Pilot Study of Applicability of a Generic Microprocessor Assembly Language

Joseph H. Bartlett

University of Central Florida

STARS Citation
PILOT STUDY OF THE APPLICABILITY OF A GENERIC
MICROPROCESSOR ASSEMBLY LANGUAGE

BY
JOSEPH HENRY BARTLETT
B.S., University of Central Florida, 1978

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Graduate Studies Program
of the College of Engineering
University of Central Florida
Orlando, Florida

Fall Term
1984
ABSTRACT

The purpose of this investigation is to research the utility of a standardized generic microprocessor assembly language. More precisely, use of a generic language implementation on a given microprocessor and its effect on programmer productivity will be investigated. Programmer productivity will be scored in terms of an inverse function of the time taken to complete a programming task correctly. Shorter times imply better programmer productivity and longer times imply the opposite.
ACKNOWLEDGEMENTS

I wish to thank my friend, Ron Elliott, for all of the assistance, guidance, encouragement and motivation which he provided at the appropriate times and without which this thesis could not have been completed.

I also wish to thank Dana Welch for being such a good friend.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II</td>
<td>THE PROBLEM</td>
<td>7</td>
</tr>
<tr>
<td>III</td>
<td>STATISTICS</td>
<td>9</td>
</tr>
<tr>
<td>IV</td>
<td>EQUIPMENT</td>
<td>13</td>
</tr>
<tr>
<td>V</td>
<td>SUBJECTS</td>
<td>15</td>
</tr>
<tr>
<td>VI</td>
<td>EXPERIMENTAL METHODOLOGY</td>
<td>16</td>
</tr>
<tr>
<td>VII</td>
<td>RESULTS</td>
<td>19</td>
</tr>
<tr>
<td>VIII</td>
<td>DISCUSSION OF RESULTS</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Appendices</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>BIBLIOGRAPHY</td>
<td>89</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

The emergence of a wide assortment of microprocessors in recent years has presented a variety of challenges to those responsible for software development. This uncontrolled proliferation of microprocessors and their associated assembly languages have been the key obstacles to wider usage of the latest microprocessors in new applications. Although the sophistication and number of high order microprocessor languages, which are essentially machine independent, have been hot topics in today's literature, the need for assembly language programming will always be with us. The improvement in speed and memory economy possible with assembly language programming make it the language of need and/or choice in hardware intensive applications and in real time situations.

The phenomena of today's microprocessor industry is not only the proliferation of faster and more efficient microprocessors, but also that hardware costs are continuing to decrease while software costs are increasing at escalating rates. Companies face the no win battle of deciding between adopting a new microprocessor because of its more desirable
capabilities (and thus abandoning their existing assembly language software base) or maintaining their current microprocessor configuration (and thus facing the consequences of forfeiting a possible competitive edge).

Assembly languages are, more often than not, determined by the manufacturers with little regard for the software problem at hand. Microprocessor manufacturers copywrite their mnemonic instruction lists to help preserve proprietary software. This has forced the use of nonstandard instruction mnemonics, assembler directives, addressing modes, etc. upon the programmer. It is obvious that the inconsistencies between assembly languages for microprocessors need to be transformed to provide a more workable software environment.

A large body of microprocessor assembly language programmers—hobbyists, students, teachers and professional programmers—should benefit greatly from use of a standardized generic microprocessor assembly language.

Time spent on the learning curve to become proficient with assembly language programming on a new microprocessor should be reduced considerably. The fact that the programmer is already knowledgeable of the instruction mnemonics, data and addressing formats, and assembler directives would allow him to concentrate on
learning only the differences in the particular assembly language standard instruction subset implemented on the current microprocessor.

Elaborate algorithms coded in a nonstandard assembly language on one microprocessor would no longer be essentially unintelligible to someone experienced with another instruction set. This would thus allow, and probably encourage, a freer flow of information in the literature.

Programming environments on dedicated development systems and mini or mainframe computers could be used more effectively. The time to develop a resident assembler and/or cross assembler would be reduced. Increasing efficiency in the development of a programming environment for a new microprocessor would allow quicker release of hardware and software support and development facilities.

Software maintenance, which can account for a very large portion of the total software cost, should also benefit from a standardized generic microprocessor assembly language. Maintenance is often performed by someone not familiar with the application being maintained and not experienced with the assembly language itself.
There would, of course, be problems associated with putting a standardized microprocessor assembly language into effect on a large scale. Actual implementation would be a relatively large if not impossible task.

The repertoire of instructions for specific microprocessors could sometimes be very different. A microprocessor dedicated to signal processing would probably have only a few instructions similar to those of the more common microprocessors.

The number and types of condition codes or status flags, and setting and resetting of these codes are all microprocessor hardware dependent.

Each programmer shall need to be thoroughly familiar with the functional operation and the standardized generic assembly language subset implemented on the given microprocessor.

The actual degree of effectiveness of a standardized microprocessor assembly language can only be truly ascertained after its extended use.
The IEEE Task P694/D11 (Fischer et al. 1979) is a standard which proposes to consolidate existing assembly language features and conventions for present and future microprocessors. Its goal is to standardize the usage of instruction names, mnemonics, address modes, labels, comments and assembler directives. The standard should help to improve programmer productivity.

Cross assemblers that are used to generate and communicate software from the host computer to the target system have been in constant demand. Methods to generate an automated cross assembler development facility for new microprocessors are currently in use (Johnson et al., 1977, Korn, 1975 and Cohen et al., 1979).

High level languages such as UCSD Pascal, which compile to P-code, are in widespread use on a variety of different microprocessors. The P-code instructions for the given implementation are converted to machine code via the microprocessor dependent P-code compiler (Crespi-Reghizzi et al., 1980).

The actual usage of given operations in instruction sets has been investigated (Fairclough, 1982). It was found that even with the large instruction sets now available on microprocessors, a relatively small number of instructions comprise the most widely used.
The "ease of use" of current mnemonic-based microprocessor assembly languages may be outclassed by symbolic assemblers and structured programming techniques (Kriger, 1979 and Crespi-Reghizzi et al., 1980).

Implementation on the Zilog Z80 microprocessor of the Intel 8080 mnemonic instruction set has been accomplished by Technical Design Labs. The TDL assembler provides an extension of 8080 mnemonics to include the entire Z80 instruction repertoire. This assembler has been in widespread usage ever since the beginning of the Z80's popularity.
CHAPTER II
THE PROBLEM

The purpose of this investigation is to research the utility of a standardized generic microprocessor assembly language. More precisely, use of a generic language implementation on a given microprocessor and its effect on programmer productivity will be investigated. Programmer productivity will be scored in terms of an inverse function of the time taken to complete a programming task correctly. Shorter times imply better programmer productivity and longer times imply the opposite.

Of particular interest is the investigation of this productivity when the microprocessor is unfamiliar to the programmer. The manufacturer's assembly language would then be a new language to learn. The standardized generic language would allow knowledge of its previous use to be carried over to the current application.

This report investigates the significance of the following hypothesis:

$H_0$: There is no difference in programmer "productivity" when using a manufacturer's assembly language versus using a standardized generic
assembly language when both are implemented on an unfamiliar microprocessor.

H₁: There is a difference in programmer productivity when using the above described assembly languages.

Instead of investigating an actual standardized generic assembly language applicable across several microprocessors (as outlined in Appendix A), the use of one of the generic assembly languages implemented will be investigated. It will be considered a generic language with extensions specific to a given microprocessor.

Technical Design Labs released their TDL mnemonics when the Zilog Z80 microprocessor first came on the scene. TDL assembly language is simply a superset of the Intel 8080 assembly language extended to include the entire Z80 instruction set. The term generic assembly language will be used in the remainder of this paper when referring to TDL assembly language.

Inference shall be made by induction that a standardized assembly language (such as IEEE Task P694/D11) does or does not improve "productivity" as previously defined.
CHAPTER III

STATISTICS

A testing procedure was designed to determine whether or not there was a difference in programmer "productivity" when using the generic assembly language (TDL Z80) versus the manufacturer's assembly language (Zilog Z80). This testing procedure consisted of one test program which the subjects coded using the generic assembly language (TDL Z80) and the manufacturer's assembly language (Zilog Z80). The purpose of this program was to locate the first occurrence of the ASCII letters "AB" in a memory block. The memory block starts at location 1000H and continues through and includes 10FFH. The memory block is first searched for the character "A". When an "A" is found, the next location is compared with the letter "B". If a "B" is found in this location the address of the start of the "AB" character sequence is written into locations 1100H and 1101H (least significant byte first). The search is continued at most 255 times if no match is found.

The test population was divided into two groups at random, each containing approximately half of the subjects. Group 1 programmed first using the manufacturer's assembly language (Zilog Z80) and then
the generic assembly language (TDL Z80). Group 2 programmed first using the generic assembly language (TDL Z80) and then the manufacturer's assembly language (Zilog, Z80). The programmer "productivity" was calculated by using the mean of each of the two groups of test times. That is, the mean time of the test using the generic assembly language (TDL Z80) was compared to the mean time of the test using the manufacturer's assembly language (Zilog Z80). The larger mean time implies lower productivity. The standard deviation was used to give a quantitative figure for how large an experimental spread existed in each group about its mean. The t-test was used to ascertain statistical significance of the experiment results.

The mean time for each group was calculated as follows:

$$t_{mn} = \frac{\sum_{i=1}^{N_n} t_{in}}{N_n}$$

where

- $n = 1$ Manufacturer group
- $n = 2$ Generic group

$N_n =$ Number of subjects for group $n$

$t_{in} =$ Time of successful completion for the $i$th subject for group $n$
and the standard deviation for each group was determined by:

\[ S = \sqrt{\frac{\sum_{i=1}^{N_n} (t_{in} - t_{out})^2}{N_n - 1}} \]

The statistical significance of the two sets of data was found by calculating the t-value:

\[ t = \frac{t_{m1} - t_{m2}}{\sqrt{s^2/N_1 + s^2/N_2}} \]

where

\[ s^2 = \frac{\sum_{i=1}^{N_1} (t_{i1} - t_{m1})^2 + \sum_{i=1}^{N_2} (t_{i2} - t_{m2})^2}{N_1 + N_2 - 2} \]

The degrees of freedom were found by:

\[ df = N_1 + N_2 - 2 \]
A significance level of 0.20 was used to indicate a correlation in the experimental data. The large significance value was used in this pilot study to better decrease the likelihood of making a type II error (test does not actually show a difference when there is a real one) for small N. This significance choice increased the chances of finding a difference if there really was one, which was the purpose of the pilot study.
CHAPTER IV  
EQUIPMENT

An Altos Z80 based microcomputer using the CP/M operating system was used to conduct the research. A Hazeltine 1500 video terminal was used for input and output, and a Qume printer was available to produce hardcopy.

A Z80 assembler capable of using either Zilog mnemonics (manufacturer) or TDL mnemonics (generic) was used to generate machine code from a stored assembly language source file. A word processor editor was used to enter and edit the source file.

Command files were used to initiate the assembler, load the object file, execute it, and display the results of each program run. The full screen editor is relatively straightforward to use, has a help function available for command reference, uses cursor control keys to facilitate moving the cursor to any position on the screen, and allows insertion and deletion of characters and lines.

The primary reason this equipment and system software was chosen is because it was readily available
on a day-to-day basis. Learning to use the assembler and editor was anticipated to be and was of little concern.
CHAPTER V
SUBJECTS

To achieve statistically valid conclusions, as large a population as possible should be used. It would have been advantageous to have had a large subject sample, but only five were found who were available. There was no requirement as to age, sex or experience.

