16-1. Digital Stopwatch I/O Configuration

Table 16-1. Input Connections for Stopwatch Keyboard

Input Bit	Keys Connected
0	'3', '5', '8'
1	'2', '6', '9'
2	'0', '1', '7'
3	'4', '.', 'GO'

Table 16-2. Output Connections for Stopwatch Keyboard

Output Bit	Keys Connected
0	'0', '2', '3', '4'
1	'1', '8', '9', 'GO'
2	'5', '6', '7', '.'

16-2

General Program Flowchart:

16-3
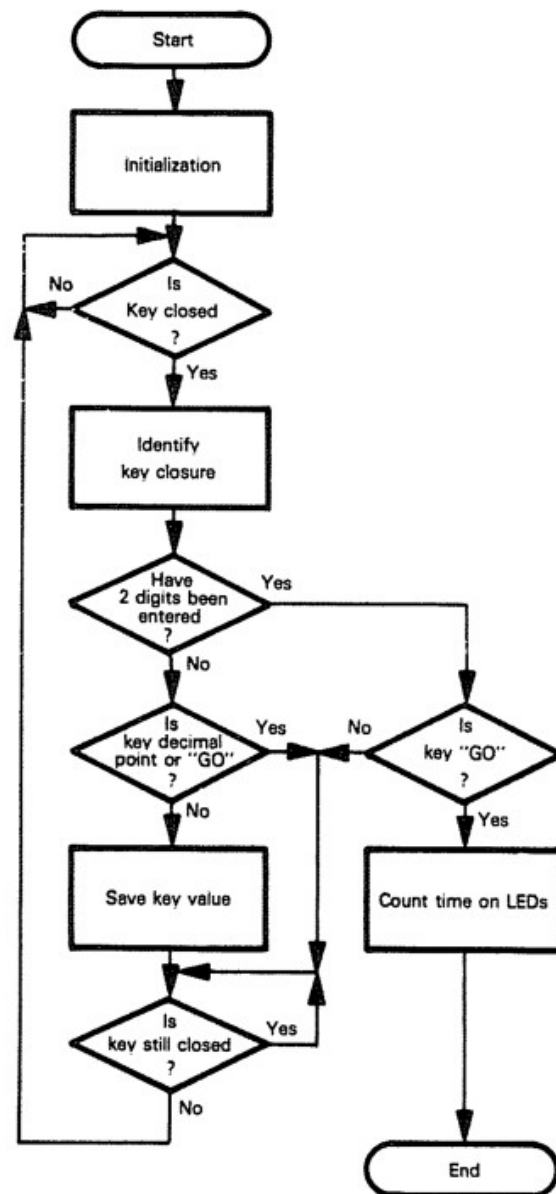
Display Connections: The displays are seven-segment displays with their own integral decoders. A typical example would be the Texas Instruments TIL309 device, which has an internal TTL MSI chip with latch, decoder, and driver. Clearly, standard seven-segment displays would be cheaper but would require some additional software (the seven-segment conversion routine shown in Chapter 7). Data is entered into the display as a single binary coded decimal digit; the digits are represented as shown in Figure 11-15. The decimal point is a single LED that is turned on when the decimal point input is a logic '1' You can find more information about displays in References 10 and 11 at the end of this chapter.

Program Description:

The program is modular and has several subroutines. The emphasis is on clarity and generality rather than efficiency; obviously, the program does not utilize the full capabilities of the Z80 processor. Each section of the listing will now be described in detail.

1) **Introductory Comments**

   The introductory comments fully describe the program; these comments are a reference so that other users can easily apply, extend, and understand the program. Standard formats, indentations, and spacings increase the readability of the program.

2) **Variable Definitions**

   All variable definitions are placed at the start of the program so that they can easily be checked and changed. Each variable is placed in a list alphabetically with other variables of the same type; comments describe the meaning of each variable. The categories are:

   a) Memory system constants that may vary from system to system depending on the memory space allocated to different programs or types of memories

   b) Temporary storage (RAM) used for variables

   c) I/O (PIO) port addresses

   d) Definitions

   The memory system constants are placed in the definitions so that the user may relocate the program, temporary storage, and memory stack without making any other changes. The memory constants can be changed to accommodate other programs or to coincide with a particular system's allocation of ROM and RAM addresses.

   Temporary storage is allocated by means of DEFS (Define Storage) pseudo-operations. An ORG (origin) pseudo-operation places the temporary storage locations in a particular part of memory. No values are placed in these locations so that the program could eventually be placed in ROM or PROM and the system could be operated from power-on reset without reloading.

   Each port address occupied by a PIO is named so that the addresses can easily be changed to handle varied configurations. The naming also serves to clearly distinguish control registers from data registers.

   The definitions clarify the meaning of certain constants and allow parameters to be changed easily. Each definition is given in the form (binary, hex, octal, ASCII, or decimal) in which its meaning is the clearest. Parameters (such as debounce time) are placed here so that they can be varied with system needs.

3) **Initialization**

Memory location 0 (the reset location on the Z80 microprocessor) contains a jump to the starting address of the main program. The main program can thus be placed anywhere in memory and reached via a "RESET" signal.

The initialization consists of four steps:

a) Place a starting value in the Stack Pointer. The Stack is used only to store subroutine return addresses.

b) Configure the PIO control registers.

c) Start the number of digit keys pressed at zero.

d) Initialize the location where the next digit key pressed will be saved to the start of the digit key array. An indirect procedure is used, in which KEYAD contains the address in which the next digit will be placed. Each time a digit key is recognized, the contents of KEYAD are incremented so that the next digit key will be placed into the next memory location.

4) **Look for Key Closure**
   Flowchart:
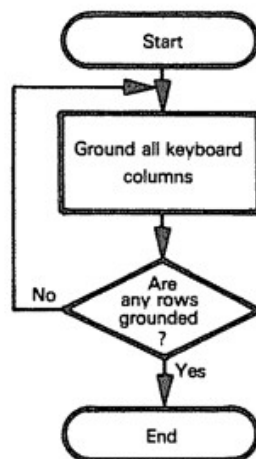
Key closures are identified by grounding all the keyboard columns and then checking for grounded rows (i.e., column-to-row switch closures). Note that the program does not assume that the unused input bits are all high; instead, the bits attached to the keyboard are isolated with a logical AND instruction.

5) **Debounce Key**

The program debounces the key closure in software by waiting for two milliseconds. This is usually long enough for a clean contact to be made. Subroutine DELAY simply counts with Register C for 1 millisecond. The number of milliseconds is in the Accumulator. DELAY would have to be adjusted if a slower clock or slower memories were being used. You could make the change simply by redefining the constant MSCNT.

6) **Identify Key Closure**

   Flowchart:
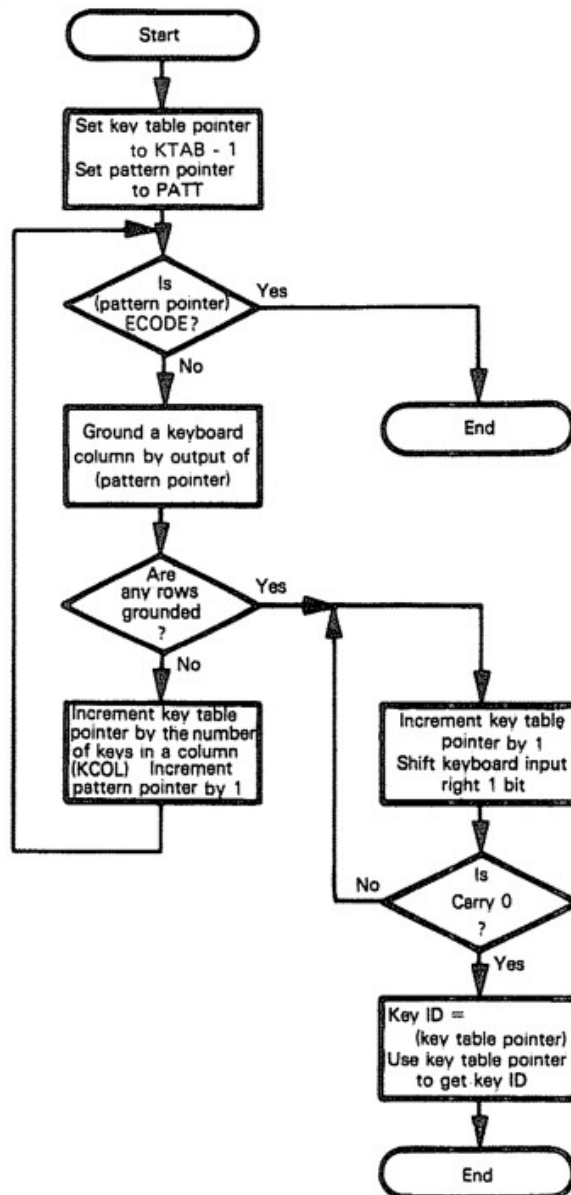
The particular key closed is identified by grounding single columns and observing whether a closure is found. Once a closure is found (so the key column is known), the key row can be determined by shifting the input.

