POCKET COMPUTER

PC-1250/1251(1250A)

MACHINE LANGUAGE REFERENCE MANUAL

Der Prozessor ist identisch mit dem PC-1401. Die Systemadressen sind meinem Handbuch zu entnehmen.

P.

84060

INTRODUCTION

FOREWORD

Since the release of the PC-1250/1251 (1250A) on market, we have had great number of questions from users regarding the machine language of the PC-1250/1251 (1250A).

To meet with such demand from ardent users, we are now sending this text for study of the machine language of the Sharp's original design SC61860 Microprocessor in concern with the PC-1250/1251 (1250A) system. Because the text is edited on the basis of user questions, it may not support quality as a guidebook. In such an event, you are suggested to make reference to microprocessor guidebooks published on market, in addition to this text.

Your opinions and questions are welcome through our products distributor.

NOTE: Machine language program, which controls hardware directly, gives you more various functions than BASIC programs. However, you should check your machine language program enough to make no error before executing it because single wrong key operation may upset the program or occasionally make the machine break down. Sharp Corporation assumes no liability or responsibility of any kind arising from the use of programs or program materials or any part thereof.

CONTENTS

Introduction

For many programmers there comes a time when, regardless of the size or sophistication of the machine they program, they become dissatisfied with the exclusive use of a high-level programming language such as BASIC. Perhaps they want to make more efficient use of the available memory, they want to decrease the execution time of programs or perhaps they simply want to understand more about how the machine solves the problems presented to it. Whatever the cause, the programmer will need to learn about the assembler language or machine language of the particular machine being programmed.

This manual has been written to introduce the PC-1250/1251's (1250A) assembler and machine language, the command language for the ESR-H central processing unit.

While this manual provides much information about the PC-1250/1251 (1250A) and its resident BASIC, it was not intended to be a technical reference manual.

The material here assumes little beyond a working knowledge of BASIC and the operation of the PC-1250/1251 (1250A). Fundamental mathematical concepts, such as binary number systems, are reviewed in the context of their application to machine code programming. Likewise, fundamental machine code concepts are reviewed in the context of their application to the ESR-H language. This manual provides all the information needed to write a program in mnemonics, translate it into machine language and enter it into memory.

The transition from BASIC to machine language programming can be difficult. Machine code commands, being closer to what the machine understands, are even further from natural languages than the high-level language BASIC. In fact, many BASIC commands require more than ten or even twenty lines of machine code to accomplish similar actions. Also, space must be thought of differently at the machine code level. One must deal with fixed registers, fixed addresses, and the particular protocols for moving information from one location to another. However, the skills one developed while programming in BASIC, or which are developed programming in almost any computer language, will be invaluable in making the transition. With a bit of patience and study, you will become an able programmer for the ESR-H.

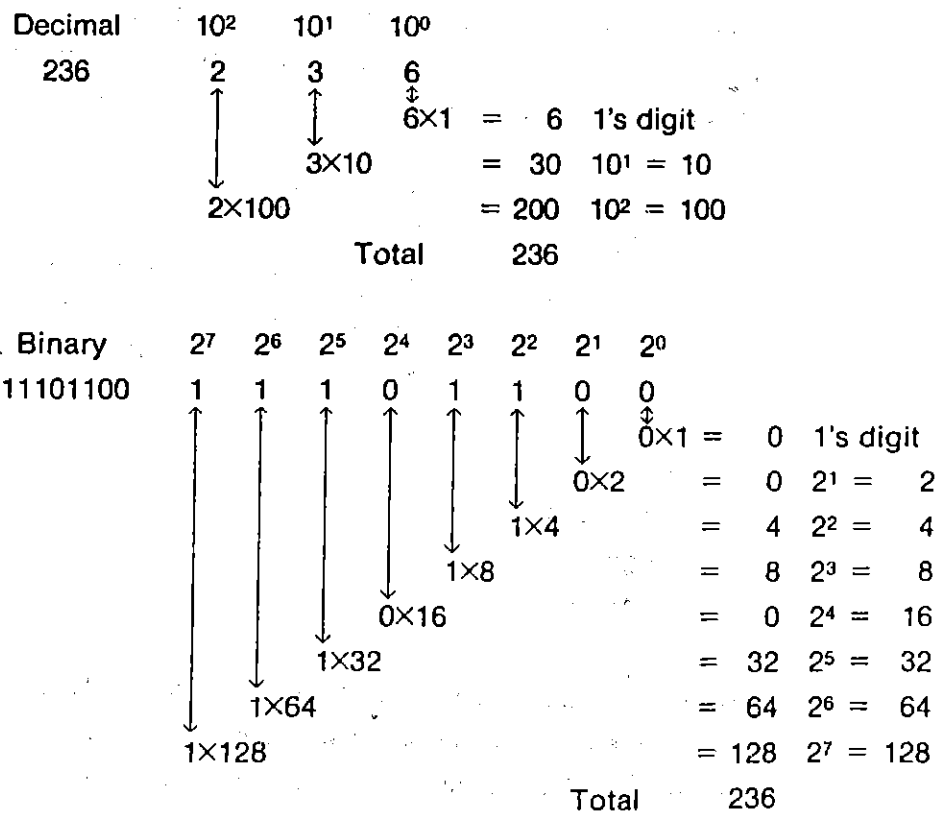
ıe Binary and Hexadecimal Number Systems

Memory in a computer consists of groups of binary digits, called "bits". A binary digit can have one of only two different values, 0 or 1. In the PC-1250/1251 (1250A), as in many other computers, 8 bits are grouped together to form one memory position, called a "byte". The left-most digit of a byte is called the "high-order digit" or "most significant bit" and the right-most is called the "low-order digit" or "least significant bit".

Each byte of memory has a unique location, and the description of that location is called an "address". Some addresses, those in internal memory, can be described with 1 byte (or sometimes even less than 1 byte) of information. Others, in external memory, require 2 bytes. Any byte of memory can contain several different kinds of information, but it is always in binary form, a series of eight 0's and 1's. The interpretation of the pattern of 0's and 1's in a particular byte is determined by the internal logic or programming of the machine or by an external program. More will be said about memory addresses and the kinds of information that can be stored in memory in a later section.

Since the only kind of numbers the computer can recognize are binary ones, any communication with the machine must be done using binary numbers. Every digit of a number in our familiar decimal system represents a power of 10. Likewise, each of the eight bits of a binary byte represents a power of 2.

The following illustration shows a decimal and a binary number having the same value, 236, and what each digit of the two numbers represents.



To convert a decimal number to binary, the following method of successive divisions by 2 can be used.

236	2 =	118	Remainder	0 — 0	(lowest bit)
118	2 =	59	Remainder	0 — 0	
59	2 =	29	Remainder	1 — 1	
29	2 =	14	Remainder	1 — 1	
14	2 =	7	Remainder	0 — 0	
7	2 =	3	Remainder	1 — 1	
3	2 =	1	Remainder	1 — 1	
1	2 =	0	Remainder	1 — 1	(highest bit)

The binary equivalent is 11101100.

Binary representation of numbers, with its series of zeros and ones, can be very confusing to humans. Because of this, various alternate ways of representing binary numbers are often used. One of these alternate notations, hexadecimal, is used in programming the PC-1250/1251 (1250A).

To convert an 8 bit binary number into hexadecimal, the 8 bits are first divided into 2 groups of four bits, then each group of 4 bits is assigned a single digit value. The result of this is a 2 digit number which has the same value as the 8 digit binary number. In order to represent each of all the possible values (0—15) of a 4 digit binary number with single digit we need 16 distinct characters, one for each of the 16 values.

0000 = 0	1010 = 10
0001 = 1	1011 = 11
0010 = 2	1100 = 12
0011 = 3	1101 = 13
0100 = 4	1110 = 14
0101 = 5	1111 = 15
0110 = 6	
0111 = 7	
1000 = 8	
1001 = 9	

Decimal representation requires 2 digits for these values

As can be seen in the table above, the decimal digits 0—9 are not sufficient to represent a of the binary combinations of 4 digits; another 6 characters are needed. Any characte could be used, but the standard for hexadecimal in computers is to use the alphabet characters A—F. 16 is the "base" of the hexadecimal system, just as is 10 (with 10 distin digits) for the decimal system and 2 (with 2 distinct digits) for the binary system.

The 16 digits of the hexadecimal system and their binary and decimal equivalents are:

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

It is important to remember that all of the numbers, in spite of their different appearance, in a single row across the 3 columns have the same actual value. 1110 = E = 14, but when one is working with these numbers E is much easier to keep track of than is 1110, especially when it is surrounded by other similar numbers. 23F016 is considerably less confusing to the human eye and brain than is 001000111111000000010110, which is the only form of the numbers that makes sense to the computer.

Binary Arithmetic

The rules for binary arithmetic are similar to those of decimal arithmetic. Addition can be summarized as follows:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10
```

Here 10, the binary equivalent of decimal 2, can be thought of as a 0 and a "carry". If for example, we add 3 and 1 in binary,

```
   11        1         1         1
 + 01      + 11      + 11      + 11
 ----      + 01      + 01      + 01
             0         00      100₂  =  4₁₀
```

we first add the one's place column. The total is 10_2, so we put a 0 in the sum's one's place and carry 1 into the two's place (the second column). The second column is then added, with again a result of 10, so a 0 is put in the two's place column and a 1 is carried to the four's place column, giving us the result of 100 (base 2) or 4 (base 10).

With eight bits, it is possible to represent numbers from 00000000 to 11111111, or in decimal, 0 to 255 ($=2^8-1$). With two bytes, of 16 bits, we can represent numbers from 0 to 65535 ($=2^{16}-1$). In order to represent negative numbers, we treat the high-order bit as a "sign bit". With single byte numbers, since this bit cannot now be used as a part of the numeric representation, the range of the number becomes −127 to +127. With two byte numbers, the range becomes −32767 to +32767.

```
Binary      −3 = 1000011,      +3 = 00000011
Binary     −03 = 100000000000011,     +3 = 000000000000011
```

One of the most commonly used forms of representation for negative binary numbers is what is known as "Two's Complement Representation". This representation allows us to add a negative number, i.e. subtract, using the addition command. A "one's complement" of a binary number is formed by reversing all of its digits. For example, the number −5, in one's complement form would be:

```
00000101    (5)
11111010    (−5)    One's Complement
```

By adding 1 to a one's complement representation of a negative number, we get the "two's complement" form of the number.

```
 11111010     One's complement form
      +1
 --------
 11111011  =  Two' complement form of −5
```

If we use the two's complement representation for negative numbers, we can use the same simple addition rules for subtraction and for addition of negative numbers. Take, for example, the subtraction 7−5. First, the five is put in two's complement form, then it is "added" to 7.