It was required, though, that subjects be familiar with Intel 8080 assembly language mnemonics (which acted as the generic language in this study), and who were relatively unfamiliar with the Z80 Zilog mnemonics, (which acted as the manufacturer's new assembly language).

A pretest interview was used to ascertained information from each subject as to their experience with assembly language, familiarity with the Z80 and 8080, and software expertise in general.
CHAPTER VI
EXPERIMENTAL METHODOLOGY

The test population was divided into two groups at random, each containing approximately half of the subjects. Group 1 programmed using the manufacturer's assembly language (Zilog Z80) first and then the generic assembly language (TDL Z80). Group 2 programmed using the generic assembly language (TDL Z80) first and then the manufacturer's assembly language (Zilog Z80).

Information on both Z80 assembly language instruction sets, the editor, the assembler and the debugger was supplied to each participant one hour prior to the start of the timed test. This fixed time provided sufficient instruction study time for all subjects to become familiar with the instruction sets, equipment and software tools to be used.

Each subject was asked to write a given program in both Zilog Z80 and TDL Z80 mnemonics. All subjects were given the same program to implement in both languages. The program algorithm was outlined in written form, Program Design Language form and flowchart form.
Information on each Z80 assembly language instruction set, editor, assembler and debugger were also available during testing.

The study's dependent variable was the time needed to correctly complete the given programming task. This is related to productivity as previously discussed.

The independent variable was the language being used in the given programming task, the manufacturer's assembly language or the standardized generic assembly language.

Unfortunately other variables may well have affected this study drastically. Variables that the author had some degree of control over through the subject selection process include the subjects software experience and previous experience with either Z80 or 8080 assembly languages.

Because the subject population was small, it was decided that each individual would program the same algorithm in each of the two languages to increase sample size for each language. This of course introduced other problems. One was in test learning. A learning process was bound to occur due to the experience gained from the use of the first language. This problem was minimized by having half the population start by using one language, and the other half start by
using the other language. Therefore, learning during the test was effectively eliminated as a concern.

Also, in planning it was recognized that there may have been particular individuals who would have problems understanding the program algorithm. To help alleviate this possibility, the algorithm was described in several different ways, in verbal form, in a flowchart and in Program Design Language form.

It was also thought that problems might have cropped up concerning ease of use of the equipment for particular subjects. The assembler and editor were user friendly which helped to reduce possible problems in this area.

Most of the problems mentioned above would not be significant if a large enough population were available. The effect of one subject (or a small number of subjects) on the statistics would not then be so catastrophic to the study.
CHAPTER VII

RESULTS

The following table shows the time each subject took to successfully complete the given programming task using both Zilog mnemonics and standardized generic TDL mnemonics.

<table>
<thead>
<tr>
<th>SUBJECT</th>
<th>TIME (MINUTES)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ZILOG MNEMONICS</td>
<td>TDL MNEMONICS</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>27</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>42</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>45</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>AVERAGE</td>
<td>33.4</td>
<td>29.6</td>
<td></td>
</tr>
<tr>
<td>STD DEV</td>
<td>9.8</td>
<td>6.1</td>
<td></td>
</tr>
</tbody>
</table>
The following histogram compares the time taken to successfully complete the program task using both Zilog and standardized generic TDL mnemonics for all subjects.
The t value is computed as described in the statistics section. It is found as follows:

$$S^2 = \frac{385.2 + 151.2}{8} = 67.05$$

$$t = \frac{33.4 - 29.6}{\sqrt{\frac{2(67.05)}{5}}} = .73$$

The experimental t-test value for 8 degrees of freedom is 0.73.
CHAPTER VIII
DISCUSSION OF RESULTS

This pilot study was conducted with the gracious aid of its five subjects who donated several hours of their time. Each subject was given the program and instructions included in Appendix B. The subjects were given one hour to become familiar with the editor, assembler and debugger, as well as the individual mnemonic instruction sets.

No particularly insurmountable problems were encountered, although each subject spent about two to three hours overall on the test. All subjects had very little trouble becoming familiar with the editor, assembler and debugger.

All five subjects used in this pilot study would be considered "expert" programmers, but they have varying degrees of assembly language experience. Subjects 1, 2 and 4 had extensive experience with both Intel 8080 and Zilog Z80 microprocessors. Subjects 3 and 5 had passing knowledge of Intel 8080 assembly language only.

Subjects 1, 3 and 5 were given the task to program using TDL standardized generic mnemonics first, and then Zilog. Subjects 2 and 4 were given the tests in the opposite order.
The literature search turned up no examples of experimental investigation into the use of generic standardized assembly language.

The general trend of the data shows slightly shorter times when using the standardized generic TDL mnemonics. But in most cases there did not seem to be a very significant difference in programming time. The standard deviation for each set of samples was calculated to be fairly large.

One subject successfully completed the task in a shorter time using Zilog mnemonics. This particular case is probably attributable to the fact that this subject was extremely familiar with Zilog assembly language.

Formally, the statistical t-test pilot study value was considerably smaller (0.73<<1.387) than the value found in the t-test tables (Shneiderman, 1980). Therefore, no significant statistical difference can be shown in this study. The pilot study hypothesis is thus left unconfirmed.

Although the null hypothesis was not disproven, further investigation into the problem area has much merit. The small population size allowed the large variety of assembly language experience in the sample population to become the major factor controlling the outcome of the study.
The t-test is a powerful statistical test, but there are several conditions, however, which must be satisfied to insure confidence in its probability statements. The following are the major conditions:

1) observations must be independent
2) observations must be normally distributed
3) observations must have the same variance.

Except for the condition of equal sample variances, these conditions are not normally tested. They are generally presumed to be true unless there is evidence to the contrary (Davies, 1960).

For this investigation, Condition 1 is inferred from the nature of the experiment. Condition 3 can be verified from the experimental data and Condition 2 can only be achieved by using enough study subjects.

The number of observations required in the t-test to compare mean values is dependent upon their standard deviation, and the desired significance of type I ($\alpha$) and II ($\beta$) errors. This number is determined from a table of N values versus $\alpha$, $\beta$, $\sigma$ and $\delta$ ($\delta$ is the smallest time that can be assumed to be significant in the study). If $\alpha = \beta = 0.05$ and $D = \frac{\delta}{\sigma} = 5/9.8 = 0.5$, then the number of observations needed in a t-test to determine the significance of the difference between two sample means is 110 (Davies, 1960).
The current pilot study does not provide for acceptance of the hypothesis, but does provide the sample group's standard deviation which is necessary to estimate the sample size to be used in a more complete study.

It is recommended that a future investigation of this type, comparing the use of Zilog and standardized generic TDL mnemonics, be implemented. This proposed study should follow the guidelines of the current pilot study. The sample size for each assembly language task group should be at least on the order of 110 to insure accuracy of the t-test statistics.

A second proposed study should eventually investigate the IEEE standardized assembly language implementation on a number of 8- and 16-bit microprocessors. Perhaps a performance test of comprehension can be better used to study a very large subject sample set than the pilot study's test procedure.

Given a particular program implemented on all available microprocessors, fill-in-the-blank questions could be asked concerning output for given inputs, inputs necessary to achieve a given output, impact of minor program alterations, and sequence of procedures executed (Shneiderman, 1980). Scoring could be less
subjective by using several graders and the test would be made less time consuming to the subject participants.

A number of problems will probably show up when a portable generic microprocessor assembly language is implemented. The microprocessor machine architecture will probably not lend itself easily to assembly language standardization. The problems involved include: differing word lengths, addressing modes and byte versus word addressing, flags and status, separate instruction and data spaces, total memory addressing space available, and the wide assortment of register and input-output configurations.

The complete study of programmer productivity when using a standardized generic microprocessor assembly language is overwhelming. It shall require many programmer subjects, encompass many processors and extend over several years of study. It is a task that shall probably be attacked in a piecemeal fashion, one facet at a time.
APPENDIX A
IEEE GENERIC INSTRUCTION SET

This section describes the functional operation of each standard instruction. The operation described may be implemented in any microprocessor (independent of word length) with the appropriate conditions. This standard does not necessarily define the circumstances under which microprocessor conditions are set or cleared, but implies possible usage. The grouping of instructions in this section is arbitrary, and is not intended to imply necessary relationships.

INSTRUCTION NAMES: The naming of instructions shall be in accordance with the following rule: instruction names shall begin with an action verb. Examples are: Add with Carry, Rotate Right, Branch if Less Than, And, Return if Zero, Shift Left, Test, etc. Certain exceptions, the results of common usage, are noted herein.

INSTRUCTION MNEMONICS: The selection of mnemonics for instructions not contained in this standard shall be in accordance with the following rules (exceptions are noted herein):
a) The first character of the mnemonic shall be the first letter of the action verb.

b) Addressing modes shall not be embedded in the mnemonic.

c) Operand designations shall not be embedded in the mnemonic.

d) Conditions shall be embedded in the mnemonic.

e) Operand type may be indicated, where appropriate, by the last character of the mnemonic as shown below (the default operand type is word):

<table>
<thead>
<tr>
<th>Character</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Byte</td>
</tr>
<tr>
<td>H</td>
<td>Halfword</td>
</tr>
<tr>
<td>L</td>
<td>Long (Double Word)</td>
</tr>
<tr>
<td>D</td>
<td>Decimal</td>
</tr>
<tr>
<td>F</td>
<td>Floating Point</td>
</tr>
<tr>
<td>l</td>
<td>Bit</td>
</tr>
<tr>
<td>4</td>
<td>Nibble or Digit</td>
</tr>
<tr>
<td>M</td>
<td>Multiple</td>
</tr>
</tbody>
</table>

SYNONYMOUS MNEMONICS: Depending on the microprocessor architecture, several standard mnemonics may assemble into the same machine instruction. In those cases, all such mnemonics shall be included in the assembly language.

MULTIFUNCTION INSTRUCTIONS: The representation of multifunction instructions shall be by the use of two or more standard mnemonics on the same line, unless a standard mnemonic exists which describes the
multifunction instruction, in which case that mnemonic shall be used in the assembly language.

CONDITIONAL INSTRUCTIONS: Conditional instruction mnemonics shall be constructed by concatenating the generic instruction name with the condition name. An example would be "Branch if Zero" (BZ), which is formed from an abbreviated Branch (B-) and "if Zero" (Z-). When the opposite condition state is used, then the letter "N" for "Not True" or "No" shall be inserted between the instruction mnemonic and the condition mnemonic to define the false condition as in "Branch if Not Zero" (BNZ).

Conditions are generally utilized with the following instruction types:

a) Branch (B-)
b) Skip (SK-)
c) Call subroutine (CALL-)
d) Return from subroutine (RET-)
e) Increment and Branch (IB-)
f) Increment and Skip (ISK-)
g) Decrement and Branch (DB-)
h) Decrement and Skip (DSK-)

The standard condition mnemonics are defined in this section. The dash character "-" represents the instruction mnemonic letter(s) to be replaced with the generic instruction name.
1. **Zero (-Z)** The instruction is executed if the zero condition is true. Note that this condition may be the same as the Equal condition.

2. **Not Zero (-NZ)** The instruction is executed if the zero condition is false. Note that this condition may be the same as the Not Equal condition.

3. **Equal (-E)** The instruction is executed if the equal condition is true. Note that this condition may be the same as the Zero condition.

4. **Not Equal (-NE)** The instruction is executed if the equal condition is false. Note that the condition may be the same as the Not Zero condition.

5. **Carry (-C)** The instruction is executed if the carry condition is true.

6. **No Carry (-NC)** The instruction is executed if the carry condition is false.

7. **Positive (-P)** The instruction is executed if the positive condition is true.

8. **Negative (-N)** The instruction is executed if the negative condition is true.