The patterns required to ground single keyboard columns are in a table PATT in memory. The final pattern in the table is a marker (ECODE) which indicates that all the columns have been grounded without a closure being found. This pattern also indicates to the main program that the closure could not be identified (e.g., the key closure ended or a hardware error occurred before we could find the closure).

The key identifications are in table KTAB in memory. The keys in the first column (attached to the least significant output bit) are followed by those in the second column, etc.

Within a column, the key in the row attached to the least significant input bit is first, etc. Thus, each time a column is scanned without finding a closure, the number of keys in a column (NROWS) must be added to the key table pointer in order to move to the next column. The key table pointer is also incremented by one before each bit in the row inputs is examined; this process stops when a zero input is found. Note that the key table pointer is started one location before the table, since it is always incremented once in the search for the proper row.

If we cannot identify the key closure, we simply ignore it and look for another closure.

7) Act on Key Identification

If the program has enough digits (two in this simple case), it looks only for the GO key and ignores all other keys. If it finds a digit key, it saves the value in the key array, increments the number of digit keys pressed, and increments the key array pointer.

If the entry is not complete, the program must wait for the key closure to end so that the system will not read the same closure again. The user must wait between key closures (i.e., release one key before pressing another one). Note that the program will identify double key closures as one key or the other, depending on which closure the identification routine finds first. An improved version of this program would display digits as they were entered and would allow the user to omit a leading or trailing zero, (i.e., key in ".", "7", "GO" to get a count of seven-tenths of a minute).

8) Set Up Display Output

The digits are placed in registers or memory locations with bit 4 set so that the output is sent to the displays. Bits 5 and 6 are set for the most significant digit to direct the output to the left display and to turn on the decimal point.

9) Pulse the LED Displays

Each display is turned on for two milliseconds. This process is repeated 1500 times in order to get a total delay of 0.1 minutes, or 6 seconds. The pulses are frequent enough so that the LED displays appear to be lit continuously.

10) **Decrement Display Count**

Flowchart:

```
                          ┌───────────┐
                          │   Start   │
                          └─────┬─────┘
                                ▼
                    ┌───────────────────────┐
                    │ Right Display = Right  │
                    │     Display - 1        │
                    └───────────┬───────────┘
                                ▼
                          ╱─────────╲
                         ╱   Is      ╲    No
                        ╱ Right Display╲──────────────┐
                        ╲     <0      ╱               │
                         ╲    ?      ╱                │
                          ╲─────────╱                 │
                                │ Yes                 │
                                ▼                     ▼
                    ┌───────────────────┐      ┌───────────┐
                    │  Left Display =   │      │    End    │
                    │  Left Display - 1 │      └───────────┘
                    └─────────┬─────────┘
                              ▼
                        ╱─────────╲
                       ╱   Is      ╲   Yes
                      ╱ Left Display ╲──────────────┐
                      ╲    < 0      ╱               │
                       ╲    ?      ╱                │
                        ╲─────────╱                 │
                              │ No                  │
                              ▼                     ▼
                    ┌───────────────────┐   ┌───────────────┐
                    │ Right Display = 9 │   │  End of timer │
                    │                   │   │    program    │
                    └─────────┬─────────┘   └───────────────┘
                              ▼
                        ┌───────────┐
                        │    End    │
                        └───────────┘
```
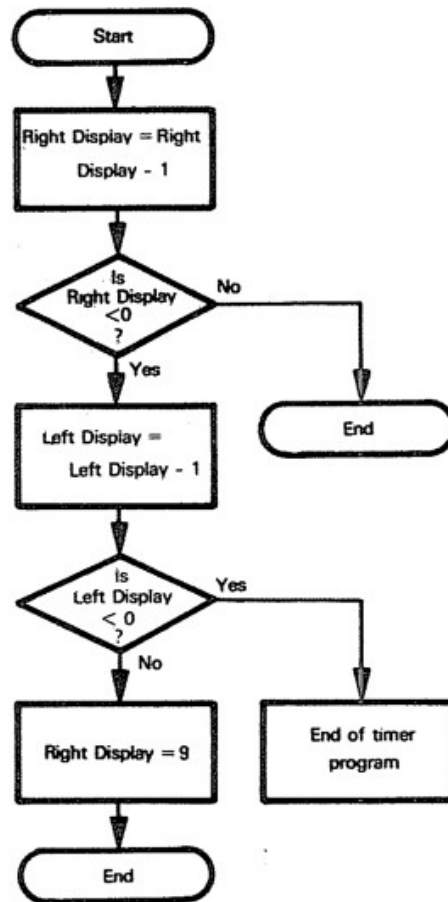
The value of the less significant digit is reduced by one. If this affects bit 4 (LEDON — used to turn the displays on), the digit has become negative. A borrow must then be obtained from the more significant digit. If the borrow from the more significant digit affects bit 4. the count has gone past zero and the countdown is finished. Otherwise. the program sets the value of the less significant digit to 9 and continues.