```
 00000111  =    7
 11111011  =   −5     (Two's complement form)
(1)00000010 =   2     (plus a binary carry)
```

If we ignore the carry, the answer is 2.

One consideration with this form of representation is that the result of an addition of 2 single byte numbers may require more than 7 bits. This condition is called "overflow", since the extra bit required to represent the results "overflows" into the high order sign bit. An overflow beyond the entire 8 bits of a byte is called a "carry". The extra bit of a carry is lost, but the occurrence of a carry causes the Carry Status Flag to be set to 1 to alert the programmer to the condition. An overflow into the high order sign bit will produce a false sign in the result of a binary addition under two conditions.

1. If both are positive and one or both have a large value.

sign bit		
(0)1111111	+127	(Largest positive number which can be represented in 7 bits)
+ (0)0000010	+ 2	
(1)0000001	−127	(False negative, interpreted as a 2's complement because of the 1 in the sign bit)

The result has a false negative sign. Any combination which would have a result of more than +127 (for a single byte number) would cause this error condition.

2. If both are negative and one or both have a large value.

sign bit		
(1)0000001	−127	(Largest negative number which can be represented in 7 bits in 2's complement notation)
+ (1)1111100	− 2	(in 2's complement notation)
−(1)(0)1111101	+125	(False positive, not interpreted as a 2's complement because of the 0 in the sign bit)
Carry is lost, Carry Flag set		

The result has a false positive sign. Any combination which would have a result of more than −127 (for a single byte number) would cause this error condition.

The programmer must check for these two error conditions by testing the Carry Flag and the sign bits themselves when they suspect that the result of an operation might cause an overflow error.

›gical and Bit Shift Operations

In addition to binary addition and subtraction, there are several binary logical operations and bit shifts which should be understood by the programmer.

Logical OR — The logical OR operation compares bit by bit all 8 bits of 2 individual bytes and produces a result based on the following conditions:

If both bits are 0, result = 0
If either bit is 1, result = 1

All of the possible combinations and results are:

Byte A 1 Bit	Byte B 1 Bit	Result
0	0	0
0	1	1
1	0	1
1	1	1

This operation can be used to place a 1 bit in selected location(s) of a byte. If we want to add a negative sign bit to a positive number, for example, to change 5 to 7, we can do the following:

A	00000101	5
OR with B	00000010	2
Result	00000111	7

Only the 2's position bit has been changed.

Logical AND — The logical AND operation compares each of the 8 bits of two bytes and produces a result based on the following conditions:

If both bits are 1, result = 1
If either bit is 0, result = 0

The possible combinations and results are:

Byte A 1 Bit	Byte B 1 Bit	Result
0	0	0
0	1	0
1	0	0
1	1	1

This operation can be used to remove or test for a 1 bit in selected location(s) of a byte. If we want to change the 7 we produced in the OR example back into a 5, we could do the following:

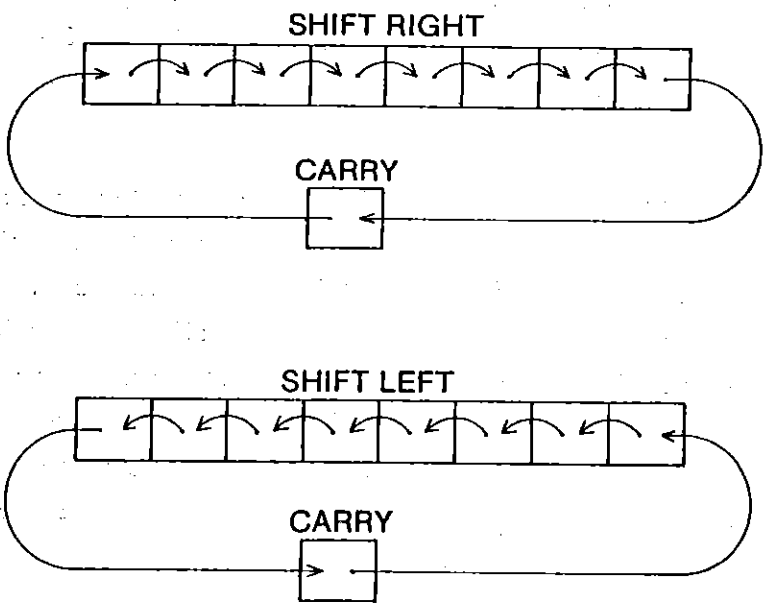
A	00000111	7
AND with B	11111101	253
Result	00000101	5

Again, only the 2's position bit has been changed.

Bit Shift Operations — Two instructions that shift the bits of a single byte to the right or left are provided in the PC-1250/1251 (1250A) instruction set.

1. Shift Right — Each bit of a byte is shifted one bit position to the right. The Least Significant Bit, which is pushed out of the byte, is stored in the Carry Flag Position and the previous contents of the Carry Flag is stored in the Most Significant Bit of the byte. This operation gives a result that is the same as dividing by two, and is useful for division routines.

2. Shift Left — Each bit of a byte is shifted 1 bit position to the left. The Most Significant Bit, which is pushed out of the byte, is stored in the Carry Flag Position and the previous contents of the Carry Flag is stored in the Least Significant Bit of the byte. This operation gives a result that is the same as multiplying by two and is useful for multiplication routines.



Binary Coded Decimal

Another type of representation of numbers that provides greater accuracy for such applications as accounting, where more precision is necessary, is called BCD or Binary Coded Decimal. The decimal numbers 0—9 can be represented in binary in four bits, one half byte (called a "nibble"). Since only a half byte is needed, two decimal numbers can be coded into each byte. This representation of decimal numbers is called "Packed BCD". Some of the binary values that can be expressed in 4 bits, that is, binary 10—15, are not needed to express the decimal digits 0—9. These unneeded values are not used in BCD and can cause some problems in BCD arithmetic. However, the BCD instructions in the PC-1250/1251 (1250A) instruction set automatically make the necessary adjustments so the programmer need not worry about them. The BCD values 0—9 are shown in the chart below:

BIN	DEC	BCD	BIN	DEC	BCD	BIN	DEC	BCD
0000 =	0	0	0101 =	5	5	1010 =	10	Not Used
0001 =	1	1	0110 =	6	6	1011 =	11	
0010 =	2	2	0111 =	7	7	1100 =	12	
0011 =	3	3	1000 =	8	8	1101 =	13	
0100 =	4	4	1001 =	9	9	1110 =	14	
						1111 =	15	

A number expressed in BCD must be limited to a fixed number of digits, in the PC-1250/1251 (1250A) it is 10 digits. In order to represent numbers that are larger than the largest number, or in the case of fractions, smaller than the smallest number that can be expressed in 10 digits, a representation called Floating Point is used. Essentially, what this format allows is the elimination of the need to represent zeros on either side of the decimal point and subsequently the elimination of the bytes needed to hold these zeros.

Equivalent numbers can be represented by shifting the location of the decimal point and multiplying them by 10 to the appropriate power. Thus the decimal number 23,000.00 could be represented as:

$$2300.00 \times 10^{1}$$
or $$230.00 \times 10^{2}$$
or $$23.00 \times 10^{3}$$
or $$2.30 \times 10^{4}$$

Numbers to the right of the decimal point are represented by exponents with a minus sign. The number .00023 could be represented as:

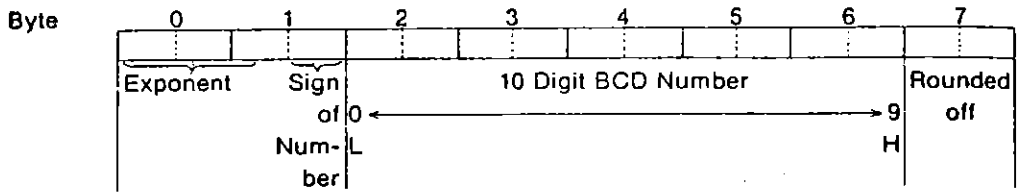
	.0023	$\times 10^{-1}$
or	.023	$\times 10^{-2}$
or	.23	$\times 10^{-3}$
or	2.3	$\times 10^{-4}$

All of these combinations are possible, but in the PC-1250/1251 (1250A) the number is represented with the decimal point to the right of the left-most digit:

2.3×10^{4}
2.3×10^{-4}

Internal Representation of Numbers and Character Strings in BASIC

In the PC-1250/1251 (1250A), the pre-allocated variables store numbers in the following floating point form:



The first three half-bytes, or nibbles, are used to store the exponent of the number Negative exponents are represented in a BCD complement form, where:

Exponent	Exponent
001 = 10^{1}	999 = 10^{-1}
002 = 10^{2}	998 = 10^{-2}
003 = 10^{3}	997 = 10^{-3}
004 = 10^{4}	996 = 10^{-4}

The sign of the number is stored in the fourth nibble. A 0 designates positive, an 8 designates negative. The next 5 bytes hold 10 BCD digits. The final byte is used for accuracy during computations, but is rounded off in the pre-allocated variable.

Character string variables use the same pre-allocated space as numerical variables, and thus are also represented in 8 bytes.

0	1	2	3	4	5	6	7
F5	CHAR	CHAR	CHAR	CHAR	CHAR	CHAR	CHAR

The first byte of a character string variable is always F5. The following bytes hold the internal representation of the individual characters. The presence of 00 anywhere in bytes 1—7 indicates the end of the string, but if the string is 7 characters long, no 00 is necessary.

The internal representation of the characters used by the PC-1250/1251 (1250A) is presented in the chart in Appendix C. Note that this is not an ASCII chart. While the characters can be treated as ASCII values in BASIC, they are not represented by their ASCII values, but by this internal representation. For example, the character string "HELLO" is represented as:

F5	58	55	5C	5C	5F	00	
	H	E	L	L	O		

System Architecture

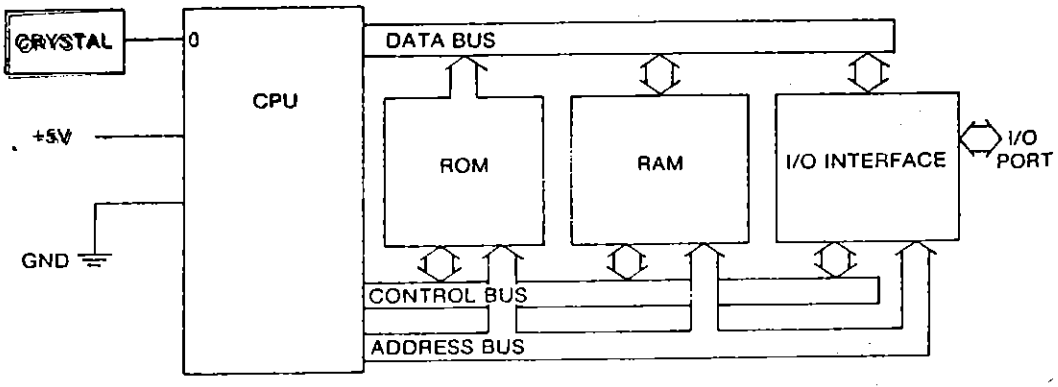
Figure 1 shows the architecture of a typical microcomputer system. The central-processing unit (the CPU) appears on the left side of the diagram. The CPU includes several internal registers, the arithmetic-logic unit (the ALU), a control unit which decodes and internally executes instruction sequences and an internal RAM area used for registers, a system stack and I/O access. The CPU includes a system clock which takes its timing from a quartz crystal external to the system, and is connected to a power supply and ground. In addition, the CPU has three buses, the 8 bit bidirectional data bus, a 16 bit unidirectional address bus, and a control bus.

Data is moved from source to destination on the data bus a single byte at a time. Data can be brought from memory to the CPU or vice versa, or moved about within the CPU, or moved between the CPU and the I/O interface. When data is moved from one memory locaton to another, it does not take a direct path, but passes through the CPU registers in an intermediate stage.

The 16 bit address bus can address 64K memory locations. The address is generated by the CPU. This address specifies where in memory the data to be transmitted on the data bus is to originate from or be sent to. The control bus coordinates the various activities in the system.

Three regions are illustrated to the right of the CPU. They are the ROM, RAM, and I/O interface. The ROM (read only memory) holds codes which can be read but cannot be altered. The command interpreter resides in an 8K byte ROM region in the CPU. The language BASIC is brought up by a 16K system ROM external to the CPU.

Fig. 1 A Simplified Diagram of System Architecture



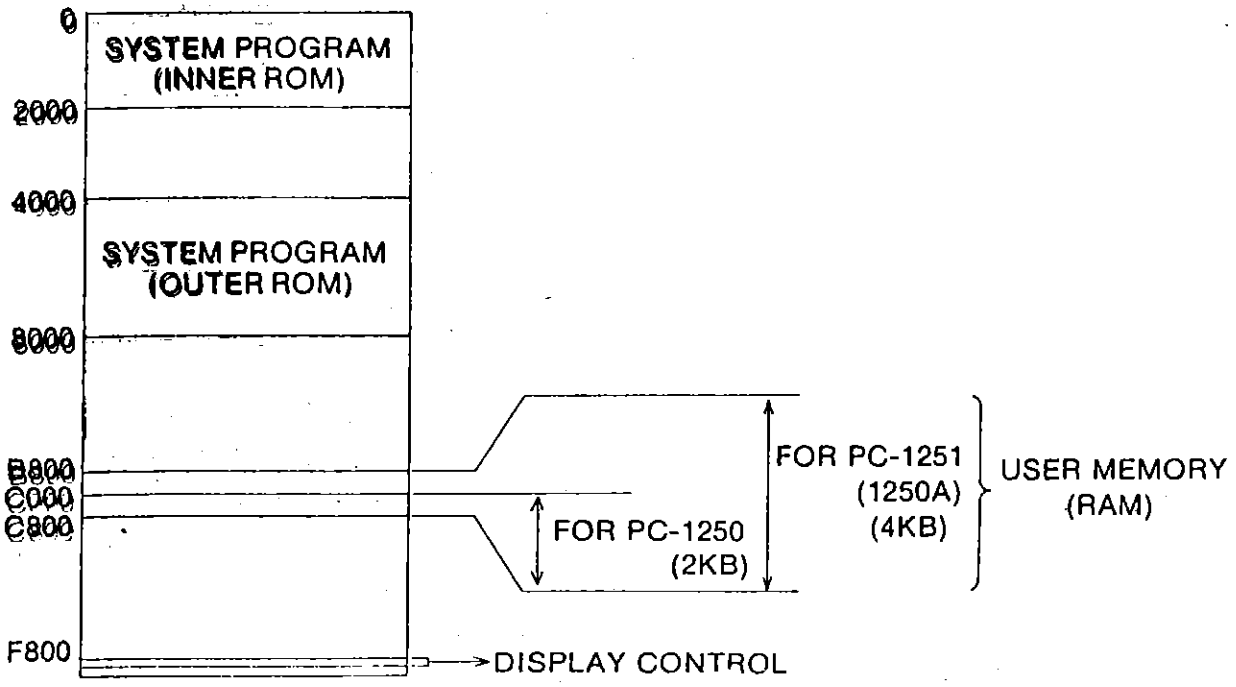
RAM (random access memory) is the read/write memory for the system. All calculations which must be made during execution of a BASIC program, from keeping track of FOR NEXT loops and subroutine nesting, to evaluating numerical equations must be done in RAM. All programs written by the user will also reside in RAM. A chart which shows the addresses of area available to the user is in a later section of the manual.

The PC-1250/1251 (1250A) must also connect to several I/O interface chips. These chips link up to externals such as the keyboards, the LCD display and the CE-125 printer & microcassette recorder. Like the RAM and ROM, the I/O interfaces must connect to all three buses. Because of the relatively complicated nature of handling I/O, we are only going to illustrate the use of one peripheral at the machine language level, the LCD display.

All other I/O can be handled by the programmer in BASIC.

Finally the actual system has many more components than are illustrated here. As they are not relevant to programming, they will not be described here.

'stem Memory Map



The CPU

The CPU (central processing unit) plays the leading part in the execution of any command. If data is to be moved, it must make at least one stop in the CPU. If data is modified, that modification happens in the CPU. In the following section we will first describe the internal structure of the CPU in greater detail, then we will step the reader through an execution cycle of a command.

Fig. 2 A Simplified Diagram of the ESR-H CPU

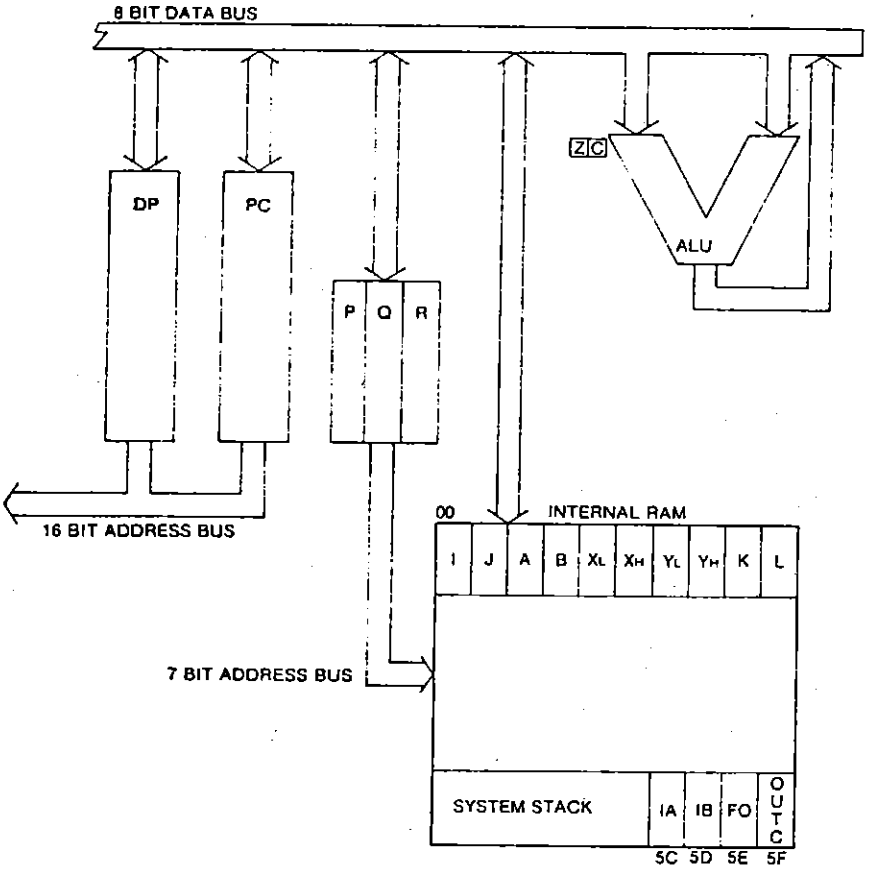


Fig. 2 is a simplified illustration of the internals of the ESR-H CPU. Both the data bus and the address bus are present in the diagram. Though we have not included the control bus in the illustration, the same critical timing control exists in the CPU. Note also, that there is a second address bus. This 7-bit bus is used exclusively to address a 96-byte area of internal RAM. More will be said about this area presently.

The ALU (the arithmetic logic unit) performs arithmetic 6 logical operations. The ALU can take one or two arguments, and stores the result in a register, usually A, of the internal RAM. Connected to the ALU are two flags, the carry flag (C) and the zero flag (Z). Flags can be thought of as single bit registers. Thus, they are either SET (=1) or RESET (=0). If the result of an operation results in a 0, then the Z flag will be set. If the result is none-zero, then the flag will be reset. The carry flag becomes set if an operation results in a carry. It will become 0 if an operation does not cause a carry. The contents of the flags can be tested and can be altered directly. Conditional instructions, instructions which cause the execution of different parts of a program, will test one of the flags before branching.

Not all instructions affect the flags. Flag information can be found in the section of the manual describing the individual instructions.

On the left side of the diagram is the DP (data pointer) register. The DP is a 2 byte register used to address external memory. All loads or stores in external memory use the DP register address. The DP can be incremented, decremented, loaded with an immediate value, or loaded with an address from the internal registers X and Y.

Next to the DP is the PC (program counter). This 2 byte register contains the address of the next instruction to be executed in a program. Execution of a program is normally sequential, but can be altered with JUMP or CALL type commands which directly alter the contents of the PC.

The P, Q, and R registers are used to address the internal RAM. P and Q are commonly used to address the registers in this area. R is used as the pointer to the system stack which also resides in the internal RAM. These registers are only 7 bits since that is sufficient to address the 96 byte internal RAM.

The 1 byte I and J registers in internal RAM are used as index registers. In commands involving block movement of data, the I or J register, depending on the command, are assumed to contain the number of bytes to be moved.

The A and B registers are used respectively as an ACCUMULATOR and as a spare register. Most arithmetic and logical operations use the A register. Most data movement operations use the ACCUMULATOR (e.g. the LOAD and STORE group).

The X and Y registers are used as address pointers. Each is two bytes large, with the high byte preceding the low byte in memory. The X register is commonly used, as with the IXL command, to point to an address from which a value is loaded into the ACCUMULATOR. The Y register is used, as with the IYS command, to point to an address into which a value in the accumulator is stored.

The K & L registers are general purpose single byte registers.

In addition to these registers, there exists an I/O register at internal RAM address 5C, 5D, 5E, 5F. This single byte can be sent to an I/O port with the command OUTA, OUTB, OUTF, OUTC.

The system stack resides in the internal RAM. The stack is a LIFO (last in first out) structure. The R register always points to the top of the stack. Elements are PUSHED onto or PULLED off of the stack. The first element PUSHED onto a stack is always at the "bottom" of the stack; the element most recently PUSHED onto the stack is the elment on the top, and is the first to be removed when a PULL is executed. In the PC-1250/1251 (1250A), the stack starts at location 5B and grows downward in memory. The stack can be used to temporarily hold data, or it can be used to hold addresses for subroutine nesting. The commands, PUSH, POP, CALL, and RTN deal with the R register automatically.

Internal RAM Registers

ADDRESS	REGISTER
00	I
01	J
02	A
03	B
04	XH L
05	XL H
06	YH L
07	YL H
08	K
09	L

HE INSTRUCTION EXECUTION CYCLE

We will now describe what happens when an instruction is executed.

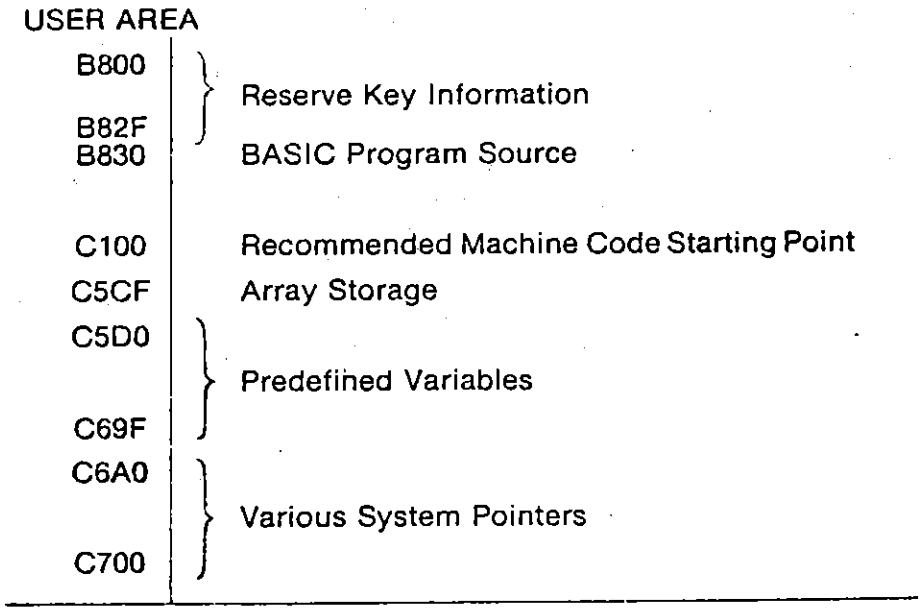
First, the contents of the PC (program counter) are placed on the address bus. The contents of that location are fetched and sent to a CPU register called the IR (instruction register). Once the instruction is in the IR, the control unit of the microprocessor decodes the contents and generates the correct sequence of internal and external signals for the execution of the instruction. There is, therefore, a short decoding delay, followed by an execution phase, the length of which depends on the nature of the instruction specified. Some instructions execute entirely in the CPU, others fetch or store data from memory. The number of 'clock cycles' required for the operation of a given instruction can be found in the section of the manual describing the individual instructions.

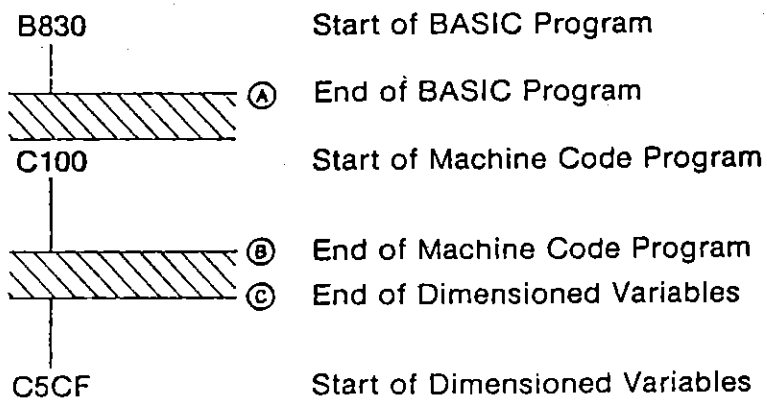
The PC is automatically incremented after the execution of an instruction, and so, is pointing to the next instruction to be executed. If an instruction requires a second or third byte for operation (e.g. LIA, or LIDP), then the PC is automatically incremented a second or third time as part of the execution cycle.

Thus, after the command has been executed, the PC will be pointing to the next successive command in a program.

Memory Area for Programs

Basic programs are stored starting at the address B830. When dimensioned variables a declared with the BASIC DIM command, space is allocated starting from address C5C and extending downward in memory as far as necessary (on the above chart, this ar would start at C5CF and appear to go upwards toward C100). The intermediate addre C100, is the recommended starting point for machine code programs, though care must taken that the BASIC program does not extend beyond C100 nor that the dimension variables extend back beyond the last address used by the machine code program.





Ⓐ must not be greater than C100

Ⓑ must not be greater than C

Ⓒ must not be less than B

The pertinent information concerning allocation of space within the user area resides in the following addresses:

Location		Contents
C6E1	Low Byte	Starting address of BASIC
C6E2	High Byte	program (contains the data FF)
C6E3	Low Byte	Ending address of BASIC
C6E4	High Byte	program (contains the data FF)
C6FC	Low Byte	Starting address of
C6FD	High Byte	dimensioned variables

PEEK, POKE & CALL

The three BASIC commands, PEEK, POKE and CALL provide the means of passing information and control between the machine language program and the BASIC program. PEEK allows us to read the contents of a memory location. "PEEK addr" will display the content of a given legal address expressed by "addr". The contents of that address may also be an address. For example, if we enter:

```
PEEK &C6E1
```

the value 48 (= HEX 30) is returned. If we PEEK at the next address, &C6E2, we find that it contains the number 184 (= HEX B8). Taking the first value as the low byte and the second value as the high byte, we have the address B830, which is, as indicated in the chart above, the starting position of BASIC programs.

The following short BASIC program can be used to "dump" the contents of a sequence of bytes by repeatedly depressing ENTER instead of specifying each byte separately. This program also has another useful feature, in that it will display the contents of the addresses in HEX format.

```
10:  INPUT "ADDR?" ;A
20:  INPUT "BYTES?" ;B
30:  FOR I=0 TO (B-1)
40:  X= PEEK (A+I)
50:  P= INT (X/16)
60:  Q= (X-16*P)
70:  IF P>9 THEN LET P=P+
     7
80:  IF Q>9 THEN LET Q=Q+
     7
90:  P=P+48:Q=Q+48
100: PRINT (A+I);"    ";
     CHR$ P; CHR$ Q;X
110: NEXT I
120: END
```

The address from which one wishes to start dumping is entered into A. The number of bytes is entered into B, and becomes the basis for the FOR loop from 30 to 110. The contents are dumped into the variable X, and then separated into the individual HEX digit values P an Q. In lines 70 and 80 a check is made to see whether or not the HEX digit is a number or the characters A—F. The number 48 added to the digits in line 90 converts the number to its ASCII equivalent. The address and its HEX contents are printed in line 100, the CHR$ function transforming the numeric ASCII value in the variables to their equivalent character representations.

Dumping the first 16 bytes starting at &B830, we can see that this is the first line of our BASIC program (assuming that nothing else is in the computer and the program has been entered as in the example).

47152	FF
47153	E0
47154	10
47155	C2
47156	12
47157	51
47158	54
47159	54
47160	62
47161	13
47162	12
47163	1C
47164	51
47165	00
47166	E0
47167	20

"FF" marks the beginning of the program "E" indicated that the next 3 digits are the lin number 010 or line 10. "C2" as seen in our chart of internal codes is the internal code for th BASIC command INPUT. "12" is the representation of " , 51 of the character A etc. 00 null- in location 47165 designates the end of this line of the program. E0, 20 tells us that th next line is line 20.

This PEEK program will be an invaluable tool for debugging machine code programs a well as for "exploring".

The POKE command is used to enter values into specific memory locations. The synta is as follows:

POKE addr, value 1, value 2, value 3,. . . value n where addr is a legal address and th values lie between 0 and 255. For example, if we enter:

```
POKE &C100, &12, &06, &02, &D7, &37
```

we can check the addresses with the PEEK program to find that the values hav successfully reached their destinations.

```
          RUN
ADDR?     &C100
BYTES?    4
49408     12
49409     06
49410     02
49411     D7
49412     37
```

To execute a machine language program, we will treat the program as if it were subroutine. We will "call" the starting address with the command CALL, that is:

```
CALL &C100
```

To return to BASIC after execution of the machine code program we use the machin command RTN following the last machine language command of the program. After a of the machine code is executed and the command RTN is reached, program executio will resume at the BASIC command following the CALL command.

Example Program

The following is a simple example program. The program will put a number in the accumulator (the number 6 was arbitrarily chosen) then print it at the memory location C6A0. This location was selected because it is the first address after the area reserved for variables and it is not being used for anything else. The assembly mnemonics for this operation are as follows:

```
LIA   06      Lade 6 → Akku
LIDP  C6A0    Lade C6A0 → DP register
STD           Schreibe Akku in die ADR des DP registers
RTN
```

The command LIA, load immediate accumulator, takes the immediately following number and places it in the A register, the accumulator. The next command, LIDP, load immediate data pointer, takes the next 2 bytes (here C6A0) and loads them into the high and low bytes of the DP register. The next command, STD, store in the DP address, takes the value in the A register and places it in the address contained in the DP register. The last statement, RTN, return, returns control back to the BASIC program. Thus, if all goes well, the program should leave a 6 in the memory location C6A0.

In order for the above program to run, it must be "hand assembled," that is, the program must be converted into its numeric, machine executable representation. To do this, a table similar to the following should be constructed:

Addr	Machine Code	Label	Op Code	Operand(s)	Comments
C100			LIA	06	value to be stored
			LIDP	C6A0	
			STD		
			RTN		

The column ADDRESS will contain the addresses in memory where the machine code will reside. The next column MACHINE CODE is where we will place our translated machine code commands. We will deal with the LABEL column later. In the OP CODE column we have the assembly code commands, one per horizontal line. Following that are the operands to the Command. Some commands, such as STD and RTN do not take arguments. Others, such as LIA, take one, or such as LIDP, take two. In the final column it is important to leave space for comments as it is all too easy to forget what is going on as the program gets larger.

After the assembly mnemonics have been written, the translation to machine code begins. The machine code equivalents of each command are found in the middle section of this book under the description of the individual commands. The commands are listed in order of type of function. Cross indices and tables are found at the end of this book to help speed up the search for the proper instruction. The numeric representation of the command is entered in the machine code column, followed by its operands, if any. The "LIA 06" for example, is translated to 0206, where the first byte, 02, designates the command LIA, and the second byte, 06, is the operand. As two bytes have been consumed by this command the next command will start at C102. This address is entered in the ADDRESS column on the line for the next command. LIDP translates to 10 with two operands, here C6 and A0 (high and low bytes of C6A0). This consumes three bytes, so the next ADDRESS entry is C105. STD takes one byte, a 52, and in the following location C106, a 37 is placed for the RTN command.

Addr	Machine Code	Label	Op Code	Operand(s)	Comments
C100	02 06		LIA	06	Value to be stored
C102	10 C6 A0		LIDP	C6 A0	Target Addr.
C105	52		STD		
C106	37		RTN		

Once the program has been translated to machine code values, it is ready to be placed memory with a BASIC program. As this machine code program is so short, we could use single POKE line to enter it; however, for illustrative purposes, we will use two POKE statements:

```
200: POKE &C100,&02,&06,&10,&C6,&A0
210: POKE &C105,&52,&37
220: END
```

You should take the time now to enter and execute this program. Assuming PEEKER program is still on lines 10—120, you should also take the time to check and that the proper values have indeed been placed in the proper locations.

```
200:POKE &4100,&02,&06,&
    10,&C6,&A0
210:POKE &4105,&52,&37
220:END
300:POKE &4000,0
310:PRINT "BEFORE";PEEK
    &4000
320:CALL &4100
330:PRINT "AFTER";PEEK &
    4000
340:END
```

PC 1401

After this has been done, try entering the following program.

