1984

Pilot Study of Applicability of a Generic Microprocessor Assembly Language

Joseph H. Bartlett
University of Central Florida

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

Chapter
I. INTRODUCTION 1
II. THE PROBLEM 7
III. STATISTICS 9
IV. EQUIPMENT. 13
V. SUBJECTS 15
VI. EXPERIMENTAL METHODOLOGY 16
VII. RESULTS. 19
VIII. DISCUSSION OF RESULTS. 22
Appendices
A. 28
B. 56
BIBLIOGRAPHY 89

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 similiar 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.

H_1: 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; 2 Generic group

N_n = Number of subjects for group n

t_{in} = Time of successful completion for the ith 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:

$$t = \frac{t_{m1} - t_{m2}}{\sqrt{\frac{S^2}{N_1} + \frac{S^2}{N_2}}}$$

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 ascertain 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.

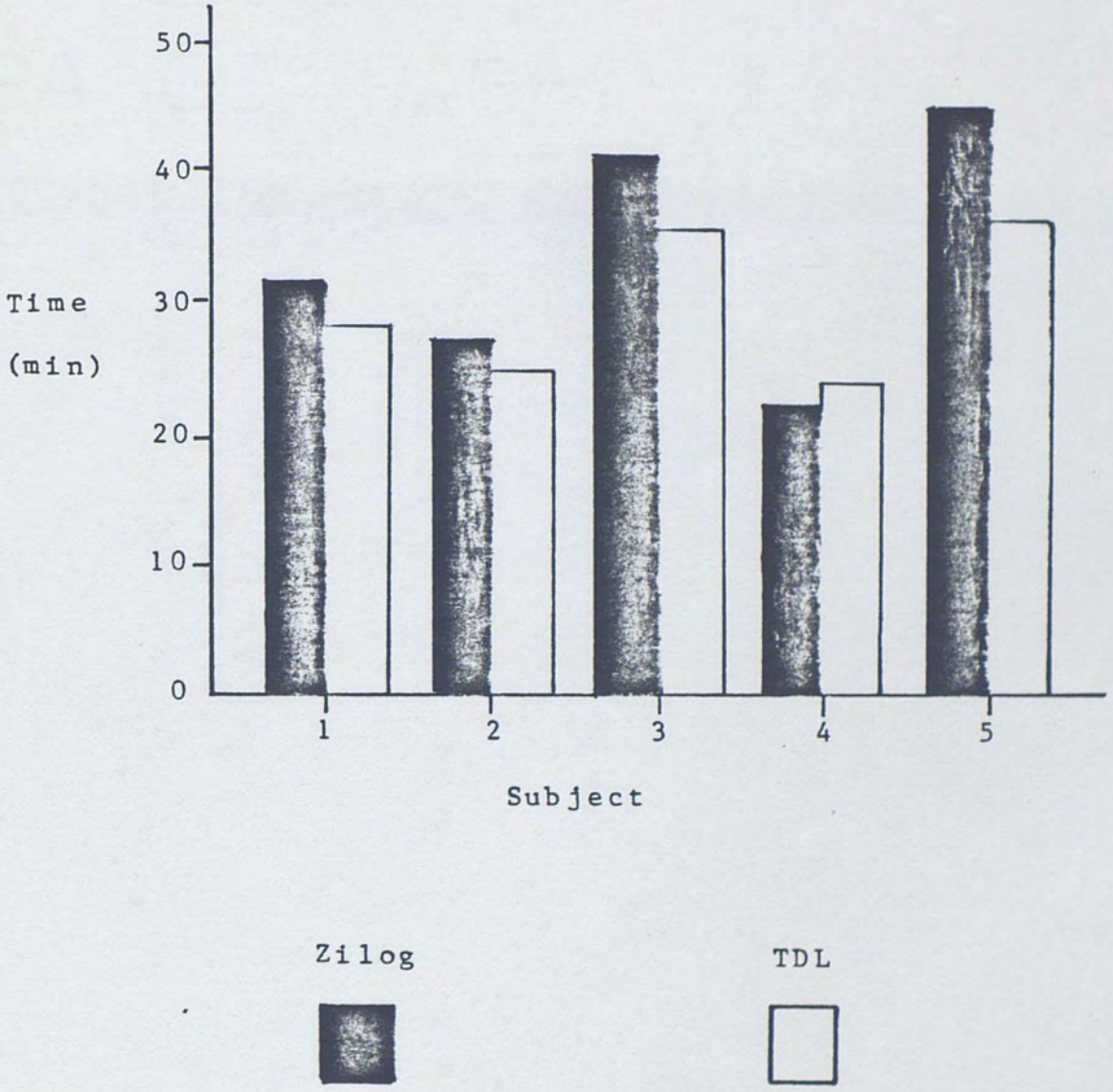
CHAPTER VII

RESULTS

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

SUBJECT	TIME (MINUTES) ZILOG MNEMONICS	TDL MNEMONICS
1	31	28
2	27	25
3	42	35
4	22	23
5	45	37
AVERAGE STD DEV	33.4 9.8	29.6 6.1

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 attributible 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 (α) and II (β) errors. This number is determined from a table of N values versus α, β, σ and δ (δ 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.