Note that comments describe both sections of the program and individual statements. The comments explain what the program is doing, not what specific instruction codes do. Spacing and indentation have been used to improve readability.

```
;PROGRAM NAME: TIMER
;DATE OF PROGRAM: 10/24/78
;PROGRAMMER: LANCE A. LEVENTHAL
;PROGRAM REQUIREMENTS: D1 (209) BYTES
;RAM REQUIREMENTS: 5 BYTES
;I/O REQUIREMENTS: 1 INPUT PORT. 1 OUTPUT PORT (1 Z80 PIO)
;
;
;THIS PROGRAM IS A SOFTWARE TIMER WHICH ACCEPTS INPUTS FROM A
;   CALCULATOR-LIKE KEYBOARD AND THEN PROVIDES A STOPWATCH
;   COUNTDOWN ON TWO 7-SEGMENT LED DISPLAYS IN MINUTES AND TENTHS
;   OF MINUTES
;
;KEYBOARD
;
;A I2-KEY KEYBOARD IS ASSUMED
;THREE COLUMN CONNECTIONS ARE OUTPUTS FROM THE PROCESSOR
;   SO THAT A COLUMN OF KEYS CAN BE GROUNDED
;FOUR ROW CONNECTIONS ARE INPUTS TO THE PROCESSOR SO THAT
;   COMPLETED CIRCUITS CAN BE IDENTIFIED
;THE KEYBOARD IS DEBOUNCED BY WAITING FOR TWO MILLISECONDS
;   AFTER A KEY CLOSURE IS RECOGNIZED
;A NEW KEY CLOSURE IS IDENTIFIED BY WAITING FOR THE OLD ONE
;   TO END SINCE NO STROBE IS USED
;THE KEYBOARD COLUMNS ARE CONNECTED TO BITS 0
;   TO 2 OF THE PIO B PORT
;THE KEYBOARD ROWS ARE CONNECTED TO BITS 0
;   TO 3 OF THE PIO A PORT
;
;DISPLAYS
;
;TWO 7-SEGMENT LED DISPLAYS ARE USED WITH SEPARATE DECODERS
;   (7447 OR 7448 DEPENDING ON THE TYPE OF DISPLAY)
;THE DECODER DATA INPUTS ARE CONNECTED TO BITS 0 TO 3
;   OF THE PIO B PORT
;BIT 4 OF THE PIO B PORT IS USED TO ACTIVATE THE LED
;   DISPLAYS (BIT 4 IS 1 TO SEND DATA TO LEDS)
;BIT 5 OF THE PIO B PORT IS USED TO SELECT WHICH
;   LED IS BEING USED (BIT 5 IS 1 IF THE LEADING DISPLAY
;   IS BEING USED, 0 IF THE TRAILING DISPLAY IS BEING USED)
;BIT 6 OF THE PIO B PORT IS USED TO LIGHT THE DECIMAL
;   POINT LED ON THE LEADING DISPLAY (BIT 6 IS 1 IF
;   THE DISPLAY IS TO BE LIT)
;
;METHOD
;
;STEP 1 - INITIALIZATION
;   THE MEMORY STACK POINTER (USED FOR SUBROUTINE RETURN
;   ADDRESSES) IS INITIALIZED. THE NUMBER OF DIGIT KEYS PRESSED IS SET
;   TO ZERO. AND THE ADDRESS INTO WHICH THE NEXT DIGIT KEY
;   IDENTIFICATION WILL BE PLACED IS INITIALIZED TO THE FIRST ADDRESS
;   IN THE DIGIT KEY ARRAY
;STEP 2 - LOOK FOR KEY CLOSURE
;   ALL KEYBOARD COLUMNS ARE GROUNDED AND THE KEYBOARD ROWS
;   ARE EXAMINED UNTIL A CLOSED CIRCUIT IS FOUND
```

```
;STEP 3 - DEBOUNCE KEY CLOSURE
;   A WAIT OF 2 MS IS INTRODUCED TO ELIMINATE KEY BOUNCE
;STEP 4 - IDENTIFY KEY CLOSURE
;   THE KEY CLOSURE IS IDENTIFIED BY GROUNDING SINGLE KEYBOARD
;   COLUMNS AND DETERMINING THE ROW AND COLUMN OF THE KEY
;   CLOSURE. A TABLE IS USED TO ENCODE THE KEYS ACCORDING TO THEIR
;   ROW AND COLUMN NUMBER
;   IN THE KEY TABLE, THE DIGITS ARE IDENTIFIED BY THEIR VALUES,
;   THE DECIMAL POINT KEY IS NO. 10, AND THE "GO" KEY IS NO. 11
;STEP 5 - SAVE KEY CLOSURE
;   DIGIT KEY CLOSURES ARE SAVED IN THE DIGIT KEY ARRAY UNTIL
;   TWO DIGITS HAVE BEEN IDENTIFIED. DECIMAL POINTS, FURTHER DIGITS,
;   AND CLOSURES OF THE "GO" KEY BEFORE TWO DIGITS HAVE BEEN
;   IDENTIFIED ARE IGNORED
;   AFTER TWO DIGITS HAVE BEEN FOUND, THE "GO" KEY IS USED TO
;   START THE COUNTDOWN PROCESS
;STEP 6 - COUNT DOWN TIMER INTERVAL ON LEDS
;   A COUNTDOWN IS PERFORMED ON THE LEDS WITH THE LEADING DIGIT
;   REPRESENTING THE REMAINING NUMBER OF MINUTES AND THE TRAILING
;   DIGIT REPRESENTING THE REMAINING NUMBER OF TENTHS OF MINUTES
;
;
;
;TIMER VARIABLE DEFINITIONS
;MEMORY SYSTEM CONSTANTS
;
BEGIN    EQU     50H              ;BEGIN IS STARTING MEMORY LOCATION
                                  ;   FOR PROG
LASTM    EQU     1000H            ;LASTM IS STARTING STACK ADDRESS
TEMP     EQU     800H             ;TEMP IS START OF RAM STORAGE
;RAM TEMPORARY STORAGE
         ORG     TEMP
KEYAD:   DEFS    2                ;KEYAD HOLDS THE ADDRESS IN THE
                                  ;   DIGIT KEY ARRAY IN WHICH THE
                                  ;   IDENTIFICATION OF THE NEXT DIGIT
                                  ;   KEY WILL BE PLACED
KEYNO:   DEFS    2                ;KEYNO IS THE DIGIT KEY ARRAY - IT
                                  ;   HOLDS THE IDENTIFICATIONS OF THE
                                  ;   DIGIT KEYS THAT HAVE BEEN PRESSED
NKEYS:   DEFS    1                ;NKEYS HOLDS NUMBER OF DIGIT KEYS
                                  ;   PRESSED
;
;I/O UNITS AND PIO ADDRESSES
;
PIODRA   EQU     0E0H             ;INPUT PIO FOR KEYBOARD
PIOCRA   EQU     0E2H
PIODRB   EQU     0E1H             ;OUTPUT PIO FOR KEYBOARD AND
                                  ;   DISPLAY
PIOCRB   EQU     0E3H
;
;DEFINITIONS
;
DECPT    EQU     6                ;BIT POSITION TO TURN ON DECIMAL
                                  ;   POINT LED
```

```
ECODE    EQU     0FFH            ;ERROR CODE IF ID ROUTINE DOES NOT FIND
                                 ;  KEY
GOKEY    EQU     11              ;IDENTIFICATION NUMBER FOR "GO" KEY
LEDON    EQU     4               ;BIT POSITION TO SEND OUTPUT TO LEDS
LEDSL    EQU     5               ;BIT POSITION TO SELECT LEADING
                                 ;  DISPLAY
MSCNT    EQU     0F9H            ;COUNT NEEDED TO GIVE 1 MS DELAY TIME
MXKEY    EQU     2               ;MAXIMUM NUMBER OF DIGIT KEY
                                 ;  CLOSURES USED
NROWS    EQU     4               ;NUMBER OF ROWS IN KEYBOARD OR KEYS
                                 ;  IN COLUMN
OPEN     EQU     00001111B       ;INPUT FROM KEYBOARD IF NO KEY
                                 ;  CLOSED
TPULS    EQU     2               ;NUMBER OF MS BETWEEN DIGIT DISPLAYS
TWAIT    EQU     2               ;NUMBER OF MS TO DEBOUNCE KEYS
;
;
;
         ORG     0
;
;RESET ROUTINE TO REACH TIMER PROGRAM
;
         JP      BEGIN           ;FIND TIMER PROGRAM
;
;INITIALIZATION OF TIMER PROGRAM
;
         ORG     BEGIN
         LD      A,01001111B     ;MAKE PIO PORT A INPUT
         OUT     (PIOCRA),A
         LD      A,00001111B     ;MAKE PIO PORT B OUTPUT
         OUT     (PIOCRB),A
         LD      SP,LASTM        ;PUT STACK AT END OF MEMORY
         SUB     A
         LD      (NKEYS),A       ;NUMBER OF DIGIT KEYS PRESSED = ZERO
         LD      HL,KEYNO        ;STARTING LOCATION FOR DIGIT KEYS
         LD      (KEYAD),HL
;
;SCAN KEYBOARD LOOKING FOR KEY CLOSURE
;
START:   CALL    SCANC           ;WAIT FOR KEY CLOSURE
;
;WAIT FOR KEY TO BE DEBOUNCED
;
         LD      A,TWAIT         ;GET DEBOUNCE TIME IN MS
         CALL    DELAY           ;WAIT FOR KEY TO STOP BOUNCING
;
;IDENTIFY WHICH KEY WAS PRESSED
;
         CALL    IDKEY           ;IDENTIFY KEY CLOSURE
         CP      ECODE           ;WAS KEY CLOSURE IDENTIFIED?
         JR      Z,START         ;NO. WAIT FOR ANOTHER CLOSURE
;
;ACT ON KEY IDENTIFICATION
```

16-11

```
            LD      B,A             ;SAVE KEY NUMBER
            LD      HL,NKEYS        ;CHECK FOR MAXIMUM NUMBER OF DIGIT
                                    ;    KEYS
            LD      A,(HL)
            CP      MXKEY           ;HAS MAXIMUM BEEN REACHED?
            JR      Z,KEYF          ;YES, LOOK FOR GO KEY
            LD      A,B             ;NO, LOOK FOR DIGIT KEYS ONLY
            CP      10              ;IS THIS KEY A DIGIT?
            JR      NC,WAITK        ;NO, IGNORE IT
            INC     (HL)            ;YES, INCREMENT DIGIT KEY COUNTER
            LD      HL,(KEYAD)      ;SAVE KEY NUMBER IN ARRAY
            LD      (HL),A
            INC     HL
            LD      (KEYAD),HL

;WAIT FOR CURRENT KEY CLOSURE TO END

WAITK:      CALL    SCANO           ;WAIT FOR KEY TO BE RELEASED
            JR      START           ;GO LOOK FOR NEXT KEY

;LOOK FOR GO KEY IF ENOUGH DIGITS FOUND

KEYF:       LD      A,B             ;GET NUMBER OF KEY PRESSED
            CP      GOKEY           ;IS IT "GO" KEY?
            JR      NZ,WAITK        ;NO, IGNORE IT

;PUT DIGITS INTO REGISTERS FOR DISPLAY

            LD      HL,KEYNO
            LD      D,(HL)          ;GET LEADING DIGIT
            SET     DECPT,D         ;TURN ON DECIMAL POINT
            SET     LEDON,D         ;SET OUTPUT TO LEDS
            SET     LEDSL,D         ;SELECT LEADING DISPLAY
            INC     HL
            LD      E,(HL)          ;GET TRAILING DIGIT
            SET     LEDON,E         ;SET OUTPUT TO LEDS

;PULSE THE LED DISPLAYS

            LD      C,PIODRB        ;GET OUTPUT PORT ADDRESS
LEDLP:      LD      H,6             ;SET COUNTERS FOR 6 SECONDS
TLOOP:      LD      B,250
LDPUL:      OUT     (C),D           ;OUTPUT LEADING DIGIT TO LED 1
            LD      A,TPULS         ;DELAY BETWEEN DIGITS
            CALL    DELAY
            OUT     (C),E           ;OUTPUT TRAILING DIGIT TO LED 2
            LD      A,TPULS         ;DELAY BETWEEN DIGITS
            CALL    DELAY
            DJNZ    LDPUL
            DEC     H
            JR      NZ,TLOOP

;DECREMENT COUNT ON LED DISPLAYS
```