```
300:  POKE  &C6A0,0
310:  PRINT  "BEFORE  ";
      PEEK  &C6A0
320:  CALL  &CA100
330:  PRINT  "AFTER  ";
      PEEK  &C6A0
340:  END
```

This program initializes our "interface" address contents to 0 and prints the contents out before and after a call to our machine code subroutine.

If all works well, when the programs are run in turn, the BEFORE value should be 0 and the AFTER value should be 6.

A Second Example: Binary to Hex. Conversion

This larger program will convert a binary byte to hexadecimal representation. Basicall it does the same thing as lines 50 through 90 of our PEEKER program, although a muc different algorithm is used. This example will illustrate some additional machine cod programming principles.

The program can be broken up into several parts. The first major section, lines C10 through C10C has a simple objective — to place "F5" in the first byte of the pre-allocate variable Y, thus indicating that Y will contain a character variable. (See section on interna variable structure.)

In the first six statements, the 16 bit address pointer Y in the internal RAM is loade with the address of the byte immediately before the first byte of the pre-allocated variabl Y. The DP register is difficult to alter directly (for example, there is no increment D instruction) but can be altered with the address registers X and Y. There are variou commands for loading the DP with the contents of the X or Y register, all of whic automatically increment or decrement X or Y. Here IY (increment Y) is used to load th DP, so a value of one less than the correct address is put in Y in the commands LIP an INCP where used to address the pointer register Y itself. All access to the internal RAM registers must be made through the P, not the DP, address pointer.

Once the DP has the correct address, the character variable header "F5" is loaded int the accumulator and then stored in memory (lines C109—C10B), in a slightly differen fashion than we did in the previous program.

You should try to load and execute these first 9 lines. Test Y$ with the PEEKER to see i the F5 reached its proper destination. (Remember to clear Y first with a Y=0 or some suc command.) As with BASIC, testing programs in smaller chunks is the recommende procedure.

BINARY TO HEXADECIMAL CONVERSION

Addr	Machine Code	Label	Mnemonic	Operand(s)	Comments
*C100	12 06		LIP	06	Address of YL
C102	02 D7		LIA	D7	
C104	DB		EXAM		
C105	50		INCP		address of YH
*C106	02 C5		LIA	C5	
C108	DB		EXAM		C5D8=Y$
C109	02 F5		LIA	F5	char variable header
C10B	26		IYS		store in Y$
*C10C	10 C6 A0		LIDP	C6A1	"window" address
C10F	57		LDD		get byte
C110	34		PUSH		save copy
C111	58		SWP		set up high nibble
C112	78 C1 1D	①	CALL		convert high nibble
*C115	5B		POP		get copy
C116	78 C1 1D		CALL	①	convert low nibble
C119	02 00		LIA	00	00=end of string
C11B	26		IYS		place null in Y$
C11C	37		RTN		return to basic
*C11D	64 0F	①	ANIA	0F	mask off top nibble
C11F	34		PUSH		save copy
C120	75 0A		SBIA	0A	will cause carry if result is negative
C122	3A 06		JRC P	②	if number is decimal, jump if hex, continue
*C124	5B		POP		get binary value
C125	74 47		ADIA	47	add ALPHA offset
C127	2C 04		JR P	③	
*C129	5B	②	POP		get binary value
C12A	74 40		ADIA	40	add NUMERAL offset
C12C	26	③	IYS		store HEX CHAR in Y$
C12D	37		RTN		return to code call

The next two commands (lines C10C—C10F) get the contents of the window and place them in the accumulator. The command PUSH is used to preserve a copy of the byte on the system stack. Care must be taken not to execute a RTN command before POP-ing the value back off the stack so that the RTN will not inadvertently place the byte in the PC and attempt to execute it.

The command SWP exchanges the high and low nibbles of the byte in the accumulator. A subroutine (lines C11D—C12D) will convert the low nibble in the accumulator to a hexadecimal character. Thus we use SWP to set up the high nibble for conversion.

The command CALL is used for subroutine calls in external RAM (CAL can be used for calls below 1FFF). CALL, much like the BASIC command, expects an absolute address to to jump to. When hand-compiling, a two-byte space must be left after the "78" Op Code of CALL. Later, when the location of the subroutine is known, the blank bytes can be filled with the address of the subroutine. Here, the CALLs in lines C112 and C116 start execution of the code at line C11D. The first CALL converts the high nibble to its character representation. The following POP instruction gets the copy of the byte we saved on line C110 and is used by the second CALL to convert the low nibble to its hexadecimal equivalent. The next two lines (C119—C11B) place a null (00) in Y$ after the two hex characters (stored during the subroutine CALL). The null indicates the end of the character string. After this instruction is executed, a RTN sends the program back to BASIC.

The subroutine from line C11D to line C12D performs the conversion itself. The nibble must be tested to see whether it can be represented by a digit or should be represented by a character. If it is in the range 0—9, adding a hex 40 will put it in the range 40—49 which corresponds to the digits 0—9 in PC-1250/1251 (1250A) code. If the nibble is in the range 10—15, then adding hex 47 will put it in the range 51—56 which corresponds to the letters A—F.

The command ANIA in line C11D is used to mask off the top nibble. A copy is saved on the stack and the copy in the accumulator is used for the range test. A 0A was subtracted because of the effect of the carry Flag. If the number in the accumulator is 0A or more, then the substraction will not result in a carry. If the value in the accumulator is 09 or less, then the Carry Flag would be set.

Jump commands test a flag before deciding whether or not to execute. In line C122, the command JRC tests the Carry Flag. If the Carry Flag is set, i.e., the result of the subtraction is less than FF, then the program jumps to line C129 indicated here by the label 2. There are two types of jumps, absolute jumps and relative jumps. If the absolute jump is

used, then two bytes following the Op Code would have to be reserved for the address from which the program is to continue execution. If the jump relative form is used, as here, then only a single byte is reserved. The content of that byte is added (or subtracted) to (from) the PC before execution continues. In this case, a single byte was reserved for the relative jump. When the address of the destination of the jump became known (C129), subtracting it from the address of the reserved byte (129–C123=6) gives us the number of bytes to jump. This value is placed in the reserved byte.

Relative jumps are trickier to calculate and program, and are more likely to cause bugs than absolute jumps. A good technique is to first write a program with only absolute jumps. After the program has been tested and is known to run properly, then relative jumps can be substituted. The advantage of relative jumps are that they execute a little faster and that they need not be changed when a program has been moved in memory.

Following a jump command in line C122, the nibble which was saved on the stack is POPed back into the accumulator and either a 40 or 47 hex is added to it, depending on whether it is a digit or a letter. The two branches join back together on line C12C where the calculated value is stored in Y$. The RTN here returns us to the statement after the CALL on line C112 or C116.

This program is integrated with a calling BASIC program as follows: the value placed in X from the PEEK command is POKED into the window at C6A0, then a CALL is made to &C100. The result which is left in Y$ can then be printed.

```
10:  INPUT "ADDR?"; A
20:  INPUT "BYTES?";3
30:  FOR I=0 TO (3-I)
40:  X= PEEK (A+I)
50:  POKE &C6A0,X
60:  CALL &C100
70:  PRINT (A+I);"       ";Y$
80:  NEXT I
90:  END
```

The hand compiled program source code is as follows:

```
400:  POKE &C100,&12,&06,&
      02,&D7,&DB,&50
410:  POKE &C106,&02,&C5,&
      DB,&02,&F5,&26
420:  POKE &C10C,&10,&C6,&
      A0,&57,&34,&58,&78,&
      C1,&1D
430:  POKE &C114,&5B,&78,&
      C1,&1D,&02,&00,&26,&
      37
440:  POKE &C11D,&64,&0F,&
      34,&75,&0A,&3A,&06
450:  POKE &C123,&5B,&74,&
      47,&2C,&04
460:  POKE &C128,&5B,&74,&
      40,&26,&37
470:  END
```

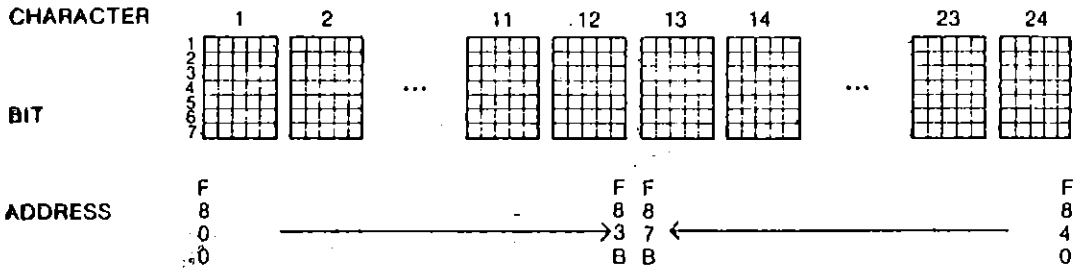
Once this program has been executed, it can be erased from memory and the space can be used for other purposes. Thus, the machine code version both runs much faster and takes up less space (not counting the bytes used at C100) than the BASIC version.

This example is much larger than the previous one. However, if you study the actions of each command and keep in mind the memory maps, then you should be able to master the example. Many of the most commonly used commands are illustrated here, and can serve as models for your own programming.

mple Program #3 — Accessing the LCD play

The following brief program is used to illustrate acess to the LCD display. The LCD display is memory mapped as follows:

LCD Display and Corresponding Memory Locations



Each of the 24 characters on the LCD screen is represented by 5 bytes of memory. A one in a bit position will activate the corresponding position on the screen. The first character on the screen is controlled by memory positions F800—F804, the next by F805—F809 and so on, up to character number 12 which is controlled by positions F837—F83B. The second 12 characters are controlled in descending order by memory positions F840—F87B, character 24 by F840—F844, character 23 by F845—F849, down to character 13 by F877—F87B.

These addresses are also used for evaluating numerical expressions. When a computation is performed, the screen is turned off and the memory locations are used as a temporary data buffer for numerical calculations. After the calculations are complete, the area is erased and the display is turned back on.

The LCD display is controlled by the low order bit of internal RAM address 5F. The display can be turned on as follows:

Addr	Machine Code	Mnemonic	Operand(s)
C100	12 5F	LIP	5F
C102	61 1	ORIM	1
C104	DF	OUTC	
C105	37	RTN	

This code can be CALLed by BASIC, as in the following program starting at line 52