9. **Overflow (-V)** The instruction is executed if the arithmetic overflow condition is true.

10. **No Overflow (-NV)** The instruction is executed if the arithmetic overflow condition is false.
11. **Greater Than** (−GT) The instruction is executed if an arithmetic (signed) greater than condition exists. This condition is not equivalent to the Higher condition.

12. **Greater Than Or Equal** (−GE) The instruction is executed if an arithmetic (signed) greater than or equal condition exists. This condition is not equivalent to the Not Lower condition.

13. **Less Than** (−LT) The instruction is executed if an arithmetic (signed) less than condition exists. This condition is not equivalent to the Lower condition.

14. **Less Than Or Equal** (−LE) The instruction is executed if an arithmetic (signed) less than or equal condition exists. This condition is not equivalent to the Not Higher condition.

15. **Higher** (−H) The instruction is executed if an unsigned greater than condition exists. This condition is not equivalent to the Greater Than condition.

16. **Not Higher** (−NH) The instruction is executed if an unsigned less than or equal condition exists. This condition is not equivalent to the Less Than or Equal condition.

17. **Lower** (−L) The instruction is executed if an unsigned less than condition exists. This condition is not equivalent to the Less Than condition.
18. **Not Lower (-NL)** The instruction is executed if an unsigned greater than or equal condition exists. This condition is not equivalent to the Greater Than or Equal condition.

19. **Parity Even (-PE)** The instruction is executed if the even parity condition exist. This condition is the negation of Parity Odd.

20. **Parity Odd (-PO)** The instruction is executed if the odd parity condition exist. This condition is the negation of Parity Even.

ARITHMETIC INSTRUCTIONS: The following are arithmetic instructions.

1. **Add (ADD)** This instruction performs an addition.

2. **Add With Carry (ADDC)** This instruction performs an addition and adds any previous carry to the result.

3. **Subtract (SUB)** This instruction performs a subtraction.

4. **Subtract Reverse (SUBR)** This instruction performs a subtraction in reverse order.

5. **Subtract with Carry/Borrow (SUBC)** This instruction performs a subtraction and incorporates a previous borrow into the result. The borrow may or may not be related to the carry.

6. **Increment (INC)** This instruction causes a one to be added to the specified operand.
7. **Decrement (DEC)** This instruction causes a one to be subtracted from the specified operand.

8. **Multiply (MUL)** This instruction performs a multiplication.

9. **Divide (DIV)** This instruction performs a division.

10. **Compare (CMP)** This instruction does a comparison and sets the appropriate condition(s) according to the results.

11. **Negate (NEG)** This instruction causes the specified operand to be replaced with its arithmetic negative (two's complement).

12. **Extend (EXT)** This instruction extends an operand to fill a specified larger field.

**LOGICAL INSTRUCTIONS:** The following are logical instructions.

1. **And (AND)** This instruction performs a logical "AND".

2. **Or (OR)** This instruction performs a logical "OR".

3. **Exclusive Or (XOR)** This instruction performs a logical "Exclusive OR". Note that this instruction mnemonic violates the mnemonic naming rule, but is retained in deference to common usage.

4. **Not (NOT)** This instruction causes the specified operand to be replaced with its one's complement (logical not).
5. **Not Carry (NOTC)** This instruction causes the carry condition to be complemented.

6. **Shift Right (SHR)** This instruction causes the specified operand to be shifted one or more places to the right (toward the LSB), with the most significant bit(s) being replaced with zero(s).

7. **Shift Left (SHL)** This instruction causes the specified operand to be shifted one or more places to the left (toward the MSB), with the least significant bit(s) being replaced with zero(s).

8. **Shift Right Arithmetic (SHRA)** This instruction causes the specified operand to be shifted one or more places to the right with the most significant bit (sign) being preserved and propagated to the right.

9. **Rotate Right (ROR)** This instruction causes the specified operand to be shifted one or more places to the right, with the MSB being replaced by the LSB on each shift.

10. **Rotate Left (ROL)** This instruction causes the specified operand to be shifted one or more places to the left, with the LSB being replaced by the MSB on each shift.

11. **Rotate Right Through Carry / Link (RORC)** This instruction causes the specified operand to be shifted one or more places to the right with the previous state
of the link being loaded into the MSB, and the LSB being loaded into the link. Note that the link may be associated with the carry flag.

12. **Rotate Left Through Carry / Link (ROLC)** This instruction causes the specified operand to be shifted one or more places to the left with the previous state of the link being loaded into the LSB, and the MSB being loaded into the link.

13. **Test (TEST)** This instruction causes the specified operand to be tested and sets the appropriate condition(s) according to the result.

**DATA TRANSFER INSTRUCTIONS:** The following are data transfer instructions.

1. **Load (LD)** This instruction causes the contents of a memory location specified as the source to be transferred to a register specified as the destination.

2. **Store (ST)** This instruction causes the contents of a register specified as the source to be transferred to a memory location specified as the destination.

3. **Move (MOV)** This instruction causes the contents of a register to be transferred to another register, or the contents of a memory location to be transferred to another memory location.

4. **Move Block (MOVBK)** This instruction causes the transfer of a block of data.
5. Move Multiple (MOVM) This instruction causes the contents of a memory location to be copied into multiple memory locations.

6. Exchange (XCH) This instruction causes the specified operands to be exchanged.

7. Input (IN) This instruction causes the data at an input port to be transferred to a register or memory location.

8. Output (OUT) This instruction causes the contents of a register or a memory location to be transferred to an output port.

9. Clear (CLR) This instruction causes the specified operand to be replaced by zero(s).

10. Clear Carry (CLRC) This instruction causes the carry to be set to the not true or no carry state.

11. Clear Overflow (CLRV) This instruction causes the overflow to be set to the not true or no overflow state.

12. Set (SET) This instruction causes the specified operand to be replaced by one(s).

13. Set Carry (SETC) This instruction causes the carry to be set to the true or carry state.

14. Set Overflow (SETV) This instruction causes the overflow to be set to the true or overflow state.

BRANCH INSTRUCTIONS: The following are branch instructions.
1. **Branch (BR)** This instruction causes the contents of the program counter to be replaced by the effective address, thereby transferring control to the memory location specified by that address. The condition(s) for execution of the following instructions were described earlier. For brevity, only the instruction titles and mnemonics of the branch instructions are given here.

2. **Branch If Zero (BZ)**
3. **Branch If Not Zero (BNZ)**
4. **Branch If Equal (BE)**
5. **Branch If Not Equal (BNE)**
6. **Branch If Carry (BC)**
7. **Branch If No Carry (BNC)**
8. **Branch If Positive (BP)**
9. **Branch If Negative (BN)**
10. **Branch If Overflow (BV)**
11. **Branch If No Overflow (BNV)**
12. **Branch If Greater Than (BGT)**
13. **Branch If Greater Than Or Equal (BGE)**
14. **Branch If Less Than (BLT)**
15. **Branch If Less Than Or Equal (BLE)**
16. **Branch If Higher (BH)**
17. **Branch If Not Higher (BNH)**
18. **Branch If Lower (BL)**
19. **Branch If Not Lower (BNL)**
20. **Branch If Parity Even (BPE)**

21. **Branch If Parity Odd (BPO)**

**SKIP INSTRUCTIONS:** The following are skip instructions.

1. **Skip (SKIP)** This instruction causes the program counter to be incremented such that the execution of the next instruction(s) is skipped. The condition(s) for execution of the following instructions were described earlier. For brevity, only the instruction titles and mnemonics of the skip instructions are given here.

2. **Skip If Zero (SKZ)**

3. **Skip If Not Zero (SKNZ)**

4. **Skip If Equal (SKE)**

5. **Skip If Not Equal (SKNE)**

6. **Skip If Carry (SKC)**

7. **Skip If Not Carry (SKNC)**

8. **Skip If Positive (SKP)**

9. **Skip If Negative (SKN)**

10. **Skip If Overflow (SKV)**

11. **Skip If No Overflow (SKNV)**

12. **Skip If Greater Than (SKGT)**

13. **Skip If Greater Than Or Equal (SKGE)**

14. **Skip If Less Than (SKLT)**

15. **Skip If Less Than Or Equal (SKLE)**

16. **Skip If Higher (SKH)**
SUBROUTINE CALL INSTRUCTIONS: The following are subroutine call instructions.

1. **Call Subroutine (CALL)** This instruction causes the program counter to be saved and replaced by the specified operand, thereby transferring control to the memory location specified by the operand. The condition(s) for execution of the following instructions were described earlier. For brevity, only the instruction titles and mnemonics of the call instructions are given here.

2. **Call If Zero (CALLZ)**

3. **Call If Not Zero (CALLNZ)**

4. **Call If Equal (CALLE)**

5. **Call If Not Equal (CALLNE)**

6. **Call If Carry (CALLC)**

7. **Call If No Carry (CALLNC)**

8. **Call If Positive (CALLP)**

9. **Call If Negative (CALLN)**

10. **Call If Overflow (CALLV)**

11. **Call If No Overflow (CALLNV)**

12. **Call If Greater Than (CALLGT)**
13. **Call If Greater Than Or Equal (CALLGE)**
14. **Call If Less Than (CALLLT)**
15. **Call If Less Than Or Equal (CALLLE)**
16. **Call If Higher (CALLH)**
17. **Call If Not Higher (CALLNH)**
18. **Call If Lower (CALLL)**
19. **Call If Not Lower (CALLNL)**
20. **Call If Parity Even (CALLPE)**
21. **Call If Parity Odd (CALLPO)**

**RETURN INSTRUCTIONS:** The following are return instructions.

1. **Return From Subroutine (RET):** This instruction causes the previously saved contents of the program counter to be restored, thereby returning control to the routine that called the subroutine or was interrupted. The condition(s) for execution of the following instructions were described earlier. For brevity, only the instruction titles and mnemonics are given for the conditional instructions.

2. **Return If Zero (RETZ)**
3. **Return If Not Zero (RETNZ)**
4. **Return If Equal (RETZ)**
5. **Return If Not Equal (RETNE)**
6. **Return If Carry (RETC)**
7. **Return If No Carry (RETNC)**
8. Return If Positive (RETP)
9. Return If Negative (RETN)
10. Return If Overflow (RETV)
11. Return If No Overflow (RETNV)
12. Return If Greater Than (RETGT)
13. Return If Greater Than Or Equal (RETGE)
14. Return If Less Than (RELT)
15. Return If Less Than Or Equal ( RETLE)
16. Return If Higher (RETH)
17. Return If Not Higher (RETNH)
18. Return If Lower (RETL)
19. Return If Not Lower (RETNL)
20. Return If Parity Even (RETPE)
21. Return If Parity Odd (RETPO)
22. Return With Skip (RETSK) This instruction causes the previously saved contents of the program counter to be incremented some amount and restored, thereby returning control to the routine that called the subroutine at some point after the subroutine call.
23. Return From Interrupt (RETI) This instruction returns control to the routine that was interrupted.

MISCELLANEOUS INSTRUCTIONS: The following are miscellaneous instructions.
1. No Operation (NOP) This instruction causes the processor to take no action other than to advance to the
next instruction. This instruction's name violates the naming rules, but is kept in deference to common usage.

2. **Push** *(PUSH)* This instruction causes the contents of the specified operand(s) to be transferred to the top of a stack.

3. **Pop** *(POP)* This instruction causes the contents of the top of a stack to be transferred to the designated operand(s).

4. **Halt** *(HALT)* This particular instruction causes the microprocessor to stop executing instructions until an external condition occurs.

5. **Wait** *(WAIT)* This particular instruction causes the microprocessor to stop executing instructions until an external or internal condition occurs or changes.