APPENDICES

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):

B: Byte
H: Halfword
L: Long (Double Word)
D: Decimal
F: Floating Point
1: Bit
4: Nibble or Digit
M: Multiple

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 conditioon 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 intruction 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 exectued 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 exist. 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 plaes 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 momory 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 Thab 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)

17. Skip If Not Higher (SKNH)

18. Skip If Lower (SKL)

19. Skip If Not Lower (SKNL)

20. Skip If Parity Even (SKPE)

21. Skip If Parity Odd (SKPO)

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 (RETE)

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. Retrun If Greater Than (RETGT)

13. Return If Greater Than Or Equal (RETGE)

14. Return If Less Than (RETLT)

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 transfered 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:

MODE	SYMBOL	EXAMPLE
Absolute	prefix /	/addr
Base page	prefix !	!addr
Indirect	prefix @	@addr
Relative	prefix $	$addr
Immediate	prefix #	#value
Index	enclosing parenthesis ()	addr(index)
Register	prefix.	.addr
Auto-pre-increment	prefix +	+addr
Auto-post-increment	postfix +	addr+
Auto-pre-decrement	prefix -	-addr
Auto-post-decrement	postfix -	addr-
Indirect-pre-indexed	prefix () @	addr(index)@
Indirect-post-indexed	prefix @, postfix ()	@addr(index)

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:

SYMBOL(S)	OPERATION
+	Add
-	Subtract
*	Multiply
/	Divide (Signed)
/ /	Divide (Unsigned)
.AND.	AND
.OR.	OR
.XOR.	Exclusive OR
.NOT.	NOT
.SHL.	Left Shift
.SHR.	Right Shift

SYMBOL(S)	OPERATION
.MOD.	Modulo
**	Exponentiate
<:>	Bit Alignment

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

INSTRUCTION	STANDARD MNEMONIC	MOTOROLA MNEMONIC
ARITHMETIC		
Add	ADD	ADD,ABA
Add with Carry	ADDC	ADC
Subtract	SUB	SUB,SBA
Increment	INC	INC,INS, INX
Decrement	DEC	DEC,DES, DEX
Compare	CMP	CMP,CBA, CPX
Negate	NEG	NEG
LOGICAL		
And	AND	AND
Or	OR	ORA
Exclusive Or	XOR	EOR
Not	NOT	COM
Shift Right	SHR	LSR
Shift Left	SHL	ASL
Shift Right Arithmetic	SHRA	ASR
Rotate Right	ROR	ROR
Rotate Left	ROL	ROL
Test	TEST	BIT,TST
DATA TRANSFER		
Load	LD	LDA,LDS, LDX
Store	ST	STA,STS, STX
Move	MOV	TAB,TBA, TAP,TPA, TSX,TXS
Clear	CLR	CLR
Clear Carry	CLRC	CLC

Clear Overflow	CLRV	CLV
Set Carry	SETC	SEC
Set Overflow	SETV	SEV

BRANCH

Branch	BR	BRA,JMP
Branch if Zero	BZ	BEQ
Branch if Not Zero	BNZ	BNE
Branch if Equal	BE	BEQ
Branch in Not Equal	BNE	BNE
Branch if Carry	BC	BCS
Branch if Positive	BP	BPL
Branch if Negative	BN	BMI
Branch if Overflow	BV	BVS
Branch if No Overflow	BNV	BVC
Branch if Greater Than	BGT	BGT
Branch if Greater Than or Equa	BGE	BGE
Branch if Less Than	BLT	BLT
Branch if Higher	BH	BHI
Branch in Not Higher	BNH	BLS
Branch if Lower	BL	BCS
Branch if Not Lower	BNL	BCC

SUBROUTINE CALL

Call Subroutine	CALL	BSR,JSR

RETURN

Return from Subroutine	RET	RTS
Returnfrom Interrupt	RETI	RTI

MISCELLANEOUS

No Operation	NOP	NOP
Push	PUSH	PSH
Pop	POP	PUL
Wait	WAIT	WAI
Adjust Decimal	ADJ	DAA
Enable Interrupt	EI	SEI
Disable Interrupt	DI	CLI
Break	BRK	SWI

STANDARD INSTRUCTION MNEMONICS FOR Z80, 8080,8085

INSTRUCTION	STANDARD MNEMONIC	ZILOG MNEMONIC	INTEL MNEMONIC
ARITHMETIC			
Add	ADD	ADD	ADD,ADI, DAD
Add with Carry	ADDC	ADC	ADC,ACI
Subtract	SUB	SUB	SUB,SUI
Subtract with Carry	SUBC	SBC	SBB,SBI
Increment	INC	INC	INX,INR
Decrement	DEC	DEC	DCX,DCR
Compare	CMP	CP,CPI, CPD	CMP,CPI
Compare, Multiple	CMPM	CPIR,CPDR	-
Negate	NEG	NEG	-
LOGICAL			
And	AND	AND	ANA,ANI
Or	OR	OR	ORA,ORI
Exclusive Or	XOR	XOR	XRA,XRI
Not	NOT	CPL	CMA
Not Carry	NOTC	CCF	CMC
Shift Right	SHR	SRL	-
Shift Left	SHL	SLA	ADD,DAD
Shift Right Arithmetic	SHRA	SRA	-
Rotate Right	ROR	RRCA,RRC	RAR
Rotate Left	ROL	RLCA,RLC	RAL
Rotate Right Through Carry	RORC	RR,RRA	RRC
Rotate Left Through Carry	ROLC	RL,RLA	RLC
Rotate Right Decimal	ROR4	RLD	-
Rotate Left Decimal	ROL4	RLD	-
Test Bit	TEST1	BIT	-