```
500:  POKE &C100,&12,&5F,&
      61,&01,&DF,&37
510:  END
520:  PRINT " "
530:  CALL &C100
540:  POKE &F800,&01,&02,&
      03,&04,&05
550:  FOR I=1 TO 100: NEXT
      I
560:  END
```

The PRINT statement in line 520 erases the display. The CALL in line 530 turns t display on. The POKE command in 540 places the data on the screen. The FOR NE loop in 550 is used as a delay to hold the display on. The display is automatically clear when the basic program ends.

HE PC-1250/1251 (1250A) Instruction List

There are 115 machine language instructions for the PC-1250/1251 (1250A) included here. The instructions may occur in the following sizes and formats:

1. **1 byte instructions**
   - A. 8 bit operation code
   - B. 2 bit operation code, 6 bit operand
2. **2 byte instructions**
   - A. 8 bit operation code, 8 bit operand
   - B. 7 bit operation code, 9 bit operand
   - C. 3 bit operation code, 13 bit operand
3. **3 byte instructions — 8 bit operation code, 16 bit operand**
4. **More than 3 byte instructions — 8 bit operation code 3 or more byte operand**

Detailed information about each instruction is given in this section of the manual. At the end of the section, the summarized information is listed alphabetically and by hexadecimal operation code. In the instruction detail list, the instructions are grouped by similarity of function and the following information is provided for each:

1. **Machine Language mnemonic code**
2. **A description and diagram of the actions performed by the instruction**
3. **The number of bytes required**
4. **The number of bits in the operation code and the operand(s)**
5. **The appearance of the instruction in memory in binary and hexadecimal**
6. **Cycles —** The number of machine cycles required for execution of the instruction. This information can be used to select the faster instructions if more than one instruction or set of instructions could be used to obtain the desired results.
7. **Flags —** Indicates whether the execution of the instruction will affect the Carry (C) or the Zero (Z) flag.
8. **Other—** Indicates other changes that may occur during the execution of the instruction, such as changes in the contents of registers.

Abbreviations used in the instruction descriptions

Registers

A	Accumulator
B	Extra Accumulator
DP	Data Pointer — External RAM address
I	Block Operation Register
J	Block Operation Register
K	General Purpose — For programmers use
L	General Purpose — For programmers use
P	Internal RAM address pointer
PC	Program Counter
Q	Internal RAM address pointer
R	Stack Pointer
X	External RAM address pointer
Y	External RAM address pointer

If the letter appears as above, the **contents of the register** is being specified.

If the letter appears within parentheses, the **contents of the memory address** that is stored in the register is being specified.

Operands

ℓ	6 bit literal = an address in internal RAM between 00 and 3F.
n	7 bit literal value (for registers P and Q) 8 bit literal value (for other registers)
nm	16 bit literal value

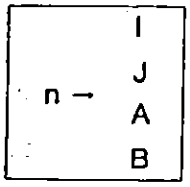
1. Move Data Instructions

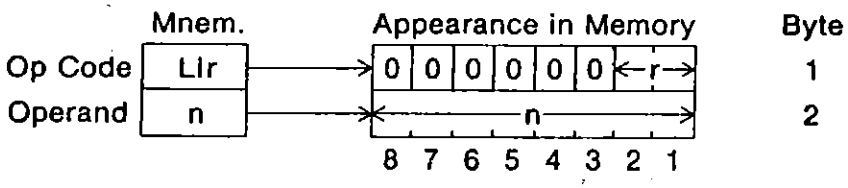
1.1 Load Immediate — The value represented by the operand (n, m, l) is moved into the specified register.

1.1.1 LIr n Load the value of n into r(egister).

r = an 8 bit register

n = 8 bits





If r =	2 Low Bits of Op Code	Hex. Op Code
I	0 0	00
J	0 1	01
A	1 0	02
B	1 1	03
	2 1	

Cycles: 4
Flags: None
Other: None

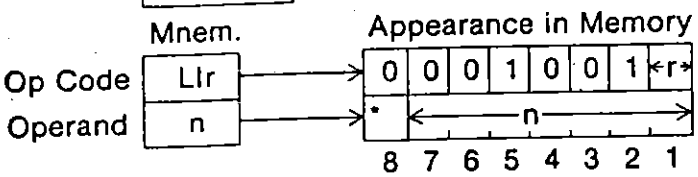
1.1.2 LIr n

Load the value of n into r(egister).

r = a 7 bit internal RAM address register

n = 7 bits

n → P, Q



If r =	Low Bits of Op Code	Hex. Op Code
P	0	12
Q	1	13

1

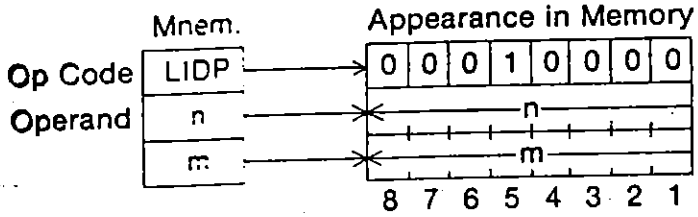
Cycles: 4
Flags: None
Other: None

1.1.3 LIDP nm

Load the value of nm into the 16 bit DP register.

nm = 16 bits

n → DP_H, m → DP_L

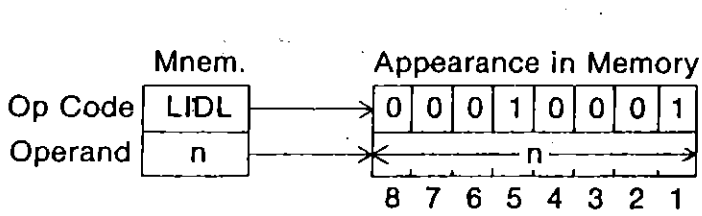


Cycles: 8
Flags: None
Other: None

1.1.4 LIDL n

Load the value of n into the low order byte of the DP register.

n → DP_L



	Hex. Op Code	Byte
Op Code	11	1
Operand		2

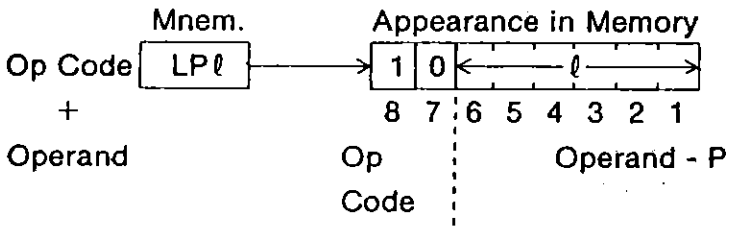
Cycles: 5
Flags: None
Other: None

1.1.5 LP ℓ

Load the value of ℓ into the P register.

ℓ = 6 bits (00—3F)

ℓ → P



	Hex. Op Code	Byte
Op Code + Operand	80+ℓ	1

Cycles: 2
Flags: None
Other: None

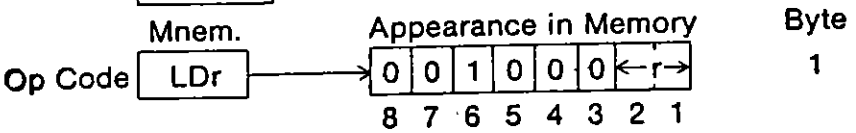
1.2 Load/Store a register into/from the accumulator

1.2.1 LDr

Load the contents of r(egister) into the accumulator (register A).

r = a 7 bit internal RAM address register

P
Q → A
R



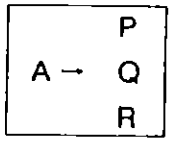
If r =	2 Low Bits of Op Code	Hex. Op Code
P	0 0	20
Q	0 1	21
R	1 0	22
	2 1	

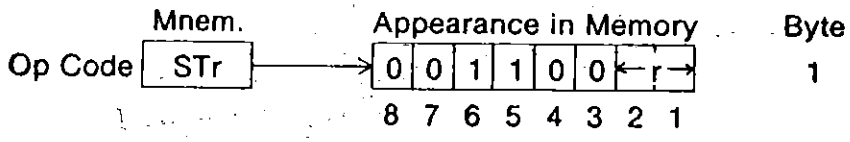
Cycles: 2
Flags: None
Other: None

1.2.2 STr

Store the contents of the accumulator (register A) into r(egister).

r = a 7 bit internal RAM address register





If r =	2 Low Bits of Op Code	Hex. Op Code
P	0 0	30
Q	0 1	31
R	1 0	32
	2 1	

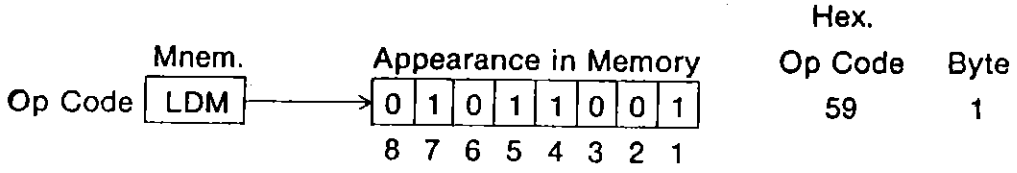
Cycles: 2
Flags: None
Other: None

1.3 Move data between memory and the accumulator — Move the contents of the accumulator or of the address in a register to/from the address in a register of the accumulator.

1.3.1 LDM

Load the contents of the address in the P register into the accumulator.

(P) → A

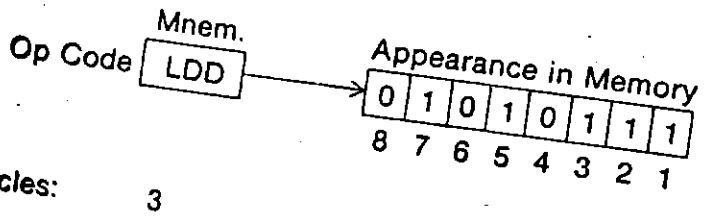


Cycles: 2
Flags: None
Other: None

1.3.2 LDD

Load the contents of the address in the DP register into the accumulator.

(DP) → A

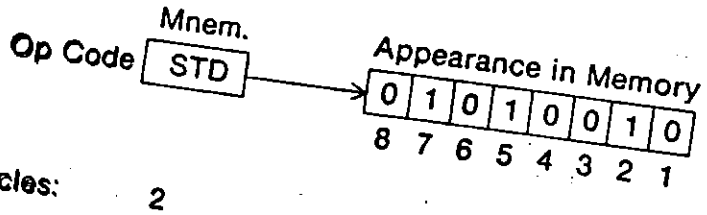


Cycles: 3
Flags: None
Other: None

1.3.3 STD

Store the contents of the accumulator in the address in the DP register.

A → (DP)

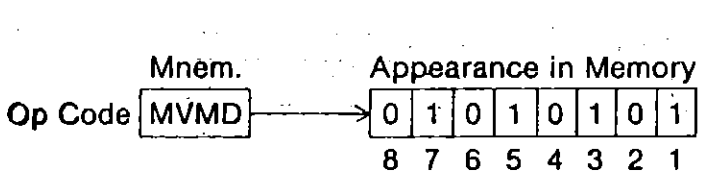


Cycles: 2
Flags: None
Other: None

1.4 Move data from one memory address to another — Move the contents of the address in a register to the address in another register.

1.4.1 MVMD Move the contents of the DP register address to the P register address.

(DP) → (P)

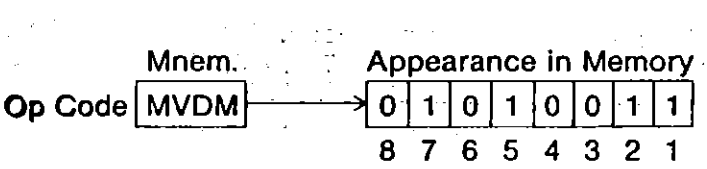


Hex. Op Code	Byte
55	1

Cycles: 3
Flags: None
Other: None

1.4.2 MVDM Move the contents of the P register address to the DP register address.

(P) → (DP)



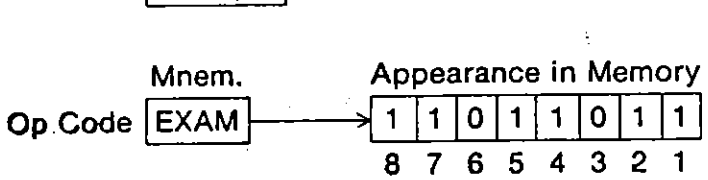
Hex. Op Code	Byte
53	1

Cycles: 3
Flags: None
Other: None

1.5 Exchange data between two registers — Exchange the data in the accumulator with that in the address in the DP register or register B.

1.5.1 EXAM Exchange the contents of the address in register P with the contents of the accumulator.

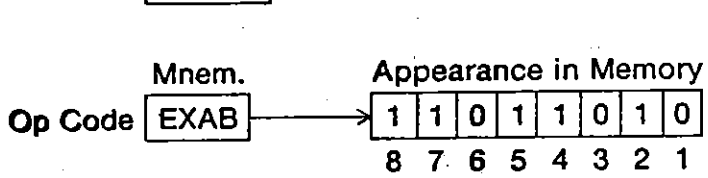
A ← (P)



Cycles: 3
Flags: None
Other: None

1.5.2 EXAB Exchange the contents of register B with the contents of the accumulator.

A ← B



Cycles: 5
Flags: None
Other: None

1.6 Block move of data in memory — Move the contents of one or more bytes of memory to another area of memory.

1.6.1 MVW
MVB

Move the contents of d+1 bytes starting with the address in register Q into the d+1 bytes starting with the address in register P.

d must be stored in register I or J

if d = 0, 1 byte will be moved

```
I
J  → d,
REPEAT
     (Q) → (P), P+1, Q+1, d−1
UNTIL d=FF
```

If:

Mnem.	Appearance in Memory	Hex. Op Code	Byte
MVW	0 0 0 0 1 0 0 0 (8 7 6 5 4 3 2 1)	08	1

Then: d = the value stored in register I

If:

Mnem.	Appearance in Memory	Hex. Op Code	Byte
MVB	0 0 0 0 1 0 1 0 (8 7 6 5 4 3 2 1)	0A	1

Then: d = the value stored in register J

Cycles: 5 + 2d
Flags: None
Other: P and Q registers are incremented

1.6.2 MVWD
MVBD

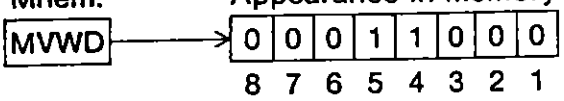
Move the contents of d+1 bytes starting with the address in the DP register into the d+1 bytes starting with the address in the P register.

d must be stored in register I or J

if d = 0, 1 byte will be moved

```
I
J  ← d,
REPEAT
     (DP) ← (P), DP+1, P+1, d−1
UNTIL d=FF
```

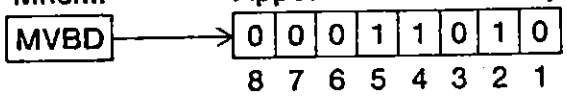
If:



Hex. Op Code	Byt
18	1

Then: d = the value stored in register I

If:



Hex. Op Code	Byt
1A	1

Then: d = the value stored in register J

Cycles: 5 + 4d
Flags: None
Other: P and DP registers are incremented

1.7 Block exchange of data in memory — Exchange the contents of one or more bytes of memory with the contents of the same number of bytes in another area of memory.

1.7.1 EXW
EXB

Exchange the contents of d+1 bytes starting with the address in the Q register with the contents of the d+1 bytes starting with the address in the P register.
d must be stored in register I or J
if d = 0, 1 byte will be exchanged

```
I
J  → d,
REPEAT
    (P) ↔ (Q), P+1, Q+1, d−1
UNTIL d=FF
```

If:

Mnem.	Appearance in Memory	Hex. Op Code	Byte
EXW	0 0 0 0 1 0 0 1 (8 7 6 5 4 3 2 1)	09	1

Then: d = the value stored in register I

If:

Mnem.	Appearance in Memory	Hex. Op Code	Byte
EXB	0 0 0 0 1 0 1 1 (8 7 6 5 4 3 2 1)	0B	1

Then: d = the value stored in register J

Cycles: 6 + 3d
Flags: None
Other: P and Q registers are incremented

1.7.2 EXWD
EXBD

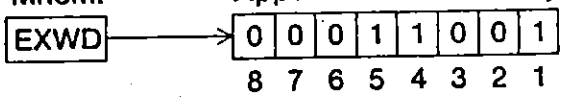
Exchange the contents of d+1 bytes starting at the address in the DP register with the contents of the d+1 bytes starting at the address in the P register.

d must be stored in register I or J

if d = 0, 1 byte will be exchanged