6. **Break** *(BRK)* This instruction causes an interrupt sequence to be initiated by the microprocessor.

7. **Adjust** *(ADJ)* This instruction makes an adjustment such that the operand or implied accumulator contents will represent the correct result, usually a binary-coded-decimal representation.

8. **Enable Interrupt** *(EI)* This instruction causes the designated interrupt(s) to be enabled.

9. **Disable Interrupt** *(DI)* This instruction causes the designated interrupt(s) to be disabled.
10. **Translate (TR)** This instruction references a specified table to replace an operand with value(s) selected from the table on the basis of the value of that operand.

**OPERANDS AND SYNTAX:** The following are operands and syntax.

1. **Addressing Modes** Addressing modes in microprocessors with more than one addressing mode shall be specified by special character(s). The special character(s) shall precede the address expression except where pre or post specification implies an operational sequence. Note that the address expression (addr) may refer to either a memory location or register. The following prefix and postfix characters shall be used to define the specified address modes:

<table>
<thead>
<tr>
<th>MODE</th>
<th>SYMBOL</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
<td>prefix /</td>
<td>/addr</td>
</tr>
<tr>
<td>Base page</td>
<td>prefix !</td>
<td>!addr</td>
</tr>
<tr>
<td>Indirect</td>
<td>prefix @</td>
<td>@addr</td>
</tr>
<tr>
<td>Relative</td>
<td>prefix $</td>
<td>$addr</td>
</tr>
<tr>
<td>Immediate</td>
<td>prefix #</td>
<td>#value</td>
</tr>
<tr>
<td>Index</td>
<td>enclosing parenthesis ()</td>
<td>addr(index)</td>
</tr>
<tr>
<td>Register</td>
<td>prefix.</td>
<td>.addr</td>
</tr>
<tr>
<td>Auto-pre-increment</td>
<td>prefix +</td>
<td>+addr</td>
</tr>
<tr>
<td>Auto-post-increment</td>
<td>postfix +</td>
<td>addr+</td>
</tr>
<tr>
<td>Auto-pre-decrement</td>
<td>prefix -</td>
<td>-addr</td>
</tr>
<tr>
<td>Auto-post-decrement</td>
<td>postfix -</td>
<td>addr-</td>
</tr>
<tr>
<td>Indirect-pre-indexed</td>
<td>prefix () @</td>
<td>addr(index)@</td>
</tr>
<tr>
<td>Indirect-post-indexed</td>
<td>prefix @,</td>
<td>@addr(index)</td>
</tr>
<tr>
<td></td>
<td>postfix ()</td>
<td></td>
</tr>
</tbody>
</table>
Assemblers may have the option of coercing the addressing mode for instructions that have only one addressing mode. As an example, a branch instruction which allows only relative addressing may be coded without the "$" character preceding the address designation in the operand field. Such coercion should be flagged in the assembly listing.

For microprocessors that have several address modes for a particular instruction, the assembler may select the address mode if the programmer does not specify it. The means used to indicate which address mode was selected shall be specified. The default address mode should be relative.

2. **Expressions** An assembler should allow the use of expressions which are evaluated at assembly time. When expression evaluation capabilities are included in the assembler, those expressions operators that are implemented shall be designated by the following infix special symbols:

<table>
<thead>
<tr>
<th>SYMBOL(S)</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>/</td>
<td>Divide (Signed)</td>
</tr>
<tr>
<td>/</td>
<td>Divide (Unsigned)</td>
</tr>
<tr>
<td>.AND.</td>
<td>AND</td>
</tr>
<tr>
<td>.OR.</td>
<td>OR</td>
</tr>
<tr>
<td>.XOR.</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>.NOT.</td>
<td>NOT</td>
</tr>
<tr>
<td>.SHL.</td>
<td>Left Shift</td>
</tr>
<tr>
<td>.SHR.</td>
<td>Right Shift</td>
</tr>
</tbody>
</table>

A bit alignment example: the expression $A^{p:q}$ means align bits $p$ through $q$ inclusive of $A$. Hierarchy is not specified. Parenthesis may be used to group expressions.

ASSEMBLER DIRECTIVES: The following are assembler directives.

1. **General** Assembler directives are commands to the assembler instead of instructions for the microprocessor. They direct the assembler to perform specific tasks during the assembly process.

   This standard does not specify the syntax necessary to support macros or conditional assembly.

   Naming of assembler directives and of assembler directive mnemonics shall follow the rules used for instructions. If the following functions are implemented, the specified mnemonic shall be used.

2. **Originate (ORG)** This assembler directive sets the current location counter to the value specified by the operand. The assembler shall initialize all location counters to zero at the beginning of the program.
3. **Equate (EQU)** This assembler directive equates a symbol to a constant, an address, or an expression.

4. **End (END)** This assembler directive informs the assembler that the end of source has been reached.

5. **Page (PAGE)** This assembler directive causes the assembler to advance the assembly listing to the top of the next page.

6. **Title (TITLE)** This assembler directive causes the assembler to advance the assembly listing to the next page and to insert the specified title into the header of that and each of the following pages.

7. **Date (DATA)** This assembler directive causes the assembler to fill the next memory location(s) with the specified value(s). A letter may be appended to the mnemonic as specified earlier to indicate data type.

8. **Reserve Memory (RES)** This assembler directive reserves a block of storage locations. The number of locations reserved is specified by a constant or an expression. The content of the reserved storage location(s) may be unspecified.

9. **Base (BASE)** This assembler directive causes the assembler to change the current implied number base.
### STANDARD INSTRUCTION MNEMONICS FOR 6800

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>STANDARD MNEMONIC</th>
<th>MOTOROLA MNEMONIC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARITHMETIC</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ADD</td>
<td>ADD,ABA</td>
</tr>
<tr>
<td>Add with Carry</td>
<td>ADDC</td>
<td>ADC</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
<td>SUB,SBA</td>
</tr>
<tr>
<td>Increment</td>
<td>INC</td>
<td>INC,INS,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INX</td>
</tr>
<tr>
<td>Decrement</td>
<td>DEC</td>
<td>DEC,DES,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DEX</td>
</tr>
<tr>
<td>Compare</td>
<td>CMP</td>
<td>CMP,CBA,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPX</td>
</tr>
<tr>
<td>Negate</td>
<td>NEG</td>
<td>NEG</td>
</tr>
<tr>
<td><strong>LOGICAL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>AND</td>
<td>AND</td>
</tr>
<tr>
<td>Or</td>
<td>OR</td>
<td>ORA</td>
</tr>
<tr>
<td>Exclusive Or</td>
<td>XOR</td>
<td>EOR</td>
</tr>
<tr>
<td>not</td>
<td>NOT</td>
<td>COM</td>
</tr>
<tr>
<td>Shift Right</td>
<td>SHR</td>
<td>LSR</td>
</tr>
<tr>
<td>Shift Left</td>
<td>SHL</td>
<td>ASL</td>
</tr>
<tr>
<td>Shift Right Arithmetic</td>
<td>SHRA</td>
<td>ASR</td>
</tr>
<tr>
<td>Rotate Right</td>
<td>ROR</td>
<td>ROR</td>
</tr>
<tr>
<td>Rotate Left</td>
<td>ROL</td>
<td>ROL</td>
</tr>
<tr>
<td>Test</td>
<td>TEST</td>
<td>BIT,TST</td>
</tr>
<tr>
<td><strong>DATA TRANSFER</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>LD</td>
<td>LDA, LDS,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LDX</td>
</tr>
<tr>
<td>Store</td>
<td>ST</td>
<td>STA,STS,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STX</td>
</tr>
<tr>
<td>Move</td>
<td>MOV</td>
<td>TAB, TBA,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TAP, TPA,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TSX, TXS</td>
</tr>
<tr>
<td>Clear</td>
<td>CLR</td>
<td>CLR</td>
</tr>
<tr>
<td>Clear Carry</td>
<td>CLRC</td>
<td>CLC</td>
</tr>
<tr>
<td>Instruction</td>
<td>Code</td>
<td>Code</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>Clear Overflow</td>
<td>CLRV</td>
<td>CLV</td>
</tr>
<tr>
<td>Set Carry</td>
<td>SETC</td>
<td>SEC</td>
</tr>
<tr>
<td>Set Overflow</td>
<td>SETV</td>
<td>SEV</td>
</tr>
<tr>
<td><strong>BRANCH</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>BR</td>
<td>BRA, JMP</td>
</tr>
<tr>
<td>Branch if Zero</td>
<td>BZ</td>
<td>BEQ</td>
</tr>
<tr>
<td>Branch if Not Zero</td>
<td>BNZ</td>
<td>BNE</td>
</tr>
<tr>
<td>Branch if Equal</td>
<td>BE</td>
<td>BEQ</td>
</tr>
<tr>
<td>Branch if Not Equal</td>
<td>BNE</td>
<td>BNE</td>
</tr>
<tr>
<td>Branch if Carry</td>
<td>BC</td>
<td>BCS</td>
</tr>
<tr>
<td>Branch if Positive</td>
<td>BP</td>
<td>BPL</td>
</tr>
<tr>
<td>Branch if Negative</td>
<td>BN</td>
<td>BMI</td>
</tr>
<tr>
<td>Branch if Overflow</td>
<td>BV</td>
<td>BVS</td>
</tr>
<tr>
<td>Branch if No Overflow</td>
<td>BNV</td>
<td>BVC</td>
</tr>
<tr>
<td>Branch if Greater Than</td>
<td>BGT</td>
<td>BGT</td>
</tr>
<tr>
<td>Branch if Greater Than or Equal</td>
<td>BGE</td>
<td>BGE</td>
</tr>
<tr>
<td>Branch if Less Than</td>
<td>BLT</td>
<td>BLT</td>
</tr>
<tr>
<td>Branch if Higher</td>
<td>BH</td>
<td>BHI</td>
</tr>
<tr>
<td>Branch in Not Higher</td>
<td>BNH</td>
<td>BLS</td>
</tr>
<tr>
<td>Branch if Lower</td>
<td>BL</td>
<td>BCS</td>
</tr>
<tr>
<td>Branch if Not Lower</td>
<td>BNL</td>
<td>BCC</td>
</tr>
<tr>
<td><strong>SUBROUTINE CALL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call Subroutine</td>
<td>CALL</td>
<td>BSR, JSR</td>
</tr>
<tr>
<td><strong>RETURN</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return from Subroutine</td>
<td>RET</td>
<td>RTS</td>
</tr>
<tr>
<td>Return from Interrupt</td>
<td>RETI</td>
<td>RTI</td>
</tr>
<tr>
<td><strong>MISCELLANEOUS</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No Operation</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>Push</td>
<td>PUSH</td>
<td>PSH</td>
</tr>
<tr>
<td>Pop</td>
<td>POP</td>
<td>PUL</td>
</tr>
<tr>
<td>Wait</td>
<td>WAIT</td>
<td>WAI</td>
</tr>
<tr>
<td>Adjust Decimal</td>
<td>ADJ</td>
<td>DAA</td>
</tr>
<tr>
<td>Enable Interrupt</td>
<td>EI</td>
<td>SEI</td>
</tr>
<tr>
<td>Disable Interrupt</td>
<td>DI</td>
<td>CLI</td>
</tr>
<tr>
<td>Break</td>
<td>BRK</td>
<td>SWI</td>
</tr>
</tbody>
</table>
### Standard Instruction Mnemonics for Z80, 8080, 8085