DATA TRANSFER			
Load	LD	LD	MOV,LXI, LHLD, LDA,MVI
Store	ST	LD	MOV,STAX, SHLD, STA
Move	MOV	LD,LDI, LDD	MOV,MVI, SPHL
Move Block	MOVBK	LDIR,LDDR	-
Exchange	XCH	EX,EXX	XCHG, XTHL
Input	IN	IN,INI, IND	IN,RIM
Input Block	INBK	INIR,INDR	-
Output	OUT	OUT,OUTI, OUTD	OUT,SIM
Output Block	OUTBK	OTIR,OTDR	-
Set Bit	SET1	DET	-
Clear Bit	CLR1	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, JR NZ	JNZ
Branch if Equal	BE	JP Z,JR Z	JZ
Branch if Not Equal	BNE	JP NZ, JR NZ	JNZ
Branch if Carry	BC	JP C,JR C	-
Branch if No Carry	BNC	JP NC, JR NC	JNC
Branch if Positive	BP	JP P	JP
Branch if Negative	BN	JP M	JM
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	CALL,RST	CALL,RST
Call if Zero	CALLZ	CALL Z	CZ
Call if Not Zero	CALLNZ	CALL NZ	CNZ
Call if Equal	CALLE	CALL Z	CZ

Call if Not Equal	CALLNE	CALL NZ	CNZ
Call if Carry	CALLC	CALL N	CC
Call if No Carry	CALLNC	CALL NC	CNC
Call if Positive	CALLP	CALL P	CP
Call if Negative	CALLN	CALL M	CM
Call if Parity Even	CALLPE	CALL PE	CPE
Call if Parity Odd	CALLPO	CALL PO	CPO
Call if Low	CALLL	CALL C	CC
Call if Not Low	CALLNL	CALL NC	CNC

RETURN

Return	RET	RET	RET
Return if Zero	RETZ	RET Z	RZ
Return if Not Zero	RETNZ	RET NZ	RNZ
Return if Equal	RETE	RET Z	RZ
Return if Not Equal	RETNE	RET NZ	RNZ
Return if Carry	RETC	RET C	RC
Return if No Carry	RETNC	RET NC	RNC
Return if Positive	RETP	RET P	RP
Return if Negative	RETN	RET M	RM
Return if Parity Even	RETPE	RET PE	RPE
Return if Parity Odd	RETPO	RET PO	RPO
Return if Lower	RETL	RET C	RC
Return if Not Lower	RETNL	RET NC	RNC
Return from Interrupt	RETI	RETI	-
Return from Interrupt Non-Maskable	RETIN	RETN	-

MISCELLANEOUS

No operation	NOP	NOP	NOP
Push	PUSH	PUSH	PUSH
Pop	POP	POP	POP
Wait	WAIT	HALT	HLT
Adjust Decimal	ADJ	DAA	DAA
Enable Interrupt	EI	EI	EI
Disable Interrupt	DI	DI	DI

IEEE STANDARD INSTRUCTION MENEMONICS FOR THE 8086

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

DATA TRANSFER

Load	LD	MOV, LEA, LES, LODS, LODW
Load, Byte	LDB	LODB
Store	ST	MOV
Store, Byte	STB	STOB
Move	MOV	MOV,LAHF
Move, Byte	MOVB	MOVB
Exchange	XCH	XHCG
In	IN	INW
In, Byte	INB	IN
Out	OUT	OUTW
Out, Byte	OUTB	OUT
Clear Carry	CLRC	CLC
Set Carry	SETC	STC
Clear Direction	CLRD	CLD
Set Direction	SETD	STD
Break	BRK	INT
Break on Overflow	BRKV	INTO
Escape	ESC	ESC
Lock	LOCK	LOCK

BRANCH

Branch	BR	JMP
Branch if Zero/Equal	BZ,BE	JZ,JE
Branch in Not Zero/Not Equal	BNZ,NBE	JNZ,JNE
Branch if Positive	BP	JS
Branch if Negative	BN	JNS
Branch if Overflow	BV	JNO
Branch if No Overflow	BNV	JNO
Branch if Greater Than	BFT	JNLE/JG
Branch if Greater Than or Equal	BHE	JNL/JGE
Branch if Less Than	BLT	JL/JNGE
Branch if Less Than or Equal	BLE	JLE/JNG
Branch if Higher	BH	JNBE/JA
Branch if Not Higher	BNH	JBE/JNA
Branch if Lower	BL	JB/JNAE
Branch if Not Lower	BNL	JNB/JAE
Branch if Parity Even	BPE	JNP/JPE
Branch if parity Odd	BPO	JNP/JPO
Branch if CX Zero	BCXZ	JCXZ
Decrement and Branch if Not Zero	DBNZ	LOOP
Decrement and Branch if Not Zero and Equal	DBNZE	LOOP/ LOOPE