16-12

```
        ;
                DEC     E               :COUNT DOWN TRAILING DIGIT
                BIT     LEDON,E         :IS TRAILING DIGIT PAST ZERO?
                JR      NZ,LEDLP        :NO, CONTINUE
                DEC     D               :COUNT DOWN LEADING DIGIT
                BIT     LEDON,D         :IS LEADING DIGIT PAST ZERO?
                JP      Z,BEGIN         :YES, WAIT FOR NEXT TIMING TASK
                LD      E,9             :NO, SET TRAILING DIGIT TO 9
                SET     LEDON,E         :SET OUTPUT TO LEDS
                JR      LEDLP           :RETURN TO DISPLAY SECTION
        ;
        :SUBROUTINE SCANC SCANS THE KEYBOARD WAITING FOR A KEY CLOSURE
        :ALL KEYBOARD INPUTS ARE GROUNDED
        ;
        SCANC:  SUB     A               :GROUND ALL KEYBOARD COLUMNS
                OUT     (PIODRB),A
                IN      A,(PIODRA)
                AND     OPEN            :IGNORE UNUSED INPUTS
                CP      OPEN            :ARE ANY KEYS CLOSED?
                JR      Z,SCANC         :NO, CONTINUE SCANNING
                RET
        ;
        :SUBROUTINE DELAY WAITS FOR THE NUMBER OF MILLISECONDS SPECIFIED
        :   IN REGISTER A
        ;
        DELAY:  EXX                     :SAVE USER REGISTERS
        DLY1:   LD      C,MSCNT         :LOAD REGISTER C FOR 1 MS
        WTLP:   DEC     C               :WAIT 1 MS
                JR      NZ,WTLP
                DEC     A               :COUNT DOWN NUMBER OF MS
                JR      NZ,DLY1
                EXX                     :RESTORE USER REGISTERS
                RET
        ;
        :SUBROUTINE IDKEY DETERMINES THE ROW AND COLUMN NUMBER OF THE
        :   KEY CLOSURE AND IDENTIFIES THE KEY BY USING A TABLE
        ;
        IDKEY:  LD      BC,PATT         :POINT TO SCAN PATTERNS
                LD      HL,KTAB-1       :START KEY TABLE POINTER
                LD      DE,NROWS        :GET NUMBER OF KEYS IN A COLUMN
        ;
        :SCAN KEYBOARD COLUMNS SUCCESSIVELY LOOKING FOR CLOSURE
        ;
        FCOL:   LD      A,(BC)          :GET PATTERN TO GROUND COLUMN
                CP      ECODE           :ALL COLUMNS SCANNED?
                RET     Z               :YES, RETURN WITH ERROR CODE
                OUT     (PIODRB),A      :SCAN COLUMN
                IN      A,(PIODRA)
                AND     OPEN            :IGNORE UNUSED INPUTS
                CP      OPEN            :ANY KEYS IN THIS COLUMN CLOSED?
                JR      NZ,FROW         :YES, GO DETERMINE CLOSURE ROW
                ADD     HL,DE           :NO, MOVE KEY TABLE POINTER TO
                                        :   NEXT COLUMN
                INC     BC              :POINT TO NEXT SCAN PATTERN
                JR      FCOL
```

```
;DETERMINE ROW NUMBER OF CLOSURE
;
FROW:   INC     HL              ;MOVE KEY TABLE POINTER TO NEXT ROW
        RRCA                    ;NEXT ROW GROUNDED?
        JR      C,FROW          ;NO, KEEP LOOKING
;
;IDENTIFY KEY FROM TABLE
;
        LD      A,(HL)          ;GET KEY NUMBER
        RET
;
;SCAN PATTERNS USED TO GROUND ONE COLUMN AT A TIME
;ERROR PATTERN USED TO INDICATE THAT ALL COLUMNS HAVE BEEN SCANNED
;THE COLUMN ATTACHED TO OUTPUT BIT 0 IS SCANNED FIRST, THEN
.   THE ONE ATTACHED TO OUTPUT BIT 1. ETC.
;
PATT:   DEFB    00000110B
        DEFB    00000101B
        DEFB    00000011B
        DEFB    ECODE
;
;KEYBOARD TABLE
;
;COLUMNS ARE PRIMARY INDEX, ROWS SECONDARY INDEX
;THE KEYS IN THE COLUMN ATTACHED TO OUTPUT BIT 0 ARE FOLLOWED
.   BY THOSE IN THE COLUMN ATTACHED TO OUTPUT BIT 1. ETC. WITHIN
.   A COLUMN. THE KEY ATTACHED TO INPUT BIT 0 IS FIRST FOLLOWED
:   BY THE ONE ATTACHED TO INPUT BIT 1. ETC.
;THE DIGIT KEYS ARE 0 TO 9, DECIMAL POINT IS 10. GO IS 11
;
KTAB:   DEFB    3               ;C0,R0
        DEFB    2               ;C0,R1
        DEFB    0               ;C0,R2
        DEFB    4               ;C0,R3
        DEFB    8               ;C1,R0
        DEFB    9               ;C1,R1
        DEFB    1               ;C1,R2
        DEFB    11              ;C1,R3
        DEFB    5               ;C2,R0
        DEFB    6               ;C2,R1
        DEFB    7               ;C2,R2
        DEFB    10              ;C2,R3
;
;SUBROUTINE SCANO SCANS THE KEYBOARD WAITING FOR KEY CLOSURE TO
,   END SO NEXT CLOSURE CAN BE FOUND
;
SCANO:  SUB     A               ;GROUND ALL KEYBOARD COLUMNS
        OUT     (PIODRB),A
        IN      A,(PIODRA)
        AND     OPEN            ;IGNORE UNUSED INPUTS
        CP      OPEN            ;ARE ANY KEYS STILL CLOSED?
        JR      NZ,SCANO        ;YES, CONTINUE SCANNING
        RET
        END
```

PROJECT #2: A Digital Thermometer

Purpose: This project is a digital thermometer which shows the temperature in degrees Celsius on two seven-segment displays.

Hardware: The project uses one input port and one output port, two seven-segment displays, a 74LS04 inverter, a 74LS00 NAND gate or a 74LS08 AND gate depending on the polarity of the displays, an Analog Devices AD7570J 8-bit monolithic A/D converter, an LM311 comparator, and various peripheral drivers, resistors, and capacitors as required by the displays and the converter. (See Chapter 11 and Reference 1 at the end of this chapter for discussions of A/D converters.)

Figure 16-2 shows the organization of the hardware. Output line 7 from PIO Port B is used to send a Start Conversion signal to the A/D converter. Input lines 0 through 7 are attached directly to the eight digital data lines from the converter. Output lines 0 through 3 are used to send BCD digits to the seven-segment decoder/drivers. Output line 4 activates the displays and output line 5 selects the left or right display (line 5 is '1' for the left display).

The analog part of the hardware is shown in Figure 16-3. The thermistor simply provides a resistance that depends on temperature. Figure 16-4 is a plot of the resistance and Figure 16-5 shows the range of current values over which the resistance is

THERMOMETER ANALOG HARDWARE

linear. The conversion to degrees Celsius in the program is performed with a calibration table. The two potentiometers can be adjusted to scale the data properly. A clock for the A/D converter is generated from an RC network. The values are R7=33 kΩ and C1=1000 pF, so that the clock frequency is about 75 kHz. At this frequency, the maximum conversion time for eight bits is about 50 microseconds. A much longer delay is allowed for conversion so that no check for the end of conversion is necessary. The 8-bit version of the converter requires the following special connections. The eight data lines are DB2 through DB9 (DB1 is always high during conversion and DB0 low). The Short Cycle 8-bit input (pin 26-$\overline{SC8}$) is tied low so that only an 8-bit conversion is performed. In the present case, High Byte Enable (pin 20-HBEN) and Low Byte Enable (pin 21-LBEN) were both tied high so that the data outputs were always enabled.

The A/D converter uses the successive approximation method to perform a conversion. The ADC's data register is connected to the inputs of an internal D/A converter whose output (available at OUT1 and OUT2) is compared to the analog input. When a conversion is initiated, the ADC logic sets the data register to all zeros with the exception of the most significant bit (MSB), which is set to one. If the analog input is less than the resulting internally generated analog value, then the MSB is reset to zero; otherwise it remains a one. The next most significant bit is then set to one and the process repeated until all eight bits have been "tested" in this way. After the eighth cycle, the value in the register is the value which most closely corresponds to the analog input.