#### Instruction

<table>
<thead>
<tr>
<th>Standard Zilog Mnemonic</th>
<th>Standard Intel Mnemonic</th>
<th>Intel Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ADD</td>
<td>ADD,ADI,</td>
</tr>
<tr>
<td>Add with Carry</td>
<td>ADDC</td>
<td>ADC,ACI</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
<td>SUB,SUI</td>
</tr>
<tr>
<td>Subtract with Carry</td>
<td>SUBC</td>
<td>SBB,SBI</td>
</tr>
<tr>
<td>Increment</td>
<td>INC</td>
<td>INX,INR</td>
</tr>
<tr>
<td>Decrement</td>
<td>DEC</td>
<td>DCX,DCR</td>
</tr>
<tr>
<td>Compare</td>
<td>CMP</td>
<td>CMP,CPI</td>
</tr>
<tr>
<td>Compare, Multiple</td>
<td>CMPM</td>
<td>CPIR, CPDR</td>
</tr>
<tr>
<td>Negate</td>
<td>NEG</td>
<td>NEG</td>
</tr>
<tr>
<td><strong>Logical</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>AND</td>
<td>ANA,ANI</td>
</tr>
<tr>
<td>Or</td>
<td>OR</td>
<td>ORA, ORI</td>
</tr>
<tr>
<td>Exclusive Or</td>
<td>XOR</td>
<td>XRA, XRI</td>
</tr>
<tr>
<td>Not</td>
<td>NOT</td>
<td>CPL, CMA</td>
</tr>
<tr>
<td>Not Carry</td>
<td>NOTC</td>
<td>CCF, CMN</td>
</tr>
<tr>
<td>Shift Right</td>
<td>SHR</td>
<td>SLA</td>
</tr>
<tr>
<td>Shift Left</td>
<td>SHL</td>
<td>SRA ADD, DAD</td>
</tr>
<tr>
<td>Shift Right Arithmetic</td>
<td>SHRA</td>
<td>SRA</td>
</tr>
<tr>
<td>Rotate Right</td>
<td>ROR</td>
<td>RRCA, RRC</td>
</tr>
<tr>
<td>Rotate Left</td>
<td>ROL</td>
<td>RLCA, RLC</td>
</tr>
<tr>
<td>Rotate Right Through Carry</td>
<td>RORC</td>
<td>RR, RRA</td>
</tr>
<tr>
<td>Rotate Left Through Carry</td>
<td>ROLC</td>
<td>RL, RLA</td>
</tr>
<tr>
<td>Rotate Right Decimal</td>
<td>ROR4</td>
<td>RLD</td>
</tr>
<tr>
<td>Rotate Left Decimal</td>
<td>ROL4</td>
<td>RLD</td>
</tr>
<tr>
<td>Test Bit</td>
<td>TEST1</td>
<td>BIT</td>
</tr>
</tbody>
</table>
DATA TRANSFER

Load

LD

MOV,LXI,
LHLD,
LDA,MVI
MOV,STAX,
SHLD,
STA

Store

ST

LD

MOV,LDI,
LDD
MOV,MVI,
SPHL

Move

MOV

LD,LDI,
LDD

Move Block

MOVKB

LDIR,LDDR

Exchange

XCH

EX,EXX

XCHG,
XTHL

Input

IN

IN,INI,
IND

Input Block

INBK

INIR,INDR

Output

OUT

OUT,OUTI,
OUT,SIM

Output Block

OUTBK

OTIR,OTDR

Set Bit

SETI

DET

Clear Bit

CLRI

RES

Set Carry

SETC

SCF

STC

Set Interrupt Mode

SETI

IM

BRANCH

Branch

BR

JP

JMP,PCHL

Branch if Zero

BZ

JP Z, JR Z

JZ

Branch if Not Zero

BNZ

JP NZ,

JNZ

Branch if Equal

BE

JP Z, JR Z

JZ

Branch if Not Equal

BNE

JP NZ,

JNZ

Branch if Carry

BC

JP C, JR C

Branch if No Carry

BNC

JP NC,

JNC

Branch if Positive

BP

JP P

Branch if Negative

BN

JP M

Branch if Parity Even

BPE

JP PE

JPE

Branch if Parity Odd

BPO

JP PO

JPO

Branch if Low

BL

JP C

JC

Branch if Not Low

BNL

JP NC

JNC

Decrement and Branch if Not Zero

DBNZ

DJNZ

CALL

CALL

CALL, RST

CALLZ

CALL Z

CZ

CALLNZ

CALL NZ

CNZ

CALLLE

CALL Z

CZ
Call if Not Equal  CALLNE
Call if Carry       CALLCC
Call if No Carry    CALLNCC
Call if Positive    CALLPC
Call if Negative    CALLMC
Call if Parity Even CALLPE
Call if Parity Odd  CALLPO
Call if Low         CALLC
Call if Not Low     CALLNC

RETURN

Return           RET
Return if Zero    RETZ
Return if Not Zero RETNZ
Return if Equal   RETE
Return if Not Equal RETNE
Return if Carry   RETC
Return if No Carry RETNC
Return if Positive RETP
Return if Negative RETN
Return if Parity Even RETPE
Return if Parity Odd RETPO
Return if Lower   RETL
Return if Not Lower RETNL
Return from Interrupt RETI
Non-Maskable

MISCELLANEOUS

No operation     NOP
Push             PUSH
Pop              POP
Wait             WAIT
Adjust Decimal   ADJ
Enable Interrupt EI
Disable Interrupt DI
IEEE STANDARD INSTRUCTION MENEMONICS FOR THE 8086

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>STANDARD MNEMONIC</th>
<th>INTEL MNEMONIC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARITHMETIC</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ADD</td>
<td>ADD</td>
</tr>
<tr>
<td>Add with Carry</td>
<td>ADDC</td>
<td>ADC</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
<td>SUB</td>
</tr>
<tr>
<td>Subtract with Carry</td>
<td>SUBC</td>
<td>SBB</td>
</tr>
<tr>
<td>Increment</td>
<td>INC</td>
<td>INC</td>
</tr>
<tr>
<td>Decrement</td>
<td>DEC</td>
<td>DEC</td>
</tr>
<tr>
<td>Negate</td>
<td>NEG</td>
<td>NEG</td>
</tr>
<tr>
<td>Multiply</td>
<td>MUL</td>
<td>IMUL</td>
</tr>
<tr>
<td>Multiply, Unsigned</td>
<td>MULU</td>
<td>MUL</td>
</tr>
<tr>
<td>Divide</td>
<td>DIV</td>
<td>IDIV</td>
</tr>
<tr>
<td>Divide, Unsigned</td>
<td>DIVU</td>
<td>DIV</td>
</tr>
<tr>
<td>Compare</td>
<td>CMP</td>
<td>CMPW</td>
</tr>
<tr>
<td>Compare, Byte</td>
<td>CMPB</td>
<td>CMPB</td>
</tr>
<tr>
<td>Compare, Block</td>
<td>CMPBKB</td>
<td>SCAW</td>
</tr>
<tr>
<td>Compare, Block, Byte</td>
<td>EXT</td>
<td>CBW</td>
</tr>
<tr>
<td>Extend</td>
<td>EXTL</td>
<td>CWD</td>
</tr>
<tr>
<td>Extend, Long</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LOGICAL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>AND</td>
<td>AND</td>
</tr>
<tr>
<td>Or</td>
<td>OR</td>
<td>OR</td>
</tr>
<tr>
<td>Exclusive Or</td>
<td>XOR</td>
<td>XOR</td>
</tr>
<tr>
<td>Not</td>
<td>NOT</td>
<td>NOT</td>
</tr>
<tr>
<td>Shift Right</td>
<td>SHR</td>
<td>SHL, SAL</td>
</tr>
<tr>
<td>Shift Left</td>
<td>SHL</td>
<td>SAR</td>
</tr>
<tr>
<td>Shift Right Arithmetic</td>
<td>SHRA</td>
<td>ROR</td>
</tr>
<tr>
<td>Rotate Right</td>
<td>ROR</td>
<td>ROL</td>
</tr>
<tr>
<td>Rotate Left</td>
<td>ROL</td>
<td>RORC</td>
</tr>
<tr>
<td>Rotate Right Through Carry</td>
<td>ROLC</td>
<td>RCR</td>
</tr>
<tr>
<td>Rotate Left Through Carry</td>
<td>TEST</td>
<td>TEST</td>
</tr>
<tr>
<td>Test</td>
<td>NOTC</td>
<td>CMC</td>
</tr>
<tr>
<td>Not Carry</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DATA TRANSFER

Load

Load, Byte

Store

Store, Byte

Move

Move, Byte

Exchange

In

In, Byte

Out

Out, Byte

Clear Carry

Set Carry

Clear Direction

Set Direction

Break

Break on Overflow

Escape

Lock

LOAD

LD

MOV, LEA,
LES,
LODS,
LODW

LDB

MOV

LODB

ST

STOB

MOV,
LAHF

LODB

MOV

MOVB

XCH

XHC

IN

INW

INB

IN

OUT

OUTW

OUTB

OUT

CLRC

CLC

SETC

STC

CLRD

CLD

SETD

STD

BRK

INT

BRKV

INTO

ESC

ESC

LOCK

LOCK

BRANCH

Branch

Branch if Zero/Equal

Branch in Not Zero/Not Equal

Branch if Positive

Branch if Negative

Branch if Overflow

Branch if No Overflow

Branch if Greater Than

Branch if Greater Than or Equal

Branch if Less Than

Branch if Less Than or Equal

Branch if Higher

Branch if Not Higher

Branch if Lower

Branch if Not Lower

Branch if Parity Even

Branch if parity Odd

Branch if CX Zero

Decrement and Branch if
Not Zero

Decrement and Branch if
Not Zero and Equal

BR

BZ, BE

BNZ, NBE

BP

BN

BV

BNV

BFT

BHE

BLT

BLE

BH

BNH

BL

BNL

BPE

BPO

BCXZ

DBNZ

DBNZE

JMP

JZ, JE

JNZ, JNE

JS

JNS

JNO

JNO

JNLE/JG

JNL/JGE

JL/JNGE

JLE/JNG

JNB/JA

JBE/JNA

JB/JNAE

JNB/JAE

JNP/JPE

JNP/JPO

JCXZ

LOOP

LOOP/

LOOPE
Decrement and Branch if Not Zero and Not Equal  

**DBNZNE**  

**LOOPNZ/ LOOPNE**

**SUBROUTINE CALL**

**CALL**

**RETURN**

**RET**  

**RETI**

**IRET**

**MISCELLANEOUS**

**HALT**  

**HLT**

**WAIT**  

**WAIT**

**EI**  

**STI**

**DI**  

**CLI**

**ADJ4S**  

**DAS**

**ADJ4A**  

**DAA**

**ADJB**  

**AAA**

**ADJBA**  

**AAM**

**CVTBD**  

**AAD**

**CVTDB**  

**PUSH**

**PUSH, PUSHF**

**POP**  

**POP, POPF**

**REP**  

**REP**

**TR**  

**XLAT**
APPENDIX B

MATERIAL PROVIDED TO SUBJECTS

The purpose of this program is to locate the first occurrence of the ASCII letters "AB" in a memory block. The memory block starts at location 1000H and continues through and including 10FFH. The memory block is first searched for the character "A". When an "A" is found, the next location is compared with the letter "B". If a "B" is found in this location the address of the start of the "AB" character sequence is written into locations 1100H and 1101H (least significant byte first). The search is continued at most 255 times if no match is found.
Start
BC<--OFFH
HL<-1000H
Do While BC>0
A<--"A"
If A=M(HL)
   Then HL<--HL+1
   A<--"B"
   If A=M(HL)
      Then HL<--HL-1
      Exit Do
   End If
Else HL<--HL+1
End If
End Do
(1100)<--HL
Finish
The Z80 microprocessor contains three groups of registers. The first group consists of a set of 8 bit registers. The 8 bit registers (A, B, C, D, E, H, L) may be used individually or as 16-bit registers in pairs (BC, DE, HL). In addition, there is an 8-bit accumulator and a flag register.