Decrement and Branch if Not Zero and Not Equal	DBNZNE	LOOPNZ/ LOOPNE
SUBROUTINE CALL		
Call	CALL	CALL
RETURN		
Return	RET	RET
Return from Interrupt	RETI	IRET
MISCELLANEOUS		
Halt	HALT	HLT
Wait	WAIT	WAIT
Enable Interrupt	EI	STI
Disable Interrupt	DI	CLI
Adjust Nibble Subtract	ADJ4S	DAS
Adjust Nibble Add	ADJ4A	DAA
Adjust Byte Subrtact	ADJBS	AAS
Adjust Byte Add	ADJBA	AAA
Convert Binary to Decimal	CVTBD	AAM
Convert Decimal to Binary	CVTDB	AAD
Push	PUSH	PUSH, PUSHF
Pop	POP	POP,POPF
Repeat	REP	REP
Translate	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.

PROGRAM DESIGN LANGUAGE

```
Start
  BC<--0FFH
  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
    C<--C-1
  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		ALTERNATE REGISTER SET	
A	Accumulator	A′	Accumulator
F	Flag Register	F′	Flag Register
B	General Purpose	B′	General Purpose
C	General Purpose	C′	General Purpose
D	General Purpose	D′	General Purpose
E	General Purpose	E′	General Purpose
H	General Purpose	H′	General Purpose
L	General Purpose	L′	General Purpose

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

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

IX Index Register

IY Index Register

SP Stack Pointer

PC Program Counter

I Interrupt Vector R Memory Refresh

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

REGISTER	SIZE (bits)	REMARKS
A,A′ Accumulator	8	Stores an operand or the results of an of an operation.
F,F′ Flags (PSW,PSW′)	8	See instruction set.
B,B′ General purpose	8	Can be used separately or as a 16-bit register with C.
C,C′ General Purpose	8	See B, above.
D,D′ General Purpose	8	Can be used separately or as a 16-bit register with E.
E,E′ General Purpose	8	See D, above.
H,H′ General Purpose	8	Can be used separately or as a 16-bit register with L.
L,L′ General Purpose	8	See H, above. 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
I Interrupt Register	8	Stores upper eight bits of memory address for vectored interrupt processing.
R Refresh Register	8	Provides user-transparent dynamic memory refresh. Automatically incremented and placed on the address bus during each instruction fetch cycle.
IX Index Register	16	Used for indexed addressing.
IY Index Register	16	Same as IX, above.
SP Stack Pionter	16	Stores addresses or data temporarily. See Push or Pop in instruction set.
PC Program Counter	16	Holds address of next instruction.

ZILOG Z80 MNEMONICS

SYMBOL	OPERATION
r	one of the 8-bit registers A,B,C,D,E,H,L
n	any 8-bit absolute value
ii	an index register reference, either X or Y
d	an 8-bi index displacement, where -128 < d < 127
zz	B for the BC register pair, D for the DE pair
nn	any 16-bit value, absolute or relocatable
rr	B for the BC register pair, D for the DE pair, H for the HL pair, SP for the stack pointer
qq	B for the BC register pair, D for the DE pair, H for the HL pair, PSW for the A/Flag pair.
s	any of r (defined above), M, or d(ii)
IFF	interrupt flip-flop
CY	carry flip-flop
ZF	zero flag
tt	B for the BC register pair, D for the DE pair, SP for the stack pointer, X for index register IX
uu	B for the BC register pair, D for the DE pair, SP for the stack pointer, Y for index register IY

b	a bit position in an 8-bit byte, where the bits are numbered from right to left 0 to 7.