```
I
J  ← d,
REPEAT
    (DP) ↔ (P), DP+1, P+1, d−1
UNTIL d=FF
```

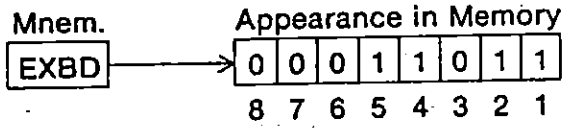
If:



Hex. Op Code	Byte
19	1

Then: d = the value stored in register I

If:



Hex. Op Code	Byte
1B	1

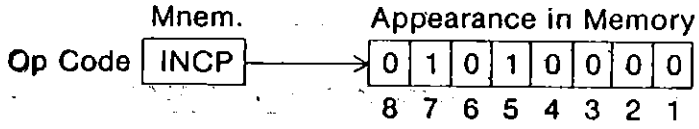
Then: d = the value stored in register J

Cycles: 7 + 6d
Flags: None
Other: P and DP registers are incremented

1.8 Increment or decrement a register — Add or subtract 1 from the contents of the register specified by the instruction.

1.8.1 INCP Add 1 to the contents of register P.

P + 1 → P

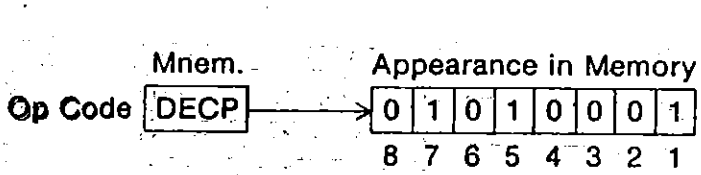


Hex. Op Code	Byte
50	1

Cycles: 2
Flags: None
Other: None

1.8.2 DECP Subtract 1 from the contents of register P.

P − 1 → P

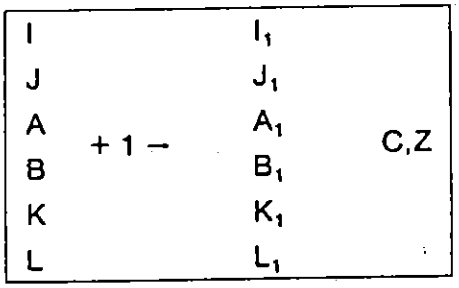


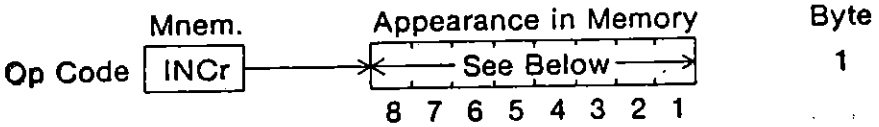
Hex. Op Code	Byte
51	1

Cycles: 2
Flags: None
Other: None

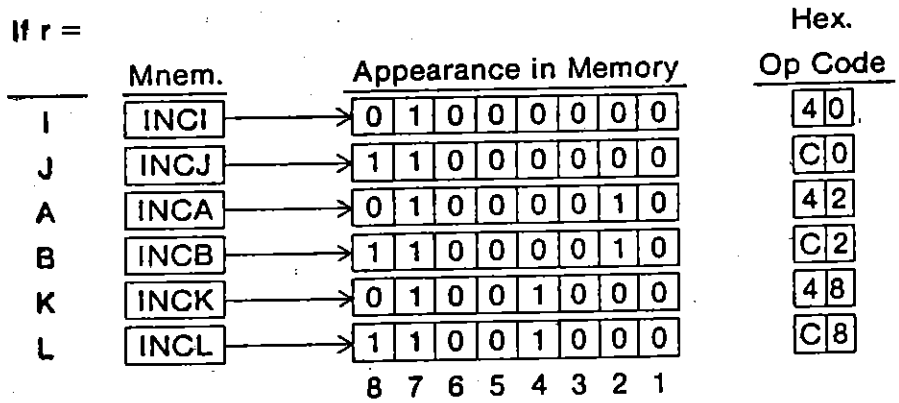
1.8.3 INCr

Increment the contents of r(egister) by 1.



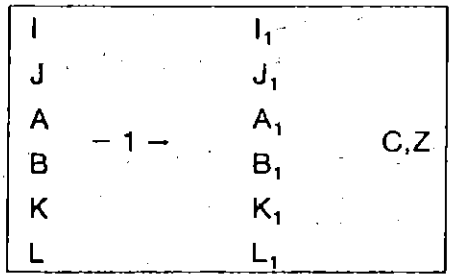


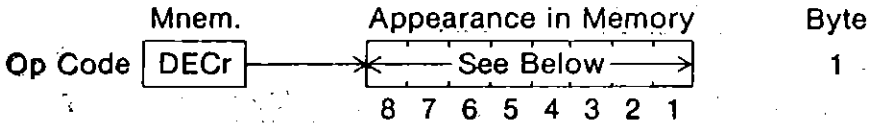
Cycles: 4
Flags: C, Z
Other: Contents of Q register change



1.8.4 DECr

Decrement the contents of r(egister) by 1.





Cycles: 4

Flags: C, Z

Other: Q register changes

If r =	Mnem.	Appearance in Memory	Hex. Op Code
I	DECI	0 1 0 0 0 0 0 1	4 1
J	DECJ	1 1 0 0 0 0 0 1	C 1
A	DECA	0 1 0 0 0 0 1 1	4 3
B	DECB	1 1 0 0 0 0 1 1	C 3
K	DECK	0 1 0 0 1 0 0 1	4 9
L	DECL	1 1 0 0 1 0 0 1	C 9

8 7 6 5 4 3 2 1

1.9 Increment or decrement an external memory address register and move the address from the register to the DP register — Add or subtract 1 to or from the address in register X or Y, move the contents of X or Y to the DP register.

1.9.1 Ir Add 1 to the memory address in r(egister) and store the incremented address in the DP register.

```
X
Y   → DP,
DP + 1 → DP, X Y
```

If r =	Mnem.	Appearance in Memory	Hex. Op Code	Byte
X	IX	0 0 0 0 1 0 0 0 (8 7 6 5 4 3 2 1)	04	1
Y	IY	0 0 0 0 1 0 1 0 (8 7 6 5 4 3 2 1)	06	1

Cycles: 6
Flags: None
Other: Q register changes

1.9.2 Dr Subtract 1 from the memory address in r(egister) and store the decremented address in the DP register.

```
X
Y   → DP,
DP − 1 → DP, X Y
```

If r =	Mnem.	Appearance in Memory	Hex. Op Code	Byte
X	DX	0 0 0 0 0 1 0 1 (8 7 6 5 4 3 2 1)	05	1
Y	DY	0 0 0 0 0 1 1 1 (8 7 6 5 4 3 2 1)	07	1

Cycles: 6
Flags: None
Other: Q register changes

1.10 Increment or decrement register X and load the contents of the register X address into the accumulator — Add or subtract 1 from or to the address in register X, load the new address into the DP register and load the contents of new address into the accumulator.

1.10.1 IXL

a. Add 1 to the address in register X.
b. Load the incremented address into the DP register.
c. Move the contents of the address in the DP register into the accumulator.

```
X → DP
DP + 1 → DP, X
(DP) → A
```

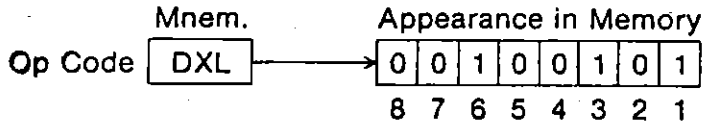
	Mnem.	Appearance in Memory	Hex. Op Code	Byte
Op Code	IXL	0 0 1 0 0 1 0 0 (8 7 6 5 4 3 2 1)	24	1

Cycles: 7
Flags: None
Other: Q register changes

1.10.2 DXL

a. Subtract 1 from the address in register X.
b. Load the decremented address into the DP register.
c. Move the contents of the address in the DP register into the accumulator.

```
X → DP
DP − 1 → DP, X
(DP) → A
```



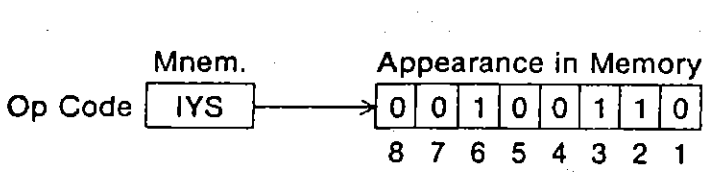
Cycles: 7
Flags: None
Other: Q register changes

1.11 Increment or decrement register Y and store the contents of the accumulator into the address in the Y register — Add or subtract 1 to or from the address in register Y, load the new address into the DP register and store the contents of the accumulator into the new address.

1.11.1 IYS

a. Add 1 to the address in register Y.
b. Load the incremented address into the DP register.
c. Move the contents of the accumulator into the DP register address.

```
Y → DP
DP + 1 → DP, Y
A → (DP)
```

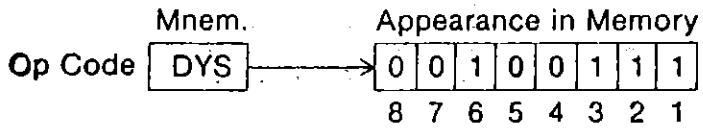


Cycles: 6
Flags: None
Other: Q register changes

1.11.2 DYS

a. Subtract 1 from the address in register Y.
b. Load the decremented address into the DP register.
c. Move the contents of the accumulator into the DP register memory address.

```
Y → DP
DP − 1 → DP, Y
A → (DP)
```



	Hex. Op Code	Byte
Op Code	27	1

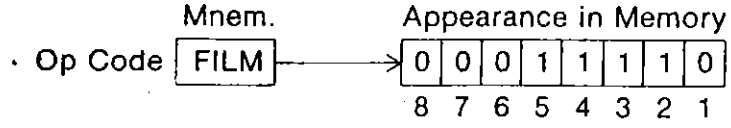
Cycles: 6
Flags: None
Other: Q register changes

1.12 Fill a block of memory with a single value — Fill either an intenal RAM or an external memory block with the value in the accumulator.

1.12.1 FILM

Store the value in the accumulator into the d+1 bytes of the internal RAM starting with the address in the P register.
d must be stored in register I
If d = 0, one byte will be filled

```
I → d
REPEAT
  A → (P), P+1, d−1
UNTIL d=FF
```



	Hex. Op Code	Byte
Op Code	1E	1

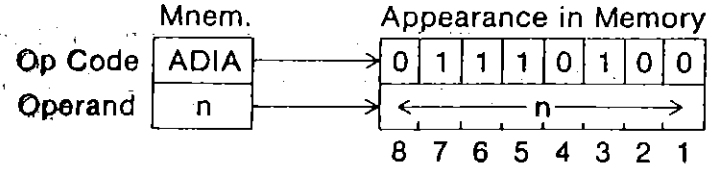
Cycles: 5 + d
Flags: None
Other: P register changes

2. Arithmetic, Logical and Shift Instructions

2.1 Add/Subtract Immediate, Accumulator — Add or subtract the value n to/from the accumulator.

2.1.1 ADIA n Add the value n to the accumulator.

A + n → A C,Z

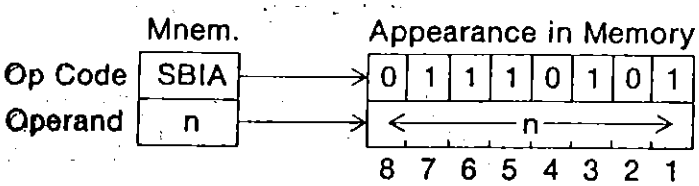


	Hex. Op Code	Byte
	74	1
		2

Cycles: 4
Flags: C, Z
Other: None

2.1.2 SBIA n Subtract the value n from the accumulator.

A − n → A C,Z



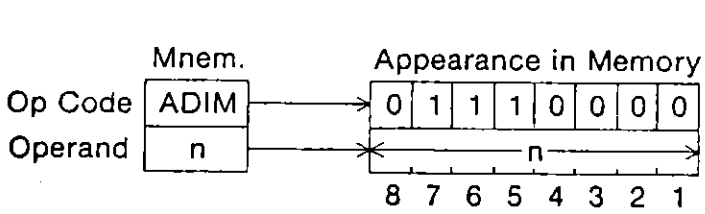
	Hex. Op Code	Byte
	75	1
		2

Cycles: 4
Flags: C, Z
Other: None

2.2 Add/Subtract Immediate, Memory — Add or subtract the value n to/from the register P memory address.

2.2.1 ADIM n Add the value n to the register P memory address.

(P) + n → (P) C,Z

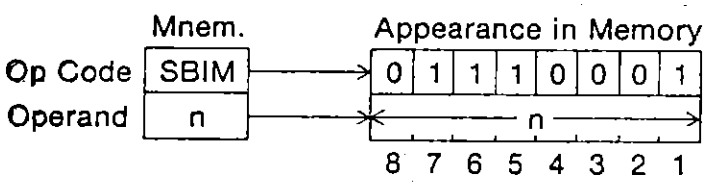


	Hex. Op Code	Byte
Op Code	70	1
Operand		2

Cycles: 4
Flags: C, Z
Other: None

2.2.2 SBIM n Subtract the value n from the register P memory address.

(P) − n → (P) C,Z



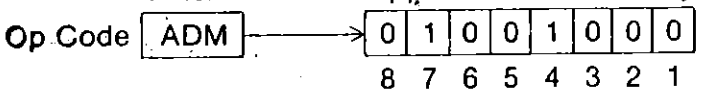
	Hex. Op Code	Byte
Op Code	71	1
Operand		2

Cycles: 4
Flags: C, Z
Other: None

2.3 1 Byte Binary Addition or Subtraction — Add or subtract the contents of the address in the P register to/from the accumulator and store the results in the address in the P register.

2.3.1 ADM Add the contents of the address in the P register to the accumulator and store the result in the address in the P register.

(P) + A → (P) C,Z

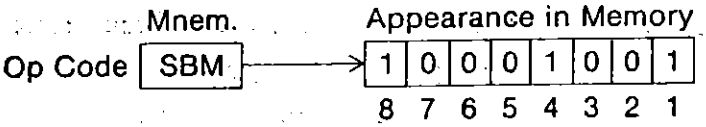


Hex. Op Code	Byte
44	1

Cycles: 3
Flags: C, Z
Other: None

2.3.2 SBM Subtract the contents of the address in the P register from the accumulator and store the result in the address in the P register.

(P) − A → (P) C,Z



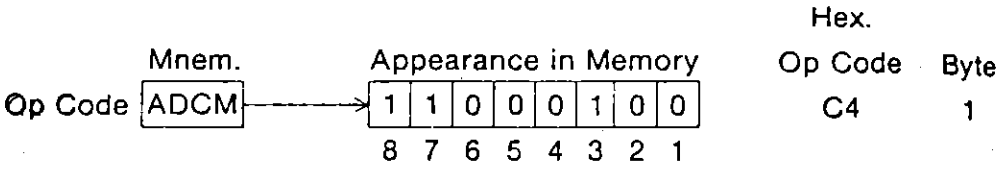
Hex. Op Code	Byte
45	1

Cycles: 3
Flags: C, Z
Other: None

2.4 1 Byte Binary Addition or Subtraction with Carry — Add or subtract the contents of the address in the P register to/from the accumulator with Carry, and store the results in the address in the P register.

2.4.1 ADCM Add the contents of the address in the P register and the Carry to the accumulator and store the result in the address in the P register.

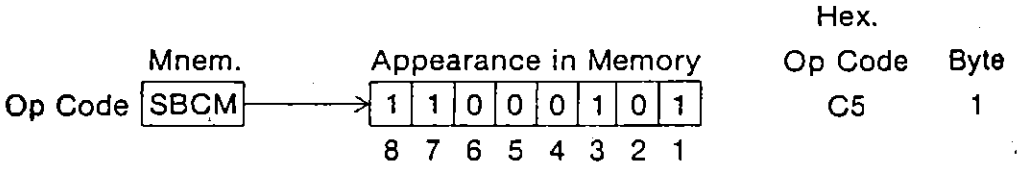
(P) + A + C → (P) C,Z



Cycles: 3
Flags: C, Z
Other: None

2.4.2 SBCM Subtract the contents of the address in the P register and the Carry from the accumulator and store the result in the address in the P register.

(P) − A − C → (P) C,Z

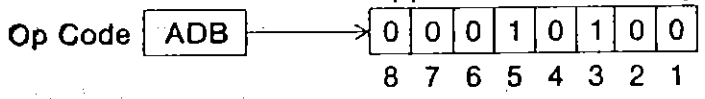


Cycles: 3
Flags: C, Z
Other: None

2.5 2 Byte Binary Addition or Subtraction — Add or subtract the contents of the address in the P register to/from registers A and B (accumulator and spare register) and store the result in the address in the P register.

2.5.1 ADB Add the contents of the address in the P register to registers A and B (16 bits) and store the result in the address in the P register.

[P + 1, P] + [BA] → [P + 1, P] C,Z

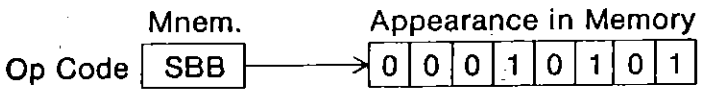


Hex. Op Code	Byte
14	1

Cycles: 5
Flags: C, Z
Other: P register changes

2.5.2 SBB Subtract the contents of the address in the P register from registers A and B (16 bits) and store the result in the address in the P register.

[P + 1, P] − [B,A] → [P + 1, P] C,Z



Hex. Op Code	Byte
15	1

Cycles: 5
Flags: C, Z
Other: P register changes

2.6 Block BCD Addition and Subtraction — Add or subtract d+1 bytes starting with the address in the P register to from the accumulator or the address in the Q register.

2.6.1 ADN Add the contents of the accumulator to the block of d+1 bytes of memory starting with the address in the P register as the right-most digit.

the contents of I are stored in d