The second group is an exact duplicate of the first. The alternate register set (A', B', C', D', E', H', L') and (B'C', D'E', H'L') is made available to the programmer via the "exchange" instruction group.

The third group of registers consists of two 16-bit index registers (IX and IY), the stack pointer (SP), the program counter (PC), as well as the interrupt vector (I) and the dynamic memory refresh register (R).
### MAIN REGISTER SET
- **A**: Accumulator
- **F**: Flag Register
- **B**: General Purpose
- **C**: General Purpose
- **D**: General Purpose
- **E**: General Purpose
- **H**: General Purpose
- **L**: General Purpose

<--------------------- 16 bits ---------------------->

### ALTERNATE REGISTER SET
- **A'**: Accumulator
- **F'**: Flag Register
- **B'**: General Purpose
- **C'**: General Purpose
- **D'**: General Purpose
- **E'**: General Purpose
- **H'**: General Purpose
- **L'**: General Purpose

<--------- 8 bits---------->

### Additional Registers
- **IX**: Index Register
- **IY**: Index Register
- **SP**: Stack Pointer
- **PC**: Program Counter
- **I**: Interrupt Vector
- **R**: Memory Refresh

<--------- 8 bits---------->
<table>
<thead>
<tr>
<th>REGISTER</th>
<th>SIZE (bits)</th>
<th>REMARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, A' Accumulator</td>
<td>8</td>
<td>Stores an operand or the results of an operation. See instruction set.</td>
</tr>
<tr>
<td>F, F' Flags</td>
<td>8</td>
<td>Can be used separately or as a 16-bit register with C. See B, above.</td>
</tr>
<tr>
<td>(PSW, PSW')</td>
<td></td>
<td>Can be used separately or as a 16-bit register with E. See D, above.</td>
</tr>
<tr>
<td>B, B' General purpose</td>
<td>8</td>
<td>Can be used separately or as a 16-bit register with L. See H, above.</td>
</tr>
<tr>
<td>C, C' General Purpose</td>
<td>8</td>
<td>Note: The (B,C), (D,E), and (H,L) sets are combined as follows: B - High byte C - Low byte D - High byte E - Low byte H - High byte L - Low byte Stores upper eight bits of memory address for vectored interrupt processing. Provides user-transparent dynamic memory refresh. Automatically incremented and placed on the address bus during each instruction fetch cycle.</td>
</tr>
<tr>
<td>D, D' General Purpose</td>
<td>8</td>
<td>Same as IX, above.</td>
</tr>
<tr>
<td>E, E' General Purpose</td>
<td>8</td>
<td>Stores addresses or data temporarily. See Push or Pop in instruction set. Holds address of next instruction.</td>
</tr>
<tr>
<td>H, H' General Purpose</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>L, L' General Purpose</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>I Interrupt Register</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>R Refresh Register</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>IX Index Register</td>
<td>16</td>
<td>Used for indexed addressing.</td>
</tr>
<tr>
<td>IY Index Register</td>
<td>16</td>
<td>Same as IX, above.</td>
</tr>
<tr>
<td>SP Stack Pointer</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>PC Program Counter</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
### ZILOG Z80 MNEMONICS

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r</code></td>
<td>one of the 8-bit registers A,B,C,D,E,H,L</td>
</tr>
<tr>
<td><code>n</code></td>
<td>any 8-bit absolute value</td>
</tr>
<tr>
<td><code>ii</code></td>
<td>an index register reference, either X or Y</td>
</tr>
<tr>
<td><code>d</code></td>
<td>an 8-bit index displacement, where <code>-128 &lt; d &lt; 127</code></td>
</tr>
<tr>
<td><code>zz</code></td>
<td>B for the BC register pair, D for the DE pair</td>
</tr>
<tr>
<td><code>nn</code></td>
<td>any 16-bit value, absolute or relocatable</td>
</tr>
<tr>
<td><code>rr</code></td>
<td>B for the BC register pair, D for the DE pair, H for the HL pair, SP for the stack pointer</td>
</tr>
<tr>
<td><code>qq</code></td>
<td>B for the BC register pair, D for the DE pair, H for the HL pair, PSW for the A/Flag pair.</td>
</tr>
<tr>
<td><code>s</code></td>
<td>any of r (defined above), M, or d(ii)</td>
</tr>
<tr>
<td><code>IFF</code></td>
<td>interrupt flip-flop</td>
</tr>
<tr>
<td><code>CY</code></td>
<td>carry flip-flop</td>
</tr>
<tr>
<td><code>ZF</code></td>
<td>zero flag</td>
</tr>
<tr>
<td><code>tt</code></td>
<td>B for the BC register pair, D for the DE pair, SP for the stack pointer, X for index register IX</td>
</tr>
<tr>
<td><code>uu</code></td>
<td>B for the BC register pair, D for the DE pair, SP for the stack pointer, Y for index register IY</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>b</code></td>
<td>a bit position in an 8-bit byte, where the bits are numbered from right to left 0 to 7.</td>
</tr>
<tr>
<td><code>PC</code></td>
<td>program counter</td>
</tr>
<tr>
<td><code>b{n}</code></td>
<td>bit <code>n</code> of the 8-bit value or register <code>v</code></td>
</tr>
<tr>
<td><code>vv/H</code></td>
<td>the most significant byte of the 16-bit value or register <code>vv</code></td>
</tr>
<tr>
<td><code>vv/L</code></td>
<td>the least significant byte of the 16-bit value or register <code>vv</code></td>
</tr>
<tr>
<td><code>Iv</code></td>
<td>an input operation on port <code>v</code></td>
</tr>
<tr>
<td><code>Ov</code></td>
<td>an output operation on port <code>v</code></td>
</tr>
<tr>
<td><code>w &lt;-- v</code></td>
<td>the value of <code>w</code> is replaced by the value of <code>v</code></td>
</tr>
<tr>
<td><code>w &lt;--&gt; v</code></td>
<td>the value of <code>w</code> is exchanged with the value of <code>v</code></td>
</tr>
</tbody>
</table>
# 8 BIT LOAD GROUP

<table>
<thead>
<tr>
<th>ZILOG MNEMONIC</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD r,r'</td>
<td>r &lt;-- r'</td>
</tr>
<tr>
<td>LD r,(HL)</td>
<td>r &lt;-- (HL)</td>
</tr>
<tr>
<td>LD r,(Iii + d)</td>
<td>r &lt;-- (Ii + d)</td>
</tr>
<tr>
<td>LD (HL),r</td>
<td>(HL) &lt;-- r</td>
</tr>
<tr>
<td>LD (Iii + d),r</td>
<td>(Ii + d) &lt;-- r</td>
</tr>
<tr>
<td>LD r,n</td>
<td>r &lt;-- n</td>
</tr>
<tr>
<td>LD (HL),n</td>
<td>(HL) &lt;-- n</td>
</tr>
<tr>
<td>LD (Iii + d),n</td>
<td>(Ii + d) &lt;-- n</td>
</tr>
<tr>
<td>LD A,(nn)</td>
<td>A &lt;-- (nn)</td>
</tr>
<tr>
<td>LD (nn),A</td>
<td>(nn) &lt;-- A</td>
</tr>
<tr>
<td>LD A,(zz)</td>
<td>A &lt;-- (zz)</td>
</tr>
<tr>
<td>LD (zz),A</td>
<td>(zz) &lt;-- A</td>
</tr>
<tr>
<td>LD A,I</td>
<td>A &lt;-- I</td>
</tr>
<tr>
<td>LD A,R</td>
<td>A &lt;-- R</td>
</tr>
<tr>
<td>LD I,A</td>
<td>I &lt;-- A</td>
</tr>
<tr>
<td>LD R,A</td>
<td>R &lt;-- A</td>
</tr>
</tbody>
</table>
### 16 BIT LOAD GROUP

#### Z80

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD rr,nn</td>
<td>rr &lt;-- nn</td>
</tr>
<tr>
<td>LD ii,nn</td>
<td>ii &lt;-- nn</td>
</tr>
<tr>
<td>LD BC,(nn)</td>
<td>B &lt;-- (nn + 1)</td>
</tr>
<tr>
<td></td>
<td>C &lt;-- (nn)</td>
</tr>
<tr>
<td>LD DE,(nn)</td>
<td>D &lt;-- (nn + 1)</td>
</tr>
<tr>
<td></td>
<td>E &lt;-- (nn)</td>
</tr>
<tr>
<td>LD HL,(nn)</td>
<td>H &lt;-- (nn + 1)</td>
</tr>
<tr>
<td></td>
<td>L &lt;-- (nn)</td>
</tr>
<tr>
<td>LD IX,(nn)</td>
<td>IX/H &lt;-- (nn + 1)</td>
</tr>
<tr>
<td></td>
<td>IX/L &lt;-- (nn)</td>
</tr>
<tr>
<td>LD IY,(nn)</td>
<td>IY/H &lt;-- (nn + 1)</td>
</tr>
<tr>
<td></td>
<td>IY/L &lt;-- (nn)</td>
</tr>
<tr>
<td>LD SP,(nn)</td>
<td>SP/H &lt;-- (nn + 1)</td>
</tr>
<tr>
<td></td>
<td>SP/L &lt;-- (nn)</td>
</tr>
<tr>
<td>LD (nn),BC</td>
<td>(nn + 1) &lt;-- B</td>
</tr>
<tr>
<td></td>
<td>(nn) &lt;-- C</td>
</tr>
<tr>
<td>LD (nn),DE</td>
<td>(nn + 1) &lt;-- D</td>
</tr>
<tr>
<td></td>
<td>(nn) &lt;-- E</td>
</tr>
<tr>
<td>LD (nn),HL</td>
<td>(nn + 1) &lt;-- H</td>
</tr>
<tr>
<td></td>
<td>(nn) &lt;-- L</td>
</tr>
<tr>
<td>LD (nn),IX</td>
<td>(nn + 1) &lt;-- IX/H</td>
</tr>
<tr>
<td></td>
<td>(nn) &lt;-- IX/L</td>
</tr>
<tr>
<td>LD (nn),IY</td>
<td>(nn + 1) &lt;-- IY/H</td>
</tr>
<tr>
<td></td>
<td>(nn) &lt;-- IY/L</td>
</tr>
<tr>
<td>LD (nn),SP</td>
<td>(nn + 1) &lt;-- SP/H</td>
</tr>
<tr>
<td></td>
<td>(nn) &lt;-- SP/L</td>
</tr>
<tr>
<td>LD SP,HL</td>
<td>SP &lt;-- HL</td>
</tr>
<tr>
<td>LD SP,IX</td>
<td>SP &lt;-- IX</td>
</tr>
<tr>
<td>LD SP,IY</td>
<td>SP &lt;-- IY</td>
</tr>
<tr>
<td>PUSH qq</td>
<td>(SP-1) &lt;-- qq/H</td>
</tr>
<tr>
<td></td>
<td>(SP-2) &lt;-- qq/L</td>
</tr>
<tr>
<td>PUSH ii</td>
<td>(SP-1) &lt;-- ii/H</td>
</tr>
<tr>
<td></td>
<td>(SP-2) &lt;-- ii/L</td>
</tr>
<tr>
<td>POP qq</td>
<td>qq/H &lt;-- (SP-1)</td>
</tr>
<tr>
<td></td>
<td>qq/L &lt;-- (SP)</td>
</tr>
<tr>
<td></td>
<td>SP &lt;-- SP-2</td>
</tr>
</tbody>
</table>
POP ii

\[ ii/H \leftarrow (SP + 1) \]
\[ ii/L \leftarrow (SP) \]
\[ SP \leftarrow SP + 2 \]
EXCHANGE, BLOCK TRANSFER, AND SEARCH GROUP