PC	program counter
b{n}	bit n of the 8-bit value or register v
vv/H	the most significant byte of the 16-bit value or register vv
vv/L	the least significant byte of the 16-bit value or register vv
Iv	an input operation on port v
Ov	an output operation on port v
w <-- v	the value of w is replaced by the value of v
w <--> v	the value of w is exchanged with the value of v

8 BIT LOAD GROUP

ZILOG MNEMONIC	OPERATION
LD r,r'	r <-- r'
LD r,(HL)	r <-- (HL)
LD r,(Iii + d)	r <-- (ii + d)
LD (HL),r	(HL) <-- r
LD (Iii + d),r	(ii + d) <-- r
LD r,n	r <-- n
LD (HL),n	(HL) <-- n
LD (Iii + d),n	(ii + d) <-- n
LD A,(nn)	A <-- (nn)
LD (nn),A	(nn) <-- A
LD A,(zz)	A <-- (zz)
LD (zz),A	(zz) <-- A
LD A,I	A <-- I
LD A,R	A <-- R
LD I,A	I <-- A
LD R,A	R <-- A

16 BIT LOAD GROUP

Z80 MNEMONIC	OPERATION
LD rr,nn	rr <-- nn
LD ii,nn	ii <-- nn
LD BC,(nn)	B <-- (nn + 1) C <-- (nn)
LD DE,(nn)	D <-- (nn + 1) E <-- (nn)
LD HL,(nn)	H <-- (nn + 1) L <-- (nn)
LD IX,(nn)	IX/H <-- (nn + 1) IX/L <-- (nn)
LD IY,(nn)	IY/H <-- (nn + 1) IY/L <-- (nn)
LD SP,(nn)	SP/H <-- (nn + 1) SP/L <-- (nn)
LD (nn),BC	(nn + 1) <-- B (nn) <-- C
LD (nn),DE	(nn + 1) <-- D (nn) <-- E
LD (nn),HL	(nn + 1) <-- H (nn) <-- L
LD (nn),IX	(nn + 1) <-- IX/H (nn) <-- IX/L
LD (nn),IY	(nn + 1) <-- IY/H (nn) <-- IY/L
LD (nn),SP	(nn + 1) <-- SP/H (nn) <-- SP/L
LD SP,HL	SP <-- HL
LD SP,IX	SP <-- IX
LD SP,IY	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

Z80 MNEMONIC	OPERATION
EX DE,HL	HL <--> DE
EX AF,AF'	PSW <--> PSW'
EXX	BCDEHL <--> BCDEHL'
EX (SP),HL	H <--> (SP + 1) L <--> (SP)
EX (SP),IX	IX/H <--> (SP + 1) IX/L <--> (SP)
EX (SP),IY	IY/H <--> (SP + 1) IY/L <--> (SP)
LDI	(DE) <-- (HL) DE <-- DE + 1 HL <-- HL + 1 BC <-- BC-1
LDIR	repeat LDI until BC=0
LDD	(DE) <-- (HL) DE <-- DE-1 HL <-- HL-1 BC <-- BC-1
LDDR	repeat LDD until BC=0
CPI	A - (HL) HL <-- HL + 1 BC <-- BC-1
CPIR	repeat CCI until A=(HL)
CPD	A - (HL) HL <-- HL-1 BC <-- BC-1
CPDR	repeat CCD until A=(HL) or BC=0

8 BIT ARITHMETIC AND LOGICAL

Z80 MNEMONIC	OPERATION
ADD A,r	A <-- A + r
ADD A,(HL)	A <-- A + (HL)
ADD A,(Iii + d)	A <-- A + (ii + d)
ADD A,n	A <-- A + n
ADC A,s	A <-- A + s + CY
ADC A,n	A <-- A + n + CY
SUB s	A <-- A - s
SUB n	A <-- A - n
SBC A,s	A <-- A - s - CY
SBC A,n	A <-- A - n - CY
AND s	A <-- A ^ s
AND n	A <-- A ^ n
OR s	A <-- A v s
OR n	A <-- A v n
XOR s	A <-- A + s
XOR n	A <-- A + n
CP s	A - s
CP n	A - n
INC r	r <-- r + 1
INC (HL)	(HL) <-- (HL) + 1
INC (Iii + d)	(ii + d) <-- (ii + d) + 1
DEC r	r <-- r - 1
DEC (HL)	(HL) <-- (HL) - 1
DEC (Iii + d)	(kk + d) <-- (ii + d) - 1

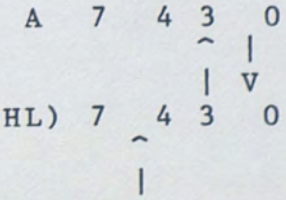
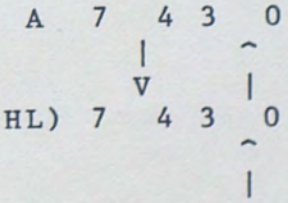
GENERAL PURPOSE ARITHMETIC AND CONTROL GROUP

Z80 MNEMONIC	OPERATION
DAA	convert A to packed BCD after an add or subrtact of packed BCD operands
CPL	A <-- ~A~A
NEG	A <-- -A
CCF	CY <-- ~CY
SCF	CY <-- 1
NOP	no operation
HALT	halt
DI	IFF <-- 0
EI	IFF <-- 1
IM0	interrupt mode 0
IM1	interrupt mode 1
IM2	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

ROTATE AND SHIFT GROUP

Z80 MNEMONIC	OPERATION
RLCA	CY <-- 7 <-- 0 <-- A
RLA	CY <-- 7 <-- 0 <-- A
RRCA	7 --> 0 --> CY A
RRA	7 --> 0 --> CY A
RLC r	Same diagram as for RLC
RLC (HL)	Same diagram as for RLC