```
I → d
REPEAT
  (P) + A → (P)   (BCD), P−1, d−1
UNTIL d=FF
```

	Mnem.	Appearance in Memory	Hex. Op Code	Byte
Op Code	ADN →	0 0 0 0 1 1 0 0 (8 7 6 5 4 3 2 1)	0C	1

Cycles: 7 + 3d
Flags: C, Z
Other: P register changes

2.6.2 SBN Subtract the contents of the accumulator from the block of d+1 bytes of memory starting with the address in the P register as the right-most digit.

the contents of I are stored in d

```
I → d
REPEAT
  (P) − A → (P)   (BCD), P−1, d−1
UNTIL d=FF
```

	Mnem.	Appearance in Memory	Hex. Op Code	Byte
Op Code	SBN →	0 0 0 0 1 1 0 1 (8 7 6 5 4 3 2 1)	0D	1

Cycles: 7 + 3d
Flags: C, Z
Other: P register changes

2.6.3 ADW

Add the contents of the block of d+1 bytes of memory starting with the address in the Q register as its right-most (low order) digit to the d+1 bytes of memory starting with the address in the P register as its right-most digit.

the contents of I are stored in d

```
I → d
REPEAT
  (P) + Q → (P)   (BCD), P−1, Q−1, d−1
UNTIL d=FF
```

Op Code	Mnem.	Appearance in Memory	Hex. Op Code	Byte
	ADW →	0 0 0 0 1 1 1 0 (8 7 6 5 4 3 2 1)	0E	1

Cycles: 7 + 3d
Flags: C, Z
Other: P & Q registers change

2.6.4 SBW

Subtract the contents of the block of d+1 bytes of memory starting with the address in the Q register as its right-most (low order) digit from the d+1 bytes of memory starting with the address in the P register as its right-most digit.

the contents of I are stored in d

```
I → d
REPEAT
  (P) − Q → (P)   (BCD), P−1, Q−1, d−1
UNTIL d=FF
```

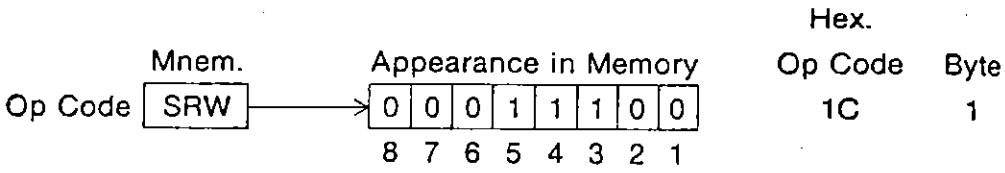
Op Code	Mnem.	Appearance in Memory	Hex. Op Code	Byte
	SBW →	0 0 0 0 1 1 1 1 (8 7 6 5 4 3 2 1)	0F	1

Cycles: 7 + 3d
Flags: C, Z
Other: P & Q registers change

2.7 Block Shift 4 Bits — Shift 4 bits of d+1 bytes starting with the address in the P register 4 bits to the right or left.

2.7.1 SRW Shift d+1 bytes 4 bits (one BCD digit) to the right starting with the address in the P register.
the contents of I are stored in d

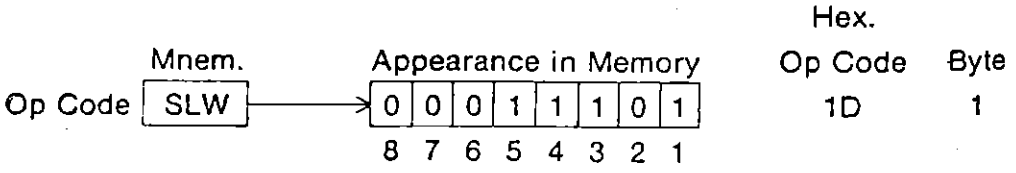
```
I − d
REPEAT
  Shift P 4 bits to right, P+1, d−1
UNTIL d=FF
```



Cycles: 5 + d
Flags: None
Other: P register changes

2.7.2 SLW Shift d+1 bytes 4 bits (one BCD digit) to the left starting with the address in the P register.
the contents of I are in d

```
I − d
REPEAT
  Shift P 4 bits to left, P−1, d−1
UNTIL d=FF
```

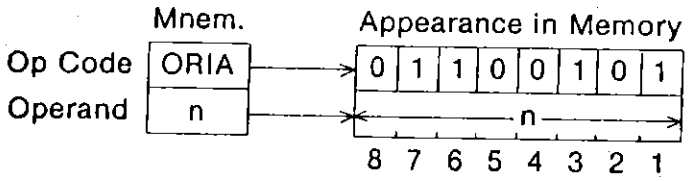


Cycles: 5 + d
Flags: None
Other: P register changes

2.8 Logical OR — OR the contents of the accumulator or a memory location with an immediate value OR the contents of an address with the accumulator.

2.8.1 ORIA n OR the contents of the accumulator with the immediate value n.

$A \vee n \rightarrow A, Z$

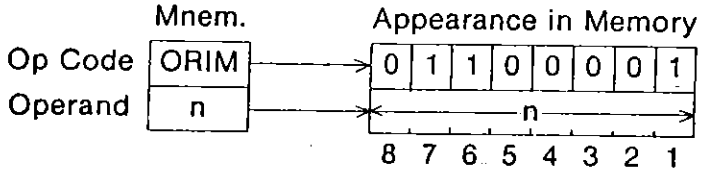


	Hex. Op Code	Byte
	65	1
		2

Cycles: 4
Flags: Z
Other: None

2.8.2 ORIM n OR the contents of the address in the P register with the immediate value n.

$(P) \vee n \rightarrow (P), Z$

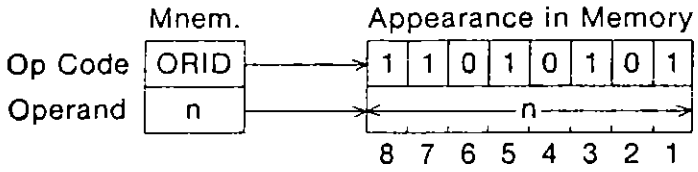


	Hex. Op Code	Byte
	61	1
		2

Cycles: 4
Flags: Z
Other: None

2.8.3 ORID n OR the contents of the address in the DP register with the immediate value n.

(DP) ∨ n → (DP), Z

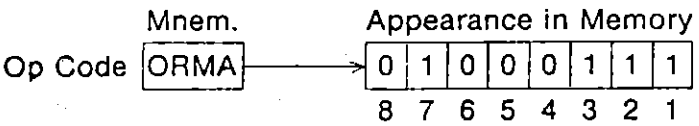


	Hex. Op Code	Byte
Op Code	D5	1
Operand		2

Cycles: 6
Flags: Z
Other: R–1 used for temporary storage

2.8.4 ORMA OR the contents of the address in the P register with the contents of the accumulator.

(P) ∨ A → (P), Z



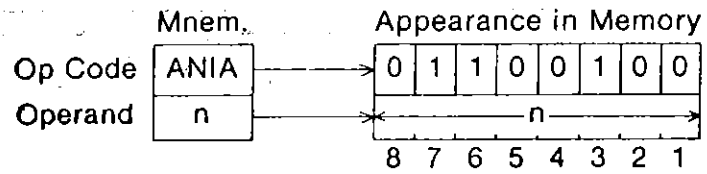
	Hex. Op Code	Byte
Op Code	47	1

Cycles: 3
Flags: Z
Other: None

2.9 Logical AND — AND the contents of the accumulator or a memory location with an immediate value AND the contents of an address with the accumulator.

2.9.1 ANIA n AND the contents of the accumulator with the immediate value n.

$A \wedge n \rightarrow A, Z$

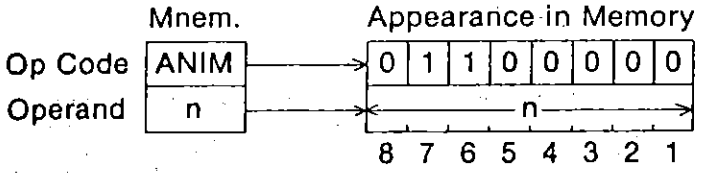


Hex. Op Code	Byte
64	1
	2

Cycles: 4
Flags: Z
Other: None

2.9.2 ANIM n AND the contents of the address in the P register with the immediate value n.

$(P) \wedge n \rightarrow (P), Z$



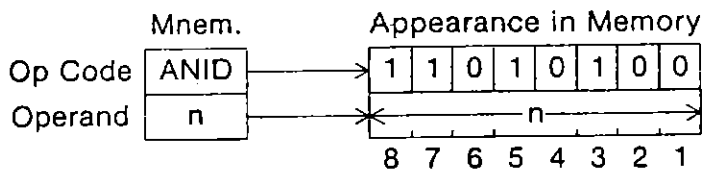
Hex. Op Code	Byte
60	1
	2

Cycles: 4
Flags: Z
Other: None

2.9.3 ANID n

AND the contents of the address in the DP register with the immediate value n.

(DP) ∧ n → (DP), Z



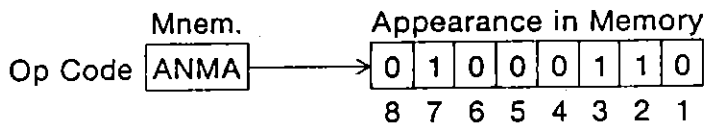
Hex. Op Code	Byte
D4	1
	2

Cycles: 6
Flags: Z
Other: R–1 used for temporary storage

2.9.4 ANMA

AND the contents of the address in the P register with the contents of the accumulator.

(P) ∧ A → (P), Z



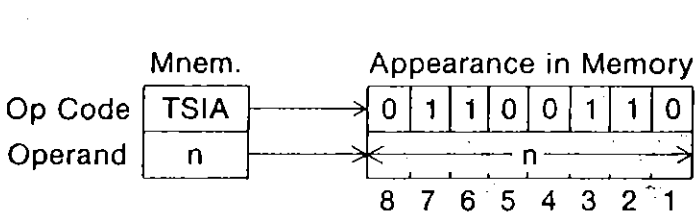
Hex. Op Code	Byte
46	1

Cycles: 3
Flags: Z
Other: None

2.10 Bit Test Immediate — AND the contents of the accumulator or a memory position with the value n, store the result in the Zero Flag.

2.10.1 TSIA n AND the contents of the accumulator with the value n and store the results in the Zero Flag.

A ∧ n Z

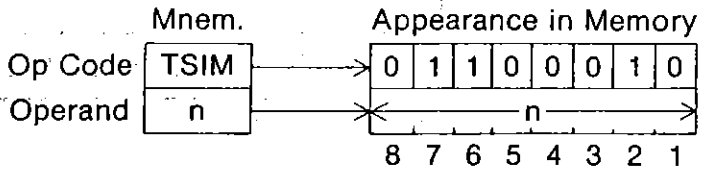


	Hex. Op Code	Byte
Op Code	66	1
Operand		2

Cycles: 4
Flags: Z
Other: None

2.10.2 TSIM n AND the contents of the address in the P register with the value n and store the result in the Zero Flag.

(P) ∧ n Z

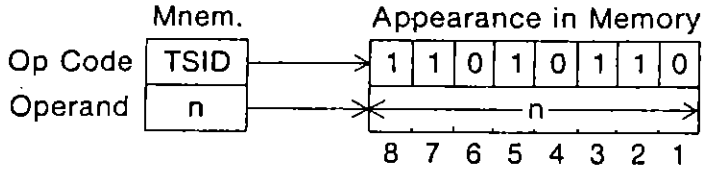


	Hex. Op Code	Byte
Op Code	62	1
Operand		2

Cycles: 4
Flags: Z
Other: None

2.10.3 TSID n AND the contents of the DP register address with the value n and store the result in the Zero Flag.

(DP) ∧ n Z



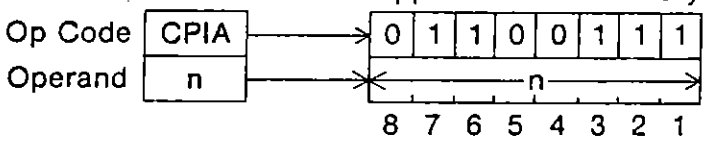
	Hex. Op Code	Byte
Op Code	D6	1
Operand		2

Cycles: 6
Flags: Z
Other: R−1 used for temporary storage

2.11 Compare Immediate — Compare the accumulator or memory location with the immediate value n.
Compare the contents of internal memory with the accumulator.

2.11.1 CPIA n Compare the accumulator with the immediate value n.

A − n C,Z



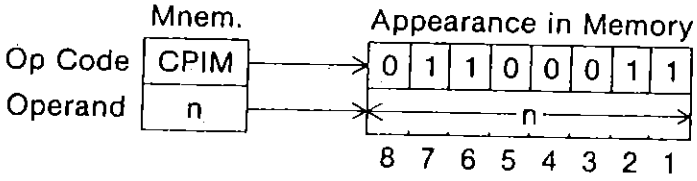
	Hex. Op Code	Byte
Op Code	67	1
Operand		2

Cycles: 4
Flags: A<n C=1 Z=0
A=n C=0 Z=1
A>n C=0 Z=0
Other: None

2.11.2 CPIM n

Compare the contents of the address in the P register with the immediate value n.

(P) − n C,Z



	Hex. Op Code	Byte
	63	1
		2

Cycles: 4

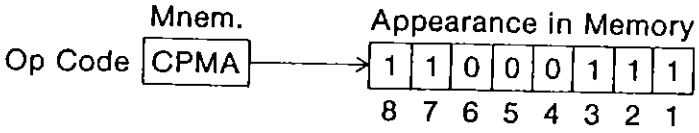
Flags: (P)<n C=1 Z=0
(P)=n C=0 Z=1
(P)>n C=0 Z=0

Other: None

2.11.3 CPMA

Compare the contents of the address in the P register to the accumulator.

(P) − A C,Z



	Hex. Op Code	Byte
	C7	1

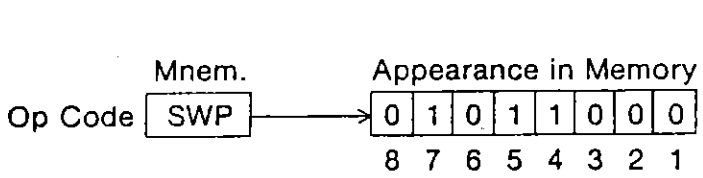
Cycles: 3

Flags: (P)<A C=1 Z=0
(P)=A C=0 Z=1
(P)>A C=0 Z=0

Other: None

2.12 SWP Exchange the contents of the 4 right-most bits of accumulator with the contents of the 4 left-most bits.

$A_{1-4} \leftrightarrow A_{5-8}$

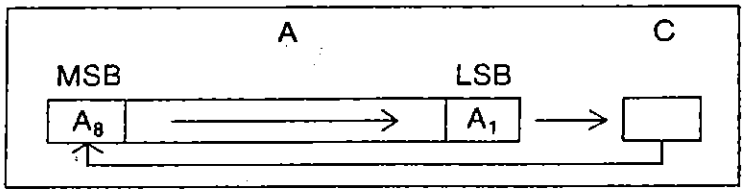


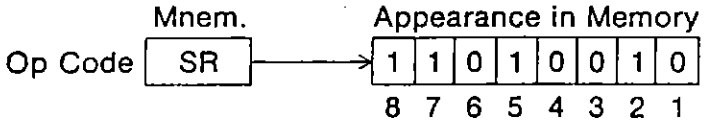
Hex. Op Code	Byte
58	1

Cycles: 2
Flags: None
Other: None

2.13 Shift Bits of a Byte — Shift the 8 bits of a byte 1 bit position to the right or left.

2.13.1 SR Shift the 8 bits of a byte 1 bit position to the right. The original LSB goes to the Carry Flag, original content of the Carry Flag goes to the byte's MSB.

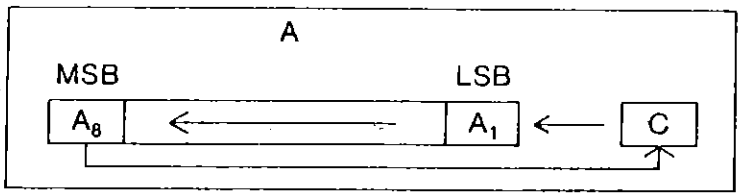




Hex. Op Code	Byte
D2	1

Cycles: 2
Flags: C
Other: None

2.13.2 SL Shift the 8 bits of a byte 1 bit position to the left. The original MSB goes to the Carry Flag, original content of the Carry Flag goes to the byte's LSB.

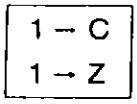


	Mnem.	Appearance in Memory	Hex. Op Code	Byte
Op Code	SL →	0 1 0 1 1 0 1 0 (8 7 6 5 4 3 2 1)	5A	1

Cycles: 2
Flags: C, Z
Other: None

2.14 Set or Reset Carry Flag — Set the Carry Flag to either 0 or 1.

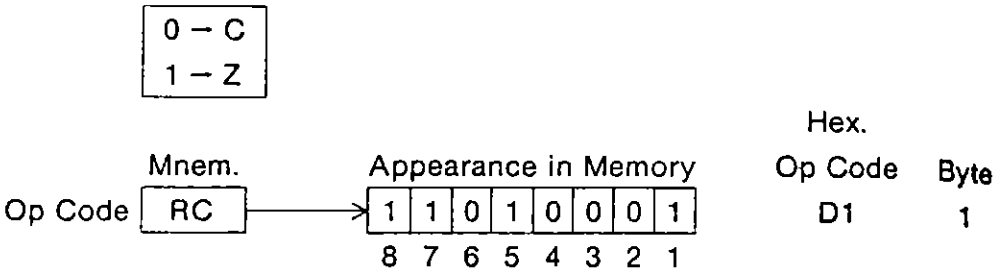
2.14.1 SC Store a 1 in the Carry Flag. Note that Zero flag is set.



	Mnem.	Appearance in Memory	Hex. Op Code	Byte
Op Code	SC →	1 1 0 1 0 0 0 0 (8 7 6 5 4 3 2 1)	D0	1

Cycles: 2
Flags: C, Z is set to 1
Other: None

2.14.2 RC Store a 0 in the Carry Flag. Note that Zero flag is set.



Cycles: 2
Flags: C is set to 0
Z is set to 1
Other: None

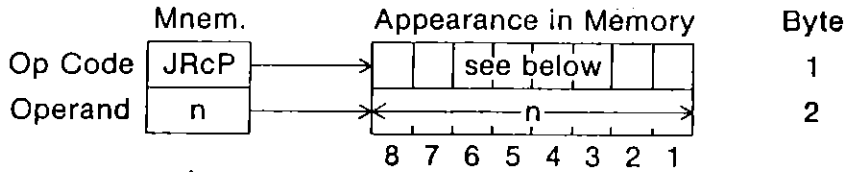
3. Jump Instructions

3.1 Jump Relative — Jump n bytes from the address of the Op Code in the plus or minus direction based on the stated conditions.

3.1.1 JRcP n Jump n bytes from the address of the Op Code in the P(lus) direction based on the stated conditions.

c = condition, see below

IF condition = true	THEN PC+n+1→PC
IF condition = false	THEN PC+2→PC



If c =	Mnem.	Appearance in Memory	Hex. Op Code
Z=1	JRZP	0 0 1 1 1 0 0 0	3 8
Z=0	JRNZP	0 0 1 0 1 0 0 0	2 8
C=1	JRCP	0 0 1 1 1 0 1 0	3 A
C=0	JRNCP	0 0 1 0 1 0 1 0	2 A
unconditional	JRP	0 0 1 0 1 1 0 0	2 C

Cycles: 7 if condition is met
4 if condition is not met

Flags: None

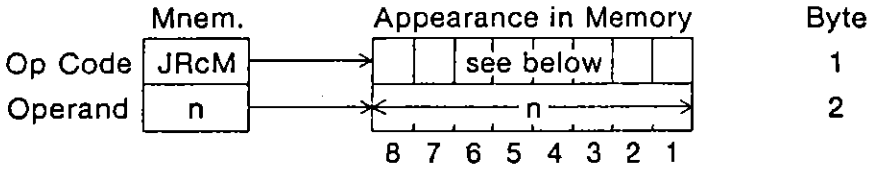
Other: None

3.1.2 JRcM n

Jump n bytes from the address of the Op Code in the M(inus) direction based on the stated conditions.

c = condition, see below

IF condition = true	THEN PC+1−n→PC
IF condition = false	THEN PC+2→PC



If c =	Mnem.	Appearance in Memory	Hex. Op Code
Z=1	JRZM	0 0 1 1 1 0 0 1	3 9
Z=0	JRNZM	0 0 1 0 1 0 0 1	2 9
C=1	JRCM	0 0 1 1 1 0 1 1	3 B
C=0	JRNCM	0 0 1 0 1 0 1 1	2 B
unconditional	JRM	0 0 1 0 1 1 0 1	2 D

Cycles: 7 if condition met
4 if condition not met

Flags: None

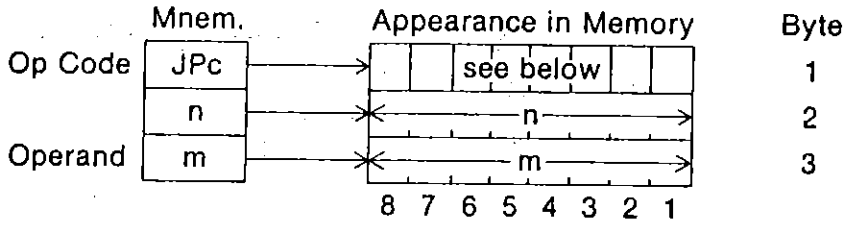
Other: None

3.2 JPc

Jump absolute —Jump to the absolute address nm based on the stated condition.

c = condition, see below

IF condition = true	THEN n→PCH	m→PCL
IF condition = false	THEN PC+3→PC	



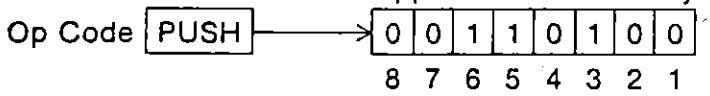
If c =	Mnem.	Appearance in Memory	Hex. Op Code
Z=1	JPZ	0 1 1 1 1 1 1 0	7E
Z=0	JPNZ	0 1 1 1 1 1 0 0	7C
C=1	JPC	0 1 1 1 1 1 1 1	7F
C=0	JPNC	0 1 1 1 1 1 0 1	7D
unconditional	JP	0 1 1 1 1 0 0 1	79

Cycles: 6
Flags: None
Other: None

4. Other Instructions

4.1 PUSH Push the contents of the accumulator onto the stack.

R − 1 → R
A → (R)

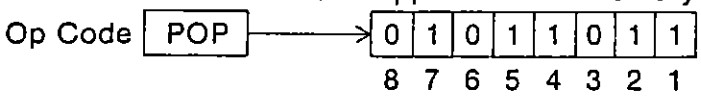


Hex. Op Code	Byte
34	1

Cycles: 3
Flags: None
Other: Register R is decremented

4.2 POP Pop the contents of the top of the stack into the accumulator.

(R) → A
R + 1 → R



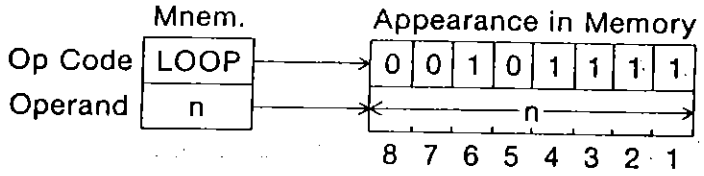
Hex. Op Code	Byte
5B	1

Cycles: 2
Flags: None
Other: Register R is incremented

4.3 LOOP

Decrement the top of the stack. If the carry flag equals 1, then execute the next instruction. If the carry equals 1, then make a relative jump from the address of the LOOP op-code to the address (PC+1−n). Note: One stack space is used for calculation.

```
(R)−1→(R)
IF C=0 THEN PC+1−n→PC
IF C=1 THEN PC+2 → PC
```



Hex. Op Code	Byte
2F	1
	2

Cycles: 10 if C=0
7 if C=1

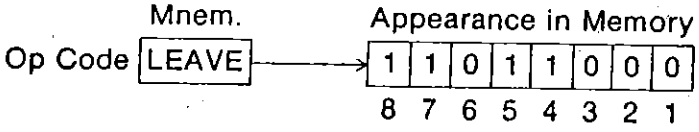
Flags: C, Z

Other: 1 stack space used for address calculation

4.4 LEAVE

Zero to top of stack - Store a zero in the top of the stack.

```
0→(R)
```



Hex. Op Code	Byte
D8	1

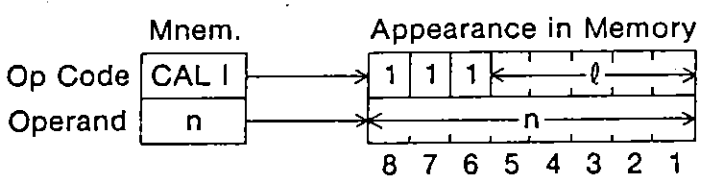
Cycles: 2

Flags: None

Other: None

4.5 CAL ln **Call Subroutine** - Store the address in the PC+2, the next command after the call, on the stack. Jump to the absolute address ln, an address in the first 8K bytes of memory (0000 1FFF).

PC+2→(R−1, R−2)
R−2→R
000ℓ→PCH
n→PCL

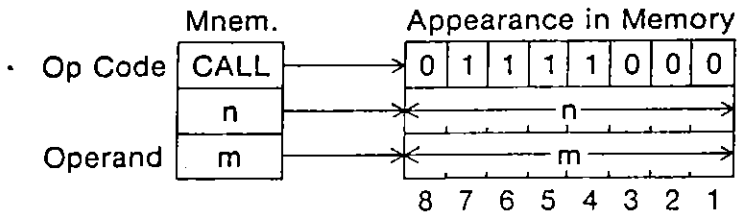


Hex. Op Code	Byte
E0+ℓ	1
n	2

Cycles: 7
Flags: None
Other: None

4.6 CALL nm **Call Subroutine** - Store the address in the PC+2, the next command after the call, on the stack. Jump to the absolute address nm, anywhere in the 64 bytes.

PC+3→(R−1, R−2)
R−2→R
n→PCH
m→PCL

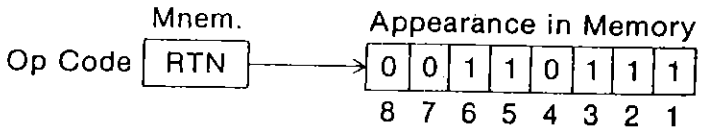


Hex. Op Code	Byte
78	1
	2
	3

Cycles: 8
Flags: None
Other: None

4.7 RTN **Return from subroutine -** Pop the address on the stack into the PC.

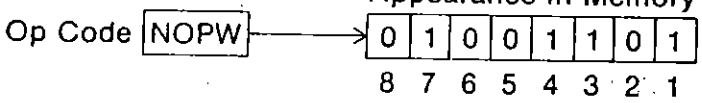
```
(R)→PCL
(R+1)→PCH
R+2→R
```



Hex. Op Code	Byte
37	1

Cycles: 4
Flags: None
Other: None

4.8 NOPW No operation — 2 cycles

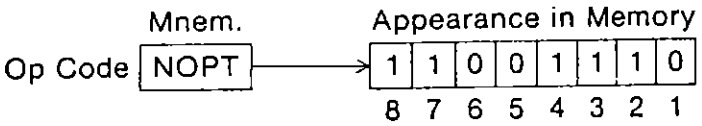


Hex. Op Code	Byte
4D	1

Cycles: 2
Flags: None
Other: None

4.9 NOPT

No operation — 3 cycles

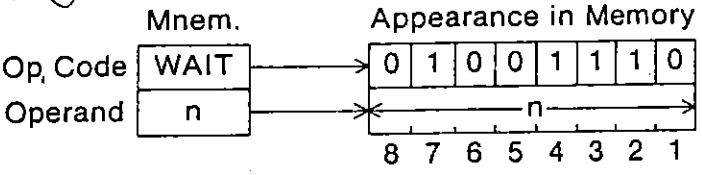


Op Code	Hex. Op Code	Byte
NOPT	CE	1

Cycles: 3
Flags: None
Other: None

4.10 WAIT

No operation — 6+n cycles



Op Code	Hex. Op Code	Byte
WAIT	4E	1
n		2

Cycles: 6+n
Flags: None
Other: None

4.11 OUTC