This method is fast, but it requires that the input be stable during the conversion process. Rapidly changing or noisy inputs would require additional signal conditioning. The references at the end of this chapter describe more accurate methods for handling analog I/O.
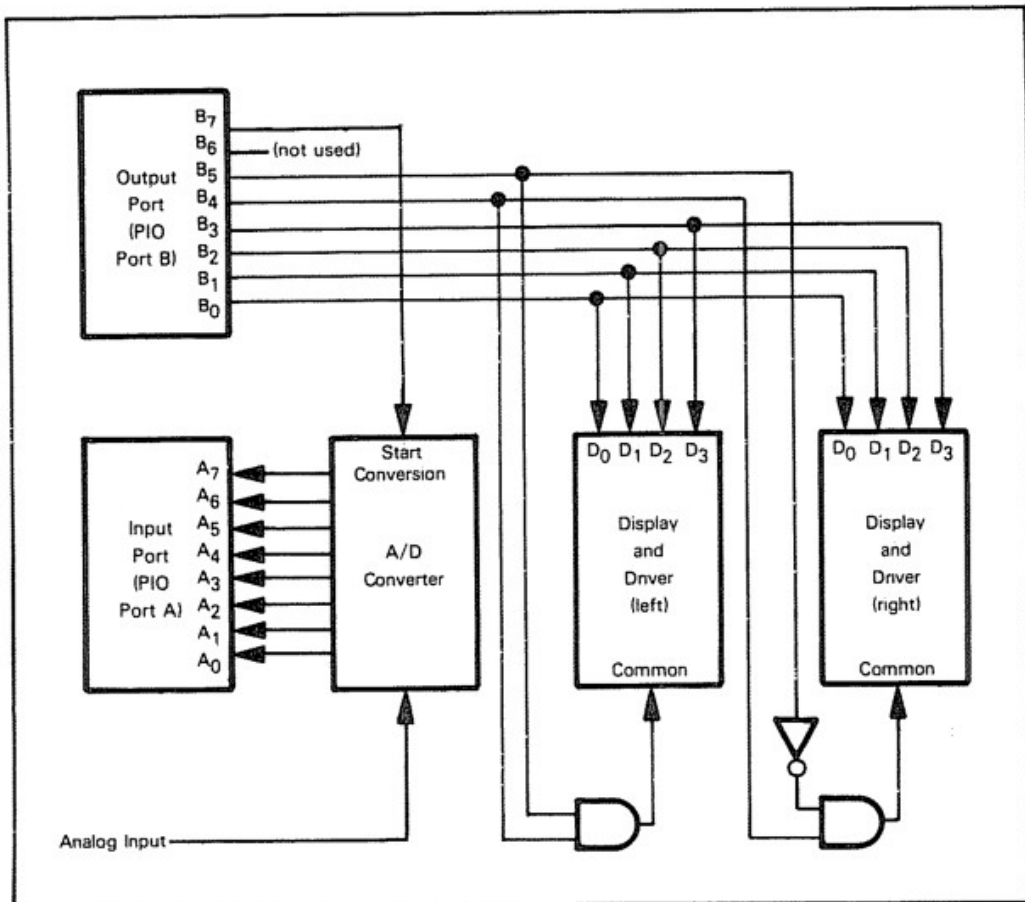
Figure 16-2. I/O Configuration for a Digital Thermometer

R6
50 kΩ
OFFSET ADJ
-15 V   +15 V

R5
2 MΩ

R3
200 Ω
V_{REF}
-10 V
V_{REF}
+15V
+5 V

22
V_{CC}
V_{DD}
1

OUT1
4
R1
1 kΩ
2 +
8
LM311
7
R2
5 kΩ

+15 V

OUT2
5
3 −
1
4
D

A

-15 V

+5 V
R7
33 kΩ

24
CLOCK
COMP
7

AD7570J
A/D
Converter

SRO
8
(not used)

SYNC
9
(not used)

+15 V
C1
1000 pF

R_T
Thermistor

A
3
ANALOG
INPUT

R4   1 kΩ
Gain Adjust

R8
68 kΩ

+5V

A

DB9
10
A7
DB8
11
A6
DB7
12
A5
DB6
13
A4
DB5
14
A3
DB4
15
A2
DB3
16
A1
DB2
17
A0

To PIO
Port A

20
HBEN
21
LBEN

From PIO Port B, bit 7
25
STRT

28
$\overline{BUSY}$
(not used)

27
BSEN

26
$\overline{SC8}$

AGND
DGND
6
23

D
A
D

DB1
18
(not used)
DB0
19
(not used)

Note: If positive V_{REF} is used, the ANALOG INPUT range is 0 to -V_{REF}, and the COMPARATOR's (-) input should be connected to OUT1 (pin 4) of the AD7570.

R_T is the thermistor. The analog input from the voltage divider is:

$$\frac{R_8}{R_8 + R_T} \times 15 \text{ Volt}$$

Since R_F = 68 kΩ, the input is: $\dfrac{1.02 \text{ M}\Omega}{R_T + 68 \text{ k}\Omega}$ Volt

R_T has a minimum value of 34 kΩ (T=**50°C**, see Figure 16-4) so full scale is 10 Volt.

Figure 16-3. Digital Thermometer Analog Hardware
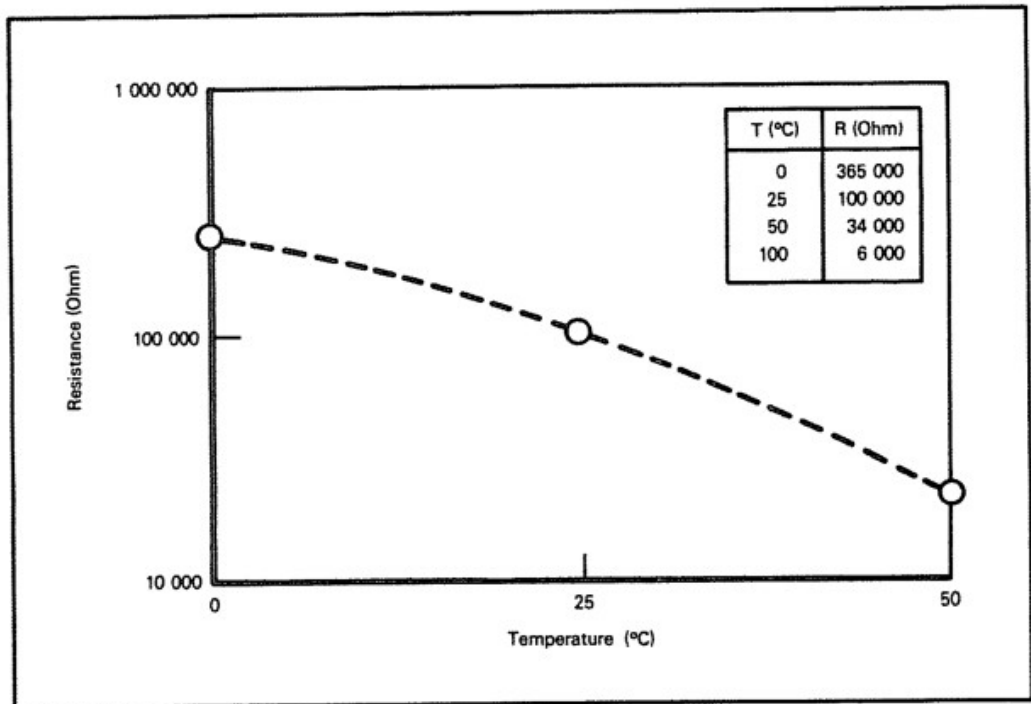
Figure 16-4. Thermistor Characteristics
(Fenwal GA51J1 Bead)

T (°C)	R (Ohm)
0	365 000
25	100 000
50	34 000
100	6 000

The curve is linear (i.e., the resistance is independent of current) for currents less than 0.1 milliampere.
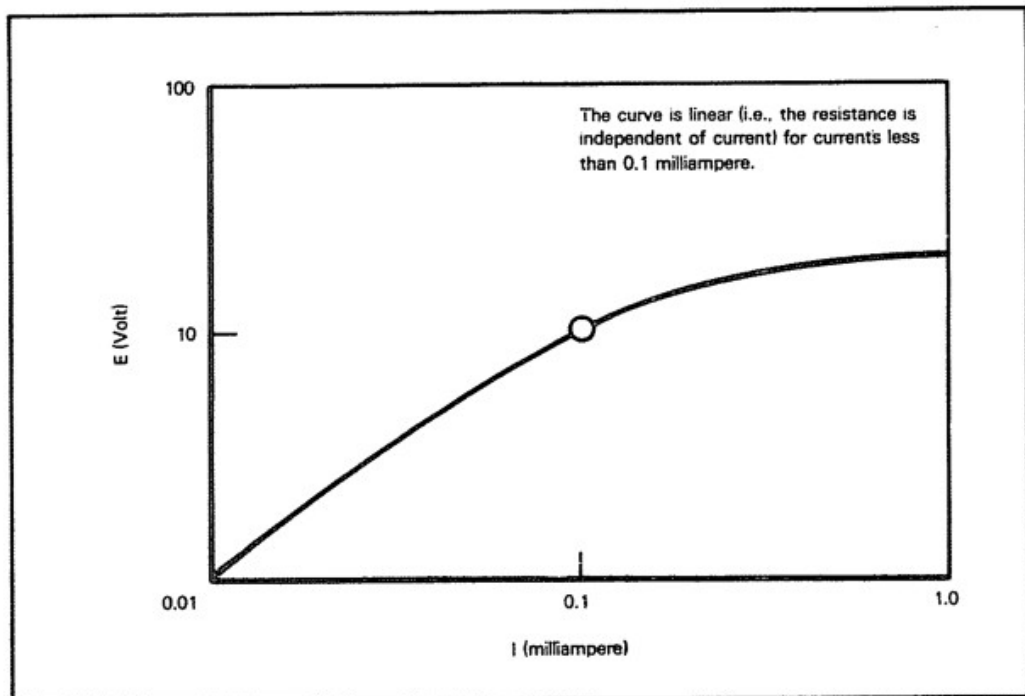
Figure 16-5. Typical E-I Curve for Thermistor (25°C)