RLC (Iii + d)	Same diagram as for RLC
RL S	Same diagram as for RAL
RRC s	Same diagram as for RRC
RR s	Same diagram as for RAR
SLA s	CY <-- 7 <-- 0 <-- 0 s
SRA s	7 --> 0 --> CY s
SRL s	0 --> 7 --> 0 --> CY s
RLD	
RRD	

BIT SET, RESET, AND TEST GROUP

Z80 MNEMONIC	OPERATION
BIT b,r	ZF <-- ~r~r{b}
BIT b,(HL)	ZF <-- ~(HL){b}
Bit b,(Iii + d)	ZF <-- (~(Iii + d){b}
SET b,r	r{b} <-- 1
SET b,(HL)	(HL){b} <-- 1
SET b,(Iii + d)	(Iii + d){b} <-- 1
RES b,s	S{b} <-- 0

JUMP GROUP

Z80 MNEMONIC	OPERATION
JP nn	PC <-- nn
JP Z,nn	if zere, then JMP else continue
JP NZ,nn	if not zero
JP C,nn	if carry
JP NC,nn	if not carry
JP PO,nn	if parity odd
JP PE,nn	if parity even
JP P,nn	if sign positive
JP M,nn	if sign negative
JP PE,nn	if overflow
JP PO,nn	if no overflow
JR e	PC <-- PC + e where e=nn - PC -126< e < 129
JR Z,e	if zero, then JMPR else continue
JR NZ,e	if not zero
JR C,e	if carry
JRNC,en	if not carry
DJNZ e	B <-- B - 1 if B=0 then continue else JMPR
JP (HL)	PC <-- HL
JP (IX)	PC <-- IX
JP (IY)	PC <-- IY

CALL AND RETURN GROUP

Z80 MNEMONIC	OPERATION
CALL nn	(SP-1) <-- PC/H (SP-2) <-- PC/L SP <-- SP-2 PC <-- nn
CALL Z,nn	if zero, then CALL else continue
CALL NZ,nn	if not zero
CALL C,nn	if carry
CALL NC,nn	if not carry
CALL PO,nn	if parity odd
CALL PE,nn	if parity even
CALL P,nn	if sign positive
CALL M,nn	if sign negative
CALL PE,nn	if overflow
CALL PO,nn	if no overflow
RET	PC/H <-- (SP + 1) PC/L <-- (SP) SP <-- SP + 2
RET Z	if zero, then RET else continue
RET NZ	if not zero
TER C	if carry
RET NC	if not carry
RET PO	if parity odd
RET PE	if parity even
RET P	if sign positive
RET M	if sign negative
RET PE	if overflow
RET PO	if no overflow
RETI	return from interrupt
RETN	return from non-maskable interrupt
RST n	(SP-1) <-- PC/H (SP-2) <-- PC/L PC <-- 8 * n where 0 < n < 8

INPUT AND OUTPUT GROUP

Z80 MNEMONIC	OPERATION
IN A,(n)	A <-- In
IN r,(C)	r <-- I(C)
INI	(HL) <-- I(C) B <-- B - 1 HL <-- HL + 1
INIR	repeat INI until B=0
IND	(HL) <-- I(C) B <-- B - 1 HL <-- HL - 1
INDR	repeat IND until B=0
OUT (n),A	On <-- A
OUT (C),r	O(C) <-- r
OUTI	O(C) <-- (HL) B <-- B - 1 HL <--HL + 1
OTIR	repeat OUTI until B=0
OUTD	O(C) <-- (HL) B <-- B - 1 HL <-- HL - 1
OUDR	repeat OUTD until B=0

TDL Z80 MNEMONICS

SYMBOL	OPERATION
r	one of the 8-bit registers A,B,C,D,E,H,L
n	any 8-bit absolute value
ii	an index register reference, either X or Y
d	an 8-bi index displacement, where -128 < d < 127
zz	B for the BC register pair, D for the DE pair
nn	any 16-bit value, absolute or relocatable
rr	B for the BC register pair, D for the DE pair, H for the HL pair, SP for the stack pointer
qq	B for the BC register pair, D for the DE pair, H for the HL pair, PSW for the A/Flag pair.
s	any of r (defined above), M, or d(ii)
IFF	interrupt flip-flop
CY	carry flip-flop
ZF	zero flag
tt	B for the BC register pair, D for the DE pair, SP for the stack pointer, X for index register IX
uu	B for the BC register pair, D for the DE pair, SP for the stack pointer, Y for index register IY

b	a bit position in an 8-bit byte, where the bits are numbered from right to left 0 to 7.
PC	program counter
b{n}	bit n of the 8-bit value or register v
vv/H	the most significant byte of the 16-bit value or register vv