Z80

MNEMONIC

EX DE, HL
EX AF, AF'
EXX
EX (SP), HL
EX (SP), IX
EX (SP), IY
LDI

OPERATION

HL <--- DE
PSW <--- PSW'
BCDEHL <--- BCDEHL'
H <--- (SP + 1)
L <--- (SP)
IX/H <--- (SP + 1)
IX/L <--- (SP)
IX/H <--- (SP + 1)
IX/L <--- (SP)
(DE) <--- (HL)
DE <--- DE + 1
HL <--- HL + 1
BC <--- BC-1
repeat LDI until BC=0

LDIR

LDI

repeat LDD until BC=0

LDD

repeat CCI until A=(HL)

LDDR

CPI

repeat CCD until A=(HL)
or BC=0

CPIR

CPD

CPDR
8 BIT ARITHMETIC AND LOGICAL

Z80

MNEMONIC

ADD A,r
ADD A,(HL)
ADD A,(ii + d)
ADD A,n
ADC A,s
ADC A,n
SUB s
SUB n
SBC A,s
SBC A,n
AND s
AND n
OR s
OR n
XOR s
XOR n
CP s
CP n
INC r
INC (HL)
INC (ii + d)
DEC r
DEC (HL)
DEC (ii + d)

OPERATION

A <-- A + r
A <-- A + (HL)
A <-- A + (ii + d)
A <-- A + n
A <-- A + s + CY
A <-- A + n + CY
A <-- A - s
A <-- A - n
A <-- A - s - CY
A <-- A - n - CY
A <-- A ^ s
A <-- A ^ n
A <-- A v s
A <-- A v n
A <-- A + s
A <-- A + n
A - s
A - n
r <-- r + 1
(HL) <-- (HL) + 1
(ii + d) <-- (ii + d) + 1
r <-- r - 1
(HL) <-- (HL) - 1
(kk + d) <-- (ii + d) - 1
GENERAL PURPOSE ARITHMETIC AND CONTROL GROUP

Z80
MNEMONIC

DAA

OPERATION

convert A to packed BCD
after an add or subtract
of packed BCD operands

A <-- ~A^A
A <-- -A
CY <-- ~CY
CY <-- 1

no operation
halt

IFF <-- 0
IFF <-- 1

interrupt mode 0
interrupt mode 1
interrupt mode 2
16 BIT ARITHMETIC GROUP

Z80

MNEMONIC | OPERATION
----------|------------------
ADD HL,rr | HL <-- HL + rr
ADC HL,rr | HL <-- HL + rr + CY
SBC HL,rr | HL <-- HL - rr - CY
ADD IX,tt | IX <-- IX + tt
ADD IY,uu | IY <-- IY + uu
INC rr    | rr <-- rr + 1
INC ii    | ii <-- ii + 1
DEC rr    | rr <-- rr - 1
DEC ii    | ii <-- ii - 1
### Z80

#### Mnemonic

<table>
<thead>
<tr>
<th></th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLCA</td>
<td>CY &lt;-- 7 &lt;-- 0 &lt;-- A</td>
</tr>
<tr>
<td>RLA</td>
<td>CY &lt;-- 7 &lt;-- 0 &lt;-- A</td>
</tr>
<tr>
<td>RRCA</td>
<td>7 --&gt; 0 --&gt; CY A</td>
</tr>
<tr>
<td>RRA</td>
<td>7 --&gt; 0 --&gt; CY A</td>
</tr>
<tr>
<td>RLC r</td>
<td>Same diagram as for RLC</td>
</tr>
<tr>
<td>RLC (HL)</td>
<td>Same diagram as for RLC</td>
</tr>
<tr>
<td>RLC (i+i + d)</td>
<td>Same diagram as for RLC</td>
</tr>
<tr>
<td>RL S</td>
<td>Same diagram as for RAL</td>
</tr>
<tr>
<td>RRC s</td>
<td>Same diagram as for RRC</td>
</tr>
<tr>
<td>RR s</td>
<td>Same diagram as for RAR</td>
</tr>
<tr>
<td>SLA s</td>
<td>CY &lt;-- 7 &lt;-- 0 &lt;-- 0 s</td>
</tr>
<tr>
<td>SRA s</td>
<td>7 --&gt; 0 --&gt; CY s</td>
</tr>
<tr>
<td>SRL s</td>
<td>0 --&gt; 7 --&gt; 0 --&gt; CY s</td>
</tr>
<tr>
<td>RLD</td>
<td>A 7 4 3 0</td>
</tr>
<tr>
<td>RRD</td>
<td>A 7 4 3 0</td>
</tr>
</tbody>
</table>
BIT SET, RESET, AND TEST GROUP

Z80

MNEMONIC

BIT b,r
BIT b,(HL)
Bit b,(Iii + d)
SET b,r
SET b,(HL)
SET b,(Iii + d)
RES b,s

OPERATION

ZF <-- \sim r\sim r\{b\}
ZF <-- \sim (HL)\{b\}
ZF <-- (\sim (Iii + d)\{b\})
r\{b\} <-- 1
(\text{HL})\{b\} <-- 1
(Iii + d )\{b\} <-- 1
S\{b\} <-- 0
Z80
MNEMONIC

JP nn
JP Z,nn

JP NZ,nn
JP C,nn
JP NC,nn
JP PO,nn
JP PE,nn
JP P,nn
JP M,nn
JP PE,nn
JP PO,nn
JR e

JR Z,e
JR NZ,e
JR C,e
JRNC,en
DJNZ e

JP (HL)
JP (IX)
JP (IY)

OPERATION

PC <-- nn
if zero, then JMP
else continue
if not zero
if carry
if not carry
if parity odd
if parity even
if sign positive
if sign negative
if overflow
if no overflow
PC <-- PC + e
where e=nn - PC
-126< e < 129
if zero, then JMPR
else continue
if not zero
if carry
if not carry
B <-- B - 1
if B=0 then continue
else JMPR
PC <-- HL
PC <-- IX
PC <-- IY
CALL AND RETURN GROUP

Z80
MNEMONIC

CALL nn

CALL Z,nn

CALL NZ,nn
CALL C,nn
CALL NC,nn
CALL PO,nn
CALL PE,nn
CALL P,nn
CALL M,nn
CALL PE,nn
CALL PO,nn
RET

RET Z

RET NZ
TER C
RET NC
RET PO
RET PE
RET P
RET M
RET PE
RET PO
RETI
RETN

RST n

OPERATION

(SP-1) <-- PC/H
(SP-2) <-- PC/L
SP <-- SP-2
PC <-- nn
if zero, then CALL
else continue
if not zero
if carry
if not carry
if parity odd
if parity even
if sign positive
if sign negative
if overflow
if no overflow
PC/H <-- (SP + 1)
PC/L <-- (SP)
SP <-- SP + 2
if zero, then RET
else continue
if not zero
if carry
if not carry
if parity odd
if parity even
if sign positive
if sign negative
if overflow
if no overflow
return from interrupt
return from non-maskable interrupt
(SP-1) <-- PC/H
(SP-2) <-- PC/L
PC <-- 8 * n
where 0 < n < 8
INPUT AND OUTPUT GROUP

Z80
MNEMONIC

IN A, (n)
IN r, (C)
INI

INIR
IND
INDR
OUT (n), A
OUT (C), r
OUTI

OTIR
OUTD
OUDR

OPERATION

A <-- In
r <-- I(C)
(HL) <-- I(C)
B <-- B - 1
HL <-- HL + 1
repeat INI until B=0

(HL) <-- I(C)
B <-- B - 1
HL <-- HL - 1
repeat IND until B=0

On <-- A
O(C) <-- r
O(C) <-- (HL)
B <-- B - 1
HL <-- HL + 1
repeat OUTI until B=0

O(C) <-- (HL)
B <-- B - 1
HL <-- HL - 1
repeat OUTD until B=0
# TDL Z80 Mnemonics

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>one of the 8-bit registers A, B, C, D, E, H, L</td>
</tr>
<tr>
<td>n</td>
<td>any 8-bit absolute value</td>
</tr>
<tr>
<td>ii</td>
<td>an index register reference, either X or Y</td>
</tr>
<tr>
<td>d</td>
<td>an 8-bit index displacement, where -128 &lt; d &lt; 127</td>
</tr>
<tr>
<td>zz</td>
<td>B for the BC register pair, D for the DE pair</td>
</tr>
<tr>
<td>nn</td>
<td>any 16-bit value, absolute or relocatable</td>
</tr>
<tr>
<td>rr</td>
<td>B for the BC register pair, D for the DE pair, H for the HL pair, SP for the stack pointer</td>
</tr>
<tr>
<td>qq</td>
<td>B for the BC register pair, D for the DE pair, H for the HL pair, PSW for the A/Flag pair.</td>
</tr>
<tr>
<td>s</td>
<td>any of r (defined above), M, or d(ii)</td>
</tr>
<tr>
<td>IFF</td>
<td>interrupt flip-flop</td>
</tr>
<tr>
<td>CY</td>
<td>carry flip-flop</td>
</tr>
<tr>
<td>ZF</td>
<td>zero flag</td>
</tr>
<tr>
<td>tt</td>
<td>B for the BC register pair, D for the DE pair, SP for the stack pointer, X for index register IX</td>
</tr>
<tr>
<td>uu</td>
<td>B for the BC register pair, D for the DE pair, SP for the stack pointer, Y for index register IY</td>
</tr>
</tbody>
</table>
\textbf{b} \quad \text{a bit position in an 8-bit byte, where the bits are numbered from right to left 0 to 7.}

\textbf{PC} \quad \text{program counter}

\textbf{b\{n\}} \quad \text{bit n of the 8-bit value or register v}

\textbf{vv/H} \quad \text{the most significant byte of the 16-bit value or register vv}

\textbf{vv/L} \quad \text{the least significant byte of the 16-bit value or register vv}

\textbf{Iv} \quad \text{an input operation on port v}

\textbf{Ov} \quad \text{an output operation on port v}

\textbf{w \leftarrow v} \quad \text{the value of w is replaced by the value of v}

\textbf{w \leftrightarrow v} \quad \text{the value of w is exchanged with the value of v}
8 BIT LOAD GROUP

TDL
MNEMONIC

MOV r,r'
MOV r,M
MOV r,d(ii)
MOV M,r
MOV d(ii),r
MVI r,n
MVI M,n
MVI d(ii), n
LDA nn
STA nn
LDAX zz
STAX zz
LDAI
LDAR
STAI
STAR

OPERATION

r <-- r'
r <-- (HL)
r <-- (ii + d)
(II) <-- r
(ii + d) <-- r
r <-- n
(II) <-- n
(ii + d) <-- n
A <-- (nn)
A <-- (zz)
(zz) <-- A
A <-- I
A <-- R
I <-- A
R <-- A
16 BIT LOAD GROUP

TDL
MNEMONIC
OPERATION

LXI rr, nn  rr <-- nn
LXI ii, nn   ii <-- nn
LBCCD nn    B <-- (nn + 1)
            C <-- (nn)
LDEED nn    D <-- (nn + 1)
            E <-- (nn)
LHLDD nn    H <-- (nn + 1)
            L <-- (nn)
LIXDD nn    IX/H <-- (nn + 1)
            IX/L <-- (nn)
LIYDD nn    IY/H <-- (nn + 1)
            IY/L <-- (nn)
LSPDD nn    SP/H <-- (nn + 1)
            SP/L <-- (nn)
SBCCD nn    (nn + 1) <-- B
            (nn)   <-- C
SDEED nn    (nn + 1) <-- D
            (nn)   <-- E
SHLDD nn    (nn + 1) <-- H
            (nn)   <-- L
SIXD nn     (nn + 1) <-- IX/H
            (nn)   <-- IX/L
SIYDD nn    (nn + 1) <-- IY/H
            (nn)   <-- IY/L
SSPDD nn    (nn + 1) <-- SP/H
            (nn)   <-- SP/L
SPHL        SP <-- HL
SPIX        SP <-- IX
SPIY        SP <-- IY
PUSH qq     (SP-1) <-- qq/H
            (SP-2) <-- qq/L
            SP <-- SP-2
PUSH ii     (SP-1) <-- ii/H
            (SP-2) <-- ii/L
            SP <-- SP-2
POP qq      qq/H <-- (SP-1)
            qq/L <-- (SP)
            SP <-- SP-2
POP ii

ii/H <-- (SP + 1)
ii/L <-- (SP)
SP   <-- SP + 2
EXCHANGE, BLOCK TRANSFER, AND SEARCH GROUP

<table>
<thead>
<tr>
<th>TDL</th>
<th>MNEMONIC</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCHG</td>
<td></td>
<td>HL &lt;-&gt; DE</td>
</tr>
<tr>
<td></td>
<td>EXAF</td>
<td>PSW &lt;-&gt; PSW'</td>
</tr>
<tr>
<td>EXX</td>
<td></td>
<td>BCDEHL &lt;-&gt; BCDEHL'</td>
</tr>
<tr>
<td>XTHL</td>
<td></td>
<td>H &lt;-&gt; (SP + 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L &lt;-&gt; (SP)</td>
</tr>
<tr>
<td>XTIX</td>
<td></td>
<td>IX/H &lt;-&gt; (IX + 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IX/L &lt;-&gt; (IX)</td>
</tr>
<tr>
<td>XTIY</td>
<td></td>
<td>IY/H &lt;-&gt; (SP + 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IY/L &lt;-&gt; (SP)</td>
</tr>
<tr>
<td>LDI</td>
<td></td>
<td>(DE) &lt;-&gt; (HL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DE &lt;-&gt; DE + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HL &lt;-&gt; HL + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BC &lt;-&gt; BC - 1</td>
</tr>
<tr>
<td>LDIR</td>
<td></td>
<td>repeat LDI until BC = 0</td>
</tr>
<tr>
<td>LDD</td>
<td></td>
<td>(DE) &lt;-&gt; (HL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DE &lt;-&gt; DE - 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HL &lt;-&gt; HL - 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BC &lt;-&gt; BC - 1</td>
</tr>
<tr>
<td>LDDR</td>
<td></td>
<td>repeat LDD until BC = 0</td>
</tr>
<tr>
<td>CCI</td>
<td></td>
<td>A - (HL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HL &lt;-&gt; HL + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BC &lt;-&gt; BC - 1</td>
</tr>
<tr>
<td>CCIR</td>
<td></td>
<td>repeat CCI until A = (HL)</td>
</tr>
<tr>
<td>CCD</td>
<td></td>
<td>A - (HL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HL &lt;-&gt; HL - 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BC &lt;-&gt; BC - 1</td>
</tr>
<tr>
<td>CCDR</td>
<td></td>
<td>repeat CCD until A = (HL) or BC = 0</td>
</tr>
</tbody>
</table>
8 BIT ARITHMETIC AND LOGICAL

TDL
MNEMONIC

ADD r
ADD M
ADD d(ii)
ADI n
ADC s
ACI n
SUB s
SUI n
SBB s
SBI n
ANA s
ANI n
ORA s
ORI n
XRA s
XRI n
CMP s
CPI n
INR r
INR M
INR d(ii)
DCR r
DCM M
DCR d(ii)

OPERATION

A <-- A + r
A <-- A + (HL)
A <-- A + (ii + d)
A <-- A + n
A <-- A + s + CY
A <-- A + n + CY
A <-- A - s
A <-- A - n
A <-- A - s - CY
A <-- A - n - CY
A <-- A ^ n
A <-- A ^ s
A <-- A ^ v s
A <-- A ^ v n
A <-- A + s
A <-- A + n
A <-- A - s
A <-- (HL) + l
(III + d) <-- (III + d) + l
r <-- r - 1
(III) <-- (HL) - 1
(kk + d) <-- (III + d) - 1
GENERAL PURPOSE ARITHMETIC AND CONTROL GROUP

TDL
MNEMONIC
DAA

OPERATION
convert A to packed BCD
after an add or subtract
of packed BCD operands
A <-- ~A
A <-- -A
CY <-- ~CY
CY <-- 1
no operation
halt
IFF <-- 0
IFF <-- 1
interrupt mode 0
interrupt mode 1
interrupt mode 2
16 BIT ARITHMETIC GROUP

<table>
<thead>
<tr>
<th>TDL</th>
<th>MNEMONIC</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAD</td>
<td>rr</td>
<td>HL &lt;-- HL + rr</td>
</tr>
<tr>
<td>DADC</td>
<td>rr</td>
<td>HL &lt;-- HL + rr + CY</td>
</tr>
<tr>
<td>DSBC</td>
<td>rr</td>
<td>HL &lt;-- HL - rr - CY</td>
</tr>
<tr>
<td>DADX</td>
<td>tt</td>
<td>IX &lt;-- IX + tt</td>
</tr>
<tr>
<td>DADY</td>
<td>uu</td>
<td>IY &lt;-- IY + uu</td>
</tr>
<tr>
<td>INX</td>
<td>rr</td>
<td>rr &lt;-- rr + 1</td>
</tr>
<tr>
<td>INX</td>
<td>ii</td>
<td>ii &lt;-- ii + 1</td>
</tr>
<tr>
<td>DCX</td>
<td>rr</td>
<td>rr &lt;-- rr - 1</td>
</tr>
<tr>
<td>DCX</td>
<td>ii</td>
<td>ii &lt;-- ii - 1</td>
</tr>
</tbody>
</table>
**ROTATE AND SHIFT GROUP**

<table>
<thead>
<tr>
<th>TDL MNEMONIC</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLC</td>
<td>CY &lt;-- 7 &lt;-- 0 &lt;-- A</td>
</tr>
<tr>
<td>RAL</td>
<td>CY &lt;-- 7 &lt;-- 0 &lt;-- A</td>
</tr>
<tr>
<td>RRC</td>
<td>7 --&gt; 0 --&gt; CY A</td>
</tr>
<tr>
<td>RAR</td>
<td>7 --&gt; 0 --&gt; CY A</td>
</tr>
<tr>
<td>RLCR r</td>
<td>Same diagram as for RLC</td>
</tr>
<tr>
<td>RLCR M</td>
<td>Same diagram as for RLC</td>
</tr>
<tr>
<td>RLCR d(ii)</td>
<td>Same diagram as for RLC</td>
</tr>
<tr>
<td>RALR s</td>
<td>Same diagram as for RAL</td>
</tr>
<tr>
<td>RRCR s</td>
<td>Same diagram as for RRC</td>
</tr>
<tr>
<td>RARR s</td>
<td>Same diagram as for RAR</td>
</tr>
<tr>
<td>SLAR s</td>
<td>CY &lt;-- 7 &lt;-- 0 &lt;-- 0 s</td>
</tr>
<tr>
<td>SRAR s</td>
<td>7 --&gt; 0 --&gt; CY s</td>
</tr>
<tr>
<td>SRLR s</td>
<td>0 --&gt; 7 --&gt; 0 --&gt; CY s</td>
</tr>
<tr>
<td>RLD</td>
<td>A 7 4 3 0</td>
</tr>
<tr>
<td></td>
<td>(HL) 7 4 3 0</td>
</tr>
<tr>
<td>RRD</td>
<td>A 7 4 3 0</td>
</tr>
<tr>
<td></td>
<td>(HL) 7 4 3 0</td>
</tr>
</tbody>
</table>
BIT SET, RESET, AND TEST GROUP

TDL

MNEMONIC

BIT b,r
BIT b,M
BIT b,d(ii)
SET b,r
SET b,m
SET b,d(ii)
RES b,s

OPERATION

ZF <-- ~r{b}
ZF <-- ~(HL){b}
ZF <-- ~(Iii + d){b}
r{b} <-- 1
(HL){b} <-- 1
(Iii + d){b} <-- 1
S{b} <-- 0
JUMP GROUP

TDL

MNEMONIC

JMP nn
JZ nn
JNZ nn
JC nn
JNC nn
JPO nn
JPE nn
JP nn
JM nn
JO nn
JNO nn
JMPR nn
JRZ nn
JRNZ nn
JRC nn
JRNC nn
DJNZ nn

PCHL
PCIX
PCIY

OPERATION

PC <-- nn
if zero, then JMP
else continue
if not zero
if carry
if not carry
if parity odd
if parity even
if sign positive
if sign negative
if overflow
if no overflow
PC <-- PC + e
where e=nn - PC
-126< e < 129
if zero, then JMPR
else continue
if not zero
if carry
if not carry
B <-- B - 1
if B=0 then continue
else JMPR
PC <-- HL
PC <-- IX
PC <-- IY
# CALL AND RETURN GROUP

<table>
<thead>
<tr>
<th>TDL MNEMONIC</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL nn</td>
<td>(SP-1) (\leftarrow) PC/H&lt;br&gt;(SP-2) (\leftarrow) PC/L&lt;br&gt;SP (\leftarrow) SP-2&lt;br&gt;PC (\leftarrow) nn&lt;br&gt;if zero, then CALL&lt;br&gt;else continue</td>
</tr>
<tr>
<td>CZ</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>CNZnn</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>CC nn</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>CNC nn</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>CPO nn</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>CPE nn</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>CP nn</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>CM nn</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>CO nn</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>CNO nn</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RET</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RZ</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RNZ</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RC</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RNC</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RPO</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RPE</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RP</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RM</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RO</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RNO</td>
<td>if not zero&lt;br&gt;if carry&lt;br&gt;if not carry&lt;br&gt;if parity odd&lt;br&gt;if parity even&lt;br&gt;if sign positive&lt;br&gt;if sign negative&lt;br&gt;if overflow&lt;br&gt;if no overflow&lt;br&gt;PC/H (\leftarrow) (SP + 1)&lt;br&gt;PC/L (\leftarrow) (SP)&lt;br&gt;SP (\leftarrow) SP + 2&lt;br&gt;if zero, then RET&lt;br&gt;else continue</td>
</tr>
<tr>
<td>RETI</td>
<td>return from interrupt&lt;br&gt;return from non-maskable interrupt&lt;br&gt;(SP-1) (\leftarrow) PC/H&lt;br&gt;(SP-2) (\leftarrow) PC/L&lt;br&gt;PC (\leftarrow) 8 * n&lt;br&gt;where 0 &lt; n &lt; 8</td>
</tr>
</tbody>
</table>
INPUT AND OUTPUT GROUP

TDL
MNEMONIC

IN n
INP r
INI

INIR
IND

INDR
OUT n
OUTP r
OUTI

OUTIR
OUTD

OUTDR

OPERATION

A <-- In
r <-- I(C)
(HL) <-- I(C)
B <-- B - 1
HL <-- HL + 1
repeat INI until B=0

(HL) <-- I(C)
B <-- B - 1
HL <-- HL - 1
repeat IND until B=0

On <-- A
O(C) <-- r
O(C) <-- (HL)
B <-- B - 1
HL <-- HL + 1
repeat OUTI until B=0

O(C) <-- (HL)
B <-- B - 1
HL <-- HL - 1
repeat OUTD until B=0
BIBLIOGRAPHY