General Program Flowchart·

```
        ( Start )
            |
            v
    +----------------+
    | Initialization |
    +----------------+
            |
     +----->|
     |      v
     |  +----------------+
     |  |  Send Start    |
     |  | Conversion signal
     |  | to A/D converter|
     |  +----------------+
     |      |
     |      v
     |  +----------------+
     |  |   Wait 1 ms    |
     |  +----------------+
     |      |
     |      v
     |  +----------------+
     |  | Read data from |
     |  |  A/D converter |
     |  +----------------+
     |      |
     |      v
     |  +----------------+
     |  | Convert data to|
     |  | degrees Celsius|
     |  +----------------+
     |      |
     |      v
     |  +----------------+
     |  |    Display     |
     |  | temperature on |
     |  |LEDs for six seconds|
     |  +----------------+
     |      |
     +------+
```
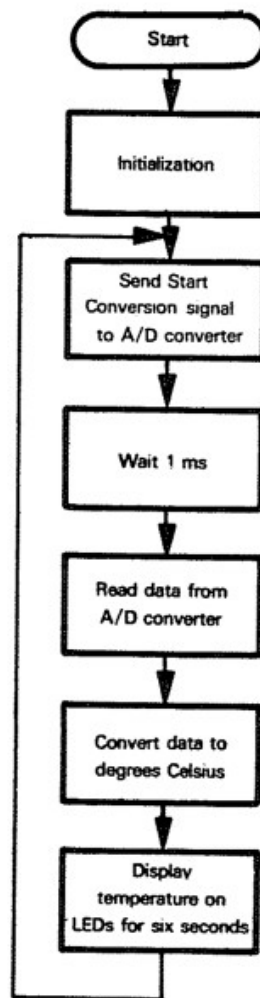
Program Description:

1) **Initialization**

   Location 0 (the Z80 microprocessor RESET location) contains a jump to the starting address of the main program. The initialization configures the PIO control registers and starts the Stack Pointer at the highest address in RAM. The Stack is used only to store subroutine return addresses.

2) **Send START CONVERSION Signal to A/D Converter**

   The CPU pulses the START CONVERSION line by first placing a '1' on line 7 of PIO Port B and then placing a '0' on that line. Each input from the converter requires a starting pulse.

3) **Wait 1 ms for Conversion**

   A delay of 1 ms after the START CONVERSION pulse guarantees a completed conversion. Actually, the converter takes only a maximum of 100 microseconds for an 8-bit conversion. We could reduce the delay by checking the $\overline{BUSY}$ signal from the converter. This signal is either a '1' (conversion complete) or '0' (conversion in progress) if the BUSY ENABLE line is addressed. In the present case there is no reason to speed the conversion process. Clearly, interrupts could be used with $\overline{BUSY}$ tied to the PIO STROBE line.

4) **Read Data from A/D Converter**

   Reading the data involves a single input operation. We should note that the Analog Devices AD7570J has an Enable input and tristate outputs so that it could be tied directly to the microprocessor Data Bus.

   The 7570 converter is, of course, underutilized in this particular application, particularly since we are interfacing it to the Z80 processor through a PIO. A simpler 8-bit A/D converter such as the National 5357 device would do the job at lower cost; this device is available in an 18-pin package, has a START CONVERSION input, and provides tristate outputs. It also has output latches and an END OF CONVERSION output signal.

5) **Convert Data to Degrees Celsius**
   Flowchart:
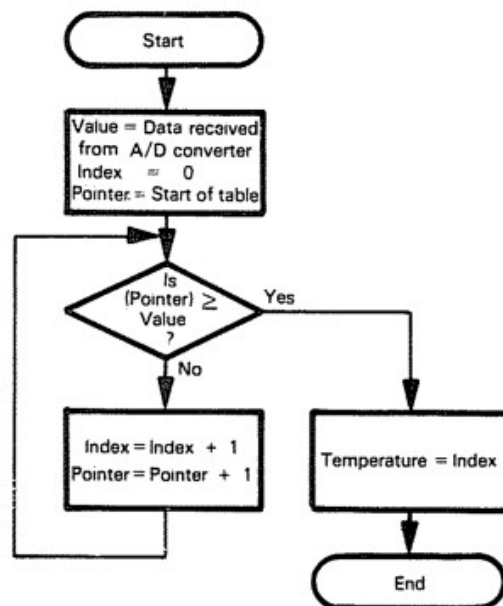
The conversion uses a table that contains the largest input value corresponding to a given temperature. The program searches the table, looking for a value greater than or equal to the value received from the converter. The first

USING A CALIBRATION TABLE

such value it finds corresponds to the required temperature; that is, if the tenth entry is the first value larger than or equal to the data, the temperature is 10 degrees. This search method is inefficient but adequate for the present application.

Note that we must keep the entry number in decimal rather than binary. The instruction sequence "ADD A.1: DAA" keeps the index as two decimal digits instead of a binary number. For example, the entry number after 9 (00001001 binary) will be decimal 10 (00010000 BCD) rather than binary ten (00001010). The reason for this is that we plan to display the temperature as two decimal digits and would have to convert it from binary to decimal otherwise.

The table could be obtained by calibration or by a mathematical approximation. The calibration method is simple, since the thermometer must be calibrated anyway. The table occupies one memory location for each temperature value to be displayed.[1]

To calibrate the thermometer, you must first adjust the potentiometers to produce the proper overall range and then determine the converter output values corresponding to specific temperatures.

6) **Prepare Data for Display**
   Flowchart:

```
                    ┌─────────────┐
                    │    Start    │
                    └──────┬──────┘
                           ▼
                 ┌───────────────────┐
                 │ Get least significant
                 │   digit and set
                 │   output to LEDs
                 └─────────┬─────────┘
                           ▼
                 ┌───────────────────┐
                 │    Get most       │
                 │  significant digit│
                 └─────────┬─────────┘
                           ▼
                        ╱     ╲
                      ╱   Is    ╲   Yes
                    ╱    most     ╲──────┐
                    ╲ significant digit ╱ │
                      ╲    0 ?    ╱       │
                        ╲     ╱          │
                          │ No           │
                          ▼              │
                 ┌───────────────────┐   │
                 │ Set output to LEDs│   │
                 └─────────┬─────────┘   │
                           ▼◄────────────┘
                    ┌─────────────┐
                    │     End     │
                    └─────────────┘
```
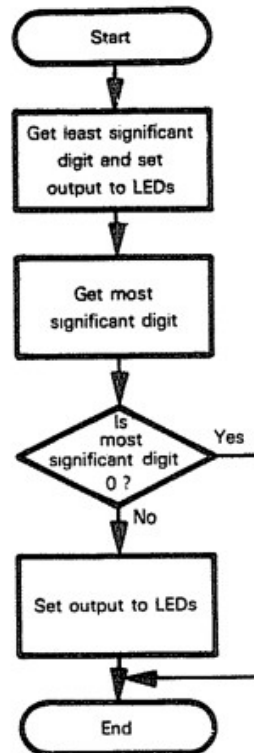
The least significant digit is masked off. We set the bit that turns on the displays. The result is saved in Register E.

The only difference for the most significant digit is that a leading zero is blanked (i.e., the displays show "blank 7" rather than "07" for 7°C). This simply involves not setting the bit that turns on the displays if the digit is zero. The result is saved in Register D.

> **BLANKING A LEADING ZERO**

7) **Display Temperature for Six Seconds**
   Flowchart:

```
                    ┌─────────────┐
                   (    Start      )
                    └──────┬──────┘
                           ▼
                    ┌─────────────┐
                    │ Count = TSAMP│
                    └──────┬──────┘
         ┌─────────────────▼
         │          ┌─────────────┐
         │          │  Send most   │
         │          │significant digit│
         │          │ to left display│
         │          └──────┬──────┘
         │                 ▼
         │          ┌─────────────┐
         │          │  Wait 2 ms   │
         │          └──────┬──────┘
         │                 ▼
         │          ┌─────────────┐
         │          │  Send least  │
         │          │significant digit│
         │          │ to right display│
         │          └──────┬──────┘
         │                 ▼
         │          ┌─────────────┐
         │          │  Wait 2 ms   │
         │          └──────┬──────┘
         │                 ▼
         │          ┌─────────────┐
         │          │Count = Count - 1│
         │          └──────┬──────┘
         │                 ▼
         │              ◇ Is  ◇
         └─── No ──────◇ Count 0 ◇
                        ◇   ?   ◇
                           │
                           ▼ Yes
                    ┌─────────────┐
                   (     End       )
                    └─────────────┘
```
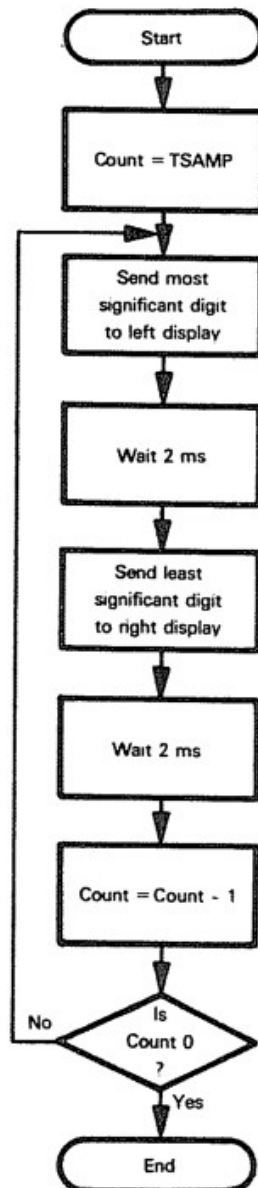
Each display is pulsed often enougn so that it appears to be lit continuously. If TPULS were made longer (say 50 ms), the displays would appear to flash on and off.