vv/L	the least significant byte of the 16-bit value or register vv
Iv	an input operation on port v
Ov	an output operation on port v
w <-- v	the value of w is replaced by the value of v
w <--> v	the value of w is exchanged with the value of v

8 BIT LOAD GROUP

TDL MNEMONIC	OPERATION
MOV r,r′	r <-- r′
MOV r,M	r <-- (HL)
MOV r,d(ii)	r <-- (ii + d)
MOV M,r	(HL) <-- r
MOV d(ii),r	(ii + d) <-- r
MVI r,n	r <-- n
MVI M,n	(HL) <-- n
MVI d(ii), n	(ii + d) <-- n
LDA nn	A <-- (nn)
STA nn	(nn) <-- A
LDAX zz	A <-- (zz)
STAX zz	(zz) <-- A
LDAI	A <-- I
LDAR	A <-- R
STAI	I <-- A
STAR	R <-- A

16 BIT LOAD GROUP

TDL MNEMONIC	OPERATION
LXI rr,nn	rr <-- nn
LXI ii,nn	ii <-- nn
LBCD nn	B <-- (nn + 1) C <-- (nn)
LDED nn	D <-- (nn + 1) E <-- (nn)
LHLD nn	H <-- (nn + 1) L <-- (nn)
LIXD nn	IX/H <-- (nn + 1) IX/L <-- (nn)
LIYD nn	IY/H <-- (nn + 1) IY/L <-- (nn)
LSPD nn	SP/H <-- (nn + 1) SP/L <-- (nn)
SBCD nn	(nn + 1) <-- B (nn) <-- C
SDED nn	(nn + 1) <-- D (nn) <-- E
SHLD nn	(nn + 1) <-- H (nn) <-- L
SIXD nn	(nn + 1) <-- IX/H (nn) <-- IX/L
SIYD nn	(nn + 1) <-- IY/H (nn) <-- IY/L
SSPD 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

TDL MNEMONIC	OPERATION
XCHG	HL <--> DE
EXAF	PSW <--> PSW'
EXX	BCDEHL <--> BCDEHL'
XTHL	H <--> (SP + 1) L <--> (SP)
XTIX	IX/H <--> (SP + 1) IX/L <--> (SP)
XTIY	IY/H <--> (SP + 1) IY/L <--> (SP)
LDI	(DE) <-- (HL) DE <-- DE + 1 HL <-- HL + 1 BC <-- BC-1
LDIR	repeat LDI until BC=0
LDD	(DE) <-- (HL) DE <-- DE-1 HL <-- HL-1 BC <-- BC-1
LDDR	repeat LDD until BC=0
CCI	A - (HL) HL <-- HL + 1 BC <-- BC-1
CCIR	repeat CCI until A=(HL)
CCD	A - (HL) HL <-- HL-1 BC <-- BC-1
CCDR	repeat CCD until A=(HL) or BC=0

8 BIT ARITHMETIC AND LOGICAL

TDL MNEMONIC	OPERATION
ADD r	A <-- A + r
ADD M	A <-- A + (HL)
ADD d(ii)	A <-- A + (ii + d)
ADI n	A <-- A + n
ADC s	A <-- A + s + CY
ACI n	A <-- A + n + CY
SUB s	A <-- A - s
SUI n	A <-- A - n
SBB s	A <-- A - s - CY
SBI n	A <-- A - n - CY
ANA s	A <-- A ^ s
ANI n	A <-- A ^ n
ORA s	A <-- A v s
ORI n	A <-- A v n
XRA s	A <-- A + s
XRI n	A <-- A + n
CMP s	A - s
CPI n	A - n
INR r	r <-- r + 1
INR M	(HL) <-- (HL) + 1
INR d(ii)	(ii + d) <-- (ii + d) + 1
DCR r	r <-- r - 1
DCM M	(HL) <-- (HL) - 1
DCR d(ii)	(kk + d) <-- (ii + d) - 1

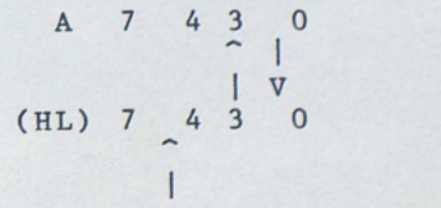
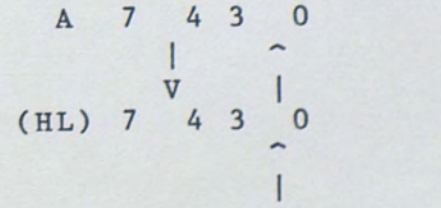
GENERAL PURPOSE ARITHMETIC AND CONTROL GROUP

TDL MNEMONIC	OPERATION
DAA	convert A to packed BCD after an add or subrtact of packed BCD operands
CMA	A <-- ~A
NEG	A <-- -A
CMC	CY <-- ~CY
STC	CY <-- 1
NOP	no operation
HLT	halt
DI	IFF <-- 0
EI	IFF <-- 1
IM0	interrupt mode 0
IM1	interrupt mode 1
IM2	interrupt mode 2

16 BIT ARITHMETIC GROUP

TDL MNEMONIC	OPERATION
DAD rr	HL <-- HL + rr
DADC rr	HL <-- HL + rr + CY
DSBC rr	HL <-- HL - rr - CY
DADX tt	IX <-- IX + tt
DADY uu	IY <-- IY + uu
INX rr	rr <-- rr + 1
INX ii	ii <-- ii + 1
DCX rr	rr <-- rr - 1
DCX ii	ii <-- ii - 1

ROTATE AND SHIFT GROUP

TDL MNEMONIC	OPERATION
RLC	CY <-- 7 <-- 0 <-- A
RAL	CY <-- 7 <-- 0 <-- A
RRC	7 --> 0 --> CY A
RAR	7 --> 0 --> CY A
RLCR r	Same diagram as for RLC
RLCR M	Same diagram as for RLC
RLCR d(ii)	Same diagram as for RLC
RALR s	Same diagram as for RAL
RRCR s	Same diagram as for RRC
RARR s	Same diagram as for RAR
SLAR s	CY <-- 7 <-- 0 <-- 0 s
SRAR s	7 --> 0 --> CY s