The program uses a 16-bit counter to count the time between temperature samples. The Z80 has instructions to increment or decrement 16-bit register pairs or index registers. However, these instructions do not affect the flags, so there is no way to directly determine when the counter reaches zero. So we make this determination by logically ORing the eight most significant and the eight least significant bits of the counter. If that result is zero, the 16-bit counter is zero.

16-23

```
;
;PROGRAM NAME: THERMOMETER
;DATE OF PROGRAM: 10/20/78
;PROGRAMMER: LANCE A. LEVENTHAL
;PROGRAM MEMORY REQUIREMENTS: 154 BYTES
;RAM REQUIREMENTS: NONE
;I/O REQUIREMENTS: 1 INPUT PORT, 1 OUTPUT PORT (1 Z80 PIO)
;
;
;THIS PROGRAM IS A DIGITAL THERMOMETER THAT ACCEPTS INPUTS FROM
;   AN A/D CONVERTER ATTACHED TO A THERMISTOR. CONVERTS THE INPUT
;   TO DEGREES CELSIUS. AND DISPLAYS THE RESULTS ON TWO
;   SEVEN-SEGMENT LED DISPLAYS
;
;A/D CONVERTER
;
;THE A/D CONVERTER IS AN ANALOG DEVICES 7570J MONOLITHIC CONVERTER
;   WHICH PRODUCES AN 8-BIT OUTPUT
;THE CONVERSION PROCESS IS STARTED BY A PULSE ON THE START
;   CONVERSION LINE (BIT 7 OF PIO PORT B)
;THE CONVERSION IS COMPLETED IN 50 MICROSECONDS AND THE
;   DIGITAL DATA IS LATCHED
;
;DISPLAYS
;
;TWO SEVEN-SEGMENT LED DISPLAYS ARE USED WITH SEPARATE DECODERS
;   (7447 OR 7448 DEPENDING ON THE TYPE OF DISPLAY)
;THE DECODER DATA INPUTS ARE CONNECTED TO BITS 0 TO 3 OF
;   PIO PORT B
;BIT 4 OF PIO PORT B IS USED TO ACTIVATE THE LED DISPLAYS
;   (BIT 4 IS 1 TO SEND DATA TO LEDS)
;BIT 5 OF PIO PORT B IS USED TO SELECT WHICH LED IS BEING
;   USED (BIT 5 IS 1 IF THE LEADING DISPLAY IS BEING USED.
;   0 IF THE TRAILING DISPLAY IS BEING USED)
;
;METHOD
;
;STEP 1 - INITIALIZATION
;   THE MEMORY STACK (USED FOR SUBROUTINE RETURN ADDRESSES) IS
;   INITIALIZED
;STEP 2 - PULSE START CONVERSION LINE
;   THE A/D CONVERTER'S START CONVERSION LINE (BIT 7 OF PIO
;   PORT B) IS PULSED
;STEP 3 - WAIT FOR A/D OUTPUT TO SETTLE
;   A WAIT OF 1 MS ALLOWS FOR COMPLETION OF THE CONVERSION
;STEP 4 - READ A/D VALUE. CONVERT TO DEGREES CELSIUS.
;   A TABLE IS USED FOR CONVERSION IT CONTAINS THE MAXIMUM
;   INPUT VALUE FOR EACH TEMPERATURE READING
;STEP 5 - DISPLAY TEMPERATURE ON LEDS
;   THE TEMPERATURE IS DISPLAYED ON THE LEDS FOR SIX SECONDS
;   BEFORE ANOTHER CONVERSION IS PERFORMED
;
;
;
;THERMOMETER VARIABLE DEFINITIONS
;
;MEMORY SYSTEM CONSTANTS
```

```
;
BEGIN    EQU    50H              ;STARTING ADDRESS OF MAIN PROGRAM
LASTM    EQU    1000H            ;STARTING ADDRESS FOR RAM STACK
;
;I/O UNITS AND PIO ADDRESSES
;
PIODRA   EQU    0E0H             ;INPUT PIO FOR CONVERTER
PIOCRA   EQU    0E2H
PIODRB   EQU    0E1H             ;OUTPUT PIO FOR DISPLAYS
PIOCRB   EQU    0E3H
;
;DEFINITIONS
;
LEDON    EQU    4                ;BIT POSITION TO SEND DATA TO LEDS
LEDSL    EQU    5                ;BIT POSITION TO SELECT LEADING DISPLAY
MSCNT    EQU    0F9H             ;COUNT NEEDED TO GIVE 1 MS DELAY
STCON    EQU    10000000B        ;OUTPUT TO BRING START CONVERSION HIGH
TPULS    EQU    2                ;DISPLAY PULSE LENGTH IN MS
TSAMP    EQU    1500             ;TSAMP IS THE NUMBER OF TIMES THE
                                 ;DISPLAYS ARE PULSED IN A
                                 ;TEMPERATURE SAMPLING PERIOD. THE
                                 ;LENGTH OF A SAMPLING PERIOD IS THUS
                                 ;2*TPULS*TSAMP MILLISECONDS.THE FACTOR
                                 ;OF 2*TPULS IS INTRODUCED BY THE FACT
                                 ;THAT EACH OF 2 DISPLAYS IS PULSED FOR
                                 ;TPULS MS


;
;
;
         ORG    0
;
;RESET ROUTINE TO REACH THERMOMETER PROGRAM
;
         JP     BEGIN            ;FIND THERMOMETER PROGRAM
;
;INITIALIZATION OF THERMOMETER PROGRAM
;
         ORG    BEGIN
         LD     A,01001111B      ;MAKE PIO PORT A INPUT
         OUT    (PIOCRA),A
         LD     A,00001111B      ;MAKE PIO PORT B OUTPUT
         OUT    (PIOCRB),A
         LD     SP,LASTM         ;PUT STACK AT END OF RAM
;
;PULSE START CONVERSION LINE
;
START:   LD     A,STCON
         OUT    (PIODRB),A       ;SEND START CONVERSION HIGH
         SUB    A
         OUT    (PIODRB),A       ;SEND START CONVERSION LOW
```

```
;
;DELAY 1 MS FOR CONVERSION
;
            LD      A.1             ;CONVERSION DELAY TIME IN MS
            CALL    DELAY           ;WAIT FOR CONVERSION

;READ DIGITAL DATA FROM CONVERTER
;
            IN      A,(PIODRA)      ;GET DATA FROM A/D CONVERTER

;CONVERT A/D DATA TO 2 BCD DIGITS
;
            CALL    CONVR           ;CONVERT DATA TO BCD

;GET LEAST SIGNIFICANT DIGIT
;
            LD      B.A             ;SAVE BCD DIGITS
            AND     0FH             ;MASK OFF LSD
            SET     LEDON.A         ;SET OUTPUT TO LEDS
            LD      E,A             ;SAVE LSD IN REGISTER E

;GET MOST SIGNIFICANT DIGIT, BLANK LEADING ZERO
;
            LD      A,B             ;RESTORE BCD DIGITS
            RRCA                    ;SHIFT MSD
            RRCA
            RRCA
            RRCA
            AND     0FH             ;MASK OFF MSD
            JR      Z.SVMSD         ;DON'T TURN DISPLAY ON IF VALUE ZERO
            SET     LEDON.A         ;SET OUTPUT TO LEDS
            SET     LEDSL.A         ;SELECT LEADING DISPLAY
SVMSD:  LD      D.A             ;SAVE MSD IN REGISTER D
;
;PULSE THE LED DISPLAYS
;
            LD      C.PIODRB        ;GET OUTPUT PORT ADDRESS
            LD      HL.TSAMP        ;GET l6-BIT PULSE COUNTER
DSPLY:  OUT     (C).D           ;OUTPUT LEADING DIGIT TO DISPLAY
            LD      A.TPULS         ;DELAY DISPLAY PULSE LENGTH
            CALL    DELAY
            OUT     (C).E           ;OUTPUT TRAILING DIGIT TO DISPLAY
            LD      A.TPULS         ;DELAY DISPLAY PULSE LENGTH
            CALL    DELAY
            DEC     HL              ;COUNT DOWN 16-BIT COUNTER
            LD      A.H             ;REMEMBER DEC HL DOES NOT SET Z FLAG
            OR      L
            JR      NZ.DSPLY        ;CONTINUE PULSING DISPLAYS
            JP      START           ;GO SAMPLE TEMPERATURE AGAIN
;
;SUBROUTINE DELAY WAITS FOR THE NUMBER OF MILLISECONDS SPECIFIED
;   IN REGISTER A
```

```
;
DELAY:  EXX                         :SAVE USER REGISTERS
DLY1:   LD      C.MSCNT             :LOAD REGISTER C FOR 1 MS DELAY
WTLP:   DEC     C                   :WAIT 1 MS
        JR      NZ.WTLP
        DEC     A                   :COUNT DOWN NUMBER OF MS
        JR      NZ.DLY1
        EXX                         :RESTORE USER REGISTERS
        RET

;
;SUBROUTINE CONVR CONVERTS INPUT FROM A/D CONVERTER TO DEGREES
:   CELSIUS BY USING A TABLE. INPUT DATA IS IN THE ACCUMULATOR.
:   RESULT IS 2 BCD DIGITS IN THE ACCUMULATOR
;
;REGISTERS USED: A.B.C.H.L
;
;
CONVR:  LD      HL.DEGTB            :GET BASE ADDRESS OF CONVERSION
                                    :   TABLE
        LD      B.A                 :SAVE A/D INPUT
        LD      C.0                 :START DEGREES AT ZERO
CHVAL:  LD      A.(HL)              :GET ENTRY FROM TABLE
        CP      B                   :IS A/D INPUT BELOW ENTRY?
        LD      A.C                 :GET VALUE IN DEGREES CELSIUS
        RET     NC                  :YES. VALUE FOUND
        ADD     A.1                 :NO, ADD 1 TO DEGREES
        DAA                         :KEEP DEGREES IN BCD
        LD      C.A
        INC     HL
        JR      CHVAL
;
;TABLE DEGTB WAS OBTAINED BY CALIBRATION WITH A KNOWN REFERENCE
;DEGTB CONTAINS THE LARGEST INPUT VALUE THAT CORRESPONDS TO A
:   PARTICULAR TEMPERATURE READING (I.E.. THE FIRST ENTRY IS DECIMAL
:   58 SO AN INPUT VALUE OF 58 IS THE LARGEST VALUE GIVING A ZERO
:   TEMPERATURE READING - VALUES BELOW ZERO ARE DISPLAYED AS ZERO
```

```
DEGTB:  DEFB  58
        DEFB  61
        DEFB  63
        DEFB  66
        DEFB  69
        DEFB  71
        DEFB  74
        DEFB  77
        DEFB  80
        DEFB  84
        DEFB  87
        DEFB  90
        DEFB  93
        DEFB  97
        DEFB 101
        DEFB 104
        DEFB 108
        DEFB 112
        DEFB 116
        DEFB 120
        DEFB 124
        DEFB 128
        DEFB 132
        DEFB 136
        DEFB 141
        DEFB 145
        DEFB 149
        DEFB 154
        DEFB 158
        DEFB 163
        DEFB 167
        DEFB 172
        DEFB 177
        DEFB 181
        DEFB 186
        DEFB 191
        DEFB 195
        DEFB 200
        DEFB 204
        DEFB 209
        DEFB 214
        DEFB 218
        DEFB 223
        DEFB 227
        DEFB 232
        DEFB 236
        DEFB 241
        DEFB 245
        DEFB 249
        DEFB 253
        DEFB 255
        END
```

References

1. A method that uses far less memory is described in T. A. Seim, "Numerical Interpolation for Microprocessor-based Systems," Computer Design, February 1978, pp. 111-116.

   See also:

2. Auslander, D. M. et al., "Direct Digital Process Control: Practice and Algorithms for Microprocessor Applications," Proceedings of the IEEE, February 1978, pp. 199-208

3. Bernstein, N., "What to Look for in Analog Input/Output Boards," Electronics, January 19, 1978, pp. 13-119

4. Bibbero, R. J., Microprocessors in Instruments and Control, Wiley, New York, 1977

5. Burton, D. P. and A. L. Dexter, Microprocessor Systems Handbook, Analog Devices, Inc., P.O. Box 796, Norwood, MA. 02062, 1977

6. Finkel, J., Computer-Aided Experimentation, Wiley, New York, 1975

7. Garrett, P. H., Analog Systems for Microprocessors and Minicomputers, Reston Publishing Co., Reston, VA., 1978

8. Hnatek, E. R., A User's Handbook of D/A and A/D Converters, Wiley, New York, 1976

9. Mrozowski, A., "Analog Output Chips Shrink A-D Conversion Software," Electronics, June 23, 1977, pp. 130-133

10. The Optoelectronics Data Book, Texas Instruments, Inc., P.O. Box 5012, Dallas, TX., 1978

11. The Optoelectronic Designer's Catalog, Hewlett-Packard Inc., 1820 Embarcadero Road, Palo Alto, CA. 94303, 1978

12. Peatman, J. B., Microcomputer-based Design, McGraw-Hill, New York, 1977

13. Rony, P. R. et al., "Microcomputer Interfacing: Sample and Hold Devices," Computer Design, December 1977, pp. 106-108

14. Sheingold, D. H. ed., Analog-Digital Conversion Notes, Analog Devices, Inc., P. O. Box 796, Norwood, MA. 02062, 1977

Index of Instruction Descriptions

Index of Instruction Descriptions (Continued)

Index

Index (Continued)

Index (Continued)

Index (Continued)

About the Author

Lance A. Leventhal is a partner in Emulative Systems Company, a San Diego-based consulting firm specializing in microprocessors and microprogramming. He serves as Technical Editor of the Society for Computer Simulation and as a Contributing Editor for Digital Design. He is a national lecturer on microprocessors for the IEEE, the author of five books and over forty articles on microprocessors, and a regular contributor to such publications as Simulation, Digital Design, and Kilobaud.

Dr. Leventhal's previous experience includes affiliations with Linkabit Corporation, Intelcom Rad Tech, Naval Electronics Laboratory Center and Harry Diamond Laboratories. He received a B.A. degree from Washington University in St. Louis, Missouri, and M.S. and Ph.D. degrees from the University of California at San Diego. He is a member of SCS, ACM, and IEEE.

OSBORNE/McGraw-Hill GENERAL BOOKS

An Introduction to Microcomputers series
by Adam Osborne
Volume 0 — The Beginner's Book
Volume 1 — Basic Concepts
Volume 2 — Some Real Microprocessors (1978 ed.)
Volume 3 — Some Real Support Devices (1978 ed.)
Volume 2 1978-1979 Update Series
Volume 3 1978-1979 Update Series

The 8089 I/O Processor Handbook
by Adam Osborne

The 8086 Book
by R. Rector and G. Alexy
8080 Programming for Logic Design
by Adam Osborne
6800 Programming for Logic Design
by Adam Osborne
Z80 Programming for Logic Design
by Adam Osborne

8080A/8085 Assembly Language Programming
by L. Leventhal
6800 Assembly Language Programming
by L. Leventhal
6502 Assembly Language Programming
by L. Leventhal
Z8000 Assembly Language Programming
by L. Leventhal et al.
Running Wild: The Next Industrial Revolution
by Adam Osborne
PET-CBM Personal Computer Guide
by Carroll Donahue and Janice Enger
PET and the IEEE 488 Bus (GPIB)
by E. Fisher and C. W. Jensen

OSBORNE/McGraw-Hill SOFTWARE

Practical Basic Programs
by L. Poole et al.
Some Common BASIC Programs
by L. Poole and M. Borchers
Payroll with Cost Accounting - CBASIC
by Lon Poole et al.
Accounts Payable and Accounts Receivable - CBASIC
by Lon Poole et al.
General Ledger - CBASIC
by Lon Poole et al.
Some Common Basic Programs — PET/CBM
edited by Lon Poole et al.

Z80

Z80 ASSEMBLY LANGUAGE PROGRAMMING
BY LANCE A. LEVENTHAL

Z80 ASSEMBLY LANGUAGE PROGRAMMING provides comprehensive coverage of the Z80 microprocessor assembly language. Programming examples illustrate software development concepts and actual assembly language usage. Assemblers and assembler directives are also explained.

Features include:

- More than 80 sample programming problems
- All problem solutions in source code and object code
- Each Z80 instruction fully explained
- Complete Z80 instruction set reference table
- Z80 Assembler conventions
- Z80 I O devices and interfacing methods
- Comparisons of Z80 and 8080A 8085 instruction sets and interrupt structure