SRLR s	0 --> 7 --> 0 --> CY s
RLD	
RRD	

BIT SET, RESET, AND TEST GROUP

TDL MNEMONIC	OPERATION
BIT b,r	ZF <-- ~r{b}
BIT b,M	ZF <-- ~(HL){b}
BIT b,d(ii)	ZF <-- ~(Iii + d){b}
SET b,r	r{b} <-- 1
SET b,m	(HL){b} <-- 1
SET b,d(ii)	(Iii + d){b} <-- 1
RES b,s	S{b} <-- 0

JUMP GROUP

TDL MNEMONIC	OPERATION
JMP nn	PC <-- nn
JZ nn	if zere, then JMP else continue
JNZ nn	if not zero
JC nn	if carry
JNC nn	if not carry
JPO nn	if parity odd
JPE nn	if parity even
JP nn	if sign positive
JM nn	if sign negative
JO nn	if overflow
JNO nn	if no overflow
JMPR nn	PC <-- PC + e where e=nn - PC -126< e < 129
JRZ nn	if zero, then JMPR else continue
JRNZ nn	if not zero
JRC nn	if carry
JRNC nn	if not carry
DJNZ nn	B <-- B - 1 if B=0 then continue else JMPR
PCHL	PC <-- HL
PCIX	PC <-- IX
PCIY	PC <-- IY

CALL AND RETURN GROUP

TDL MNEMONIC	OPERATION
CALL nn	(SP-1) <-- PC/H (SP-2) <-- PC/L SP <-- SP-2 PC <-- nn
CZ	if zero, then CALL else continue
CNZnn	if not zero
CC nn	if carry
CNC nn	if not carry
CPO nn	if parity odd
CPE nn	if parity even
CP nn	if sign positive
CM nn	if sign negative
CO nn	if overflow
CNO nn	if no overflow
RET	PC/H <-- (SP + 1) PC/L <-- (SP) SP <-- SP + 2
RZ	if zero, then RET else continue
RNZ	if not zero
RC	if carry
RNC	if not carry
RPO	if parity odd
RPE	if parity even
RP	if sign positive
RM	if sign negative
RO	if overflow
RNO	if no overflow
RETI	return from interrupt
RETN	return from non-maskable interrupt
RST n	(SP-1) <-- PC/H (SP-2) <-- PC/L PC <-- 8 * n where 0 < n < 8

INPUT AND OUTPUT GROUP

TDL MNEMONIC	OPERATION
IN n	A <-- In
INP r	r <-- I(C)
INI	(HL) <-- I(C) B <-- B - 1 HL <-- HL + 1
INIR	repeat INI until B=0
IND	(HL) <-- I(C) B <-- B - 1 HL <-- HL - 1
INDR	repeat IND until B=0
OUT n	On <-- A
OUTP r	O(C) <-- r
OUTI	O(C) <-- (HL) B <-- B - 1 HL <--HL + 1
OUTIR	repeat OUTI until B=0
OUTD	O(C) <-- (HL) B <-- B - 1 HL <-- HL - 1
OUTDR	repeat OUTD until B=0

BIBLIOGRAPHY

Cohen, Harvey A., and Francis, Rhys S. "Macro-Assemblers and Macro-Based Languages in Microprocessor Software Development." Computer (February 1979): 43 - 57.

Crespi - Reghizzi, Stefano; Corti, Pierlvigi; and Dapra, Alberto. "A Survey of Microprocessor Languages." Computer (January 1980): 37 - 46.

Davies, Owen L. The Design and Analysis of Industrial Experiments. New York, New York: Longman, 1960.

Fairclough, Dennis A. "A Unique Microprocessor Instruction Set." Micro (May 1982): 8 - 19.

Fischer, Wayne P. "Microprocessor Assembly Language Draft Standard." Computer (December 1979): 16-40.

Johnson, Gearold R., and Mueller, Robert A. "Automated Generation of Cross - System Software for Microcomputers." Computer (January 1977): 10 - 17.

Korn, Granino A. "A Proposed Method for Simplified Microcomputer Programming." Computer (October 1975): 55 - 66.

Kriger, Morris. Structured Microprocessor Programming New York, New York: Yourdon, 1979.

Patterson, David A., and Piepho, Richard S. "Assessing RISCs in High-Level Language Support." Micro (November, 1982): 9 - 19.

Shneiderman, Ben. Software Psychology. Cambridge, Massachusett: Winthrop Publishers, Inc, 1980.

Tanenbaum, Andrew S.; Klint, Paul; and Bohm, Wim. "Guidelines for Software Portability." Software - Practice and Experience 8 (1978): 59 - 65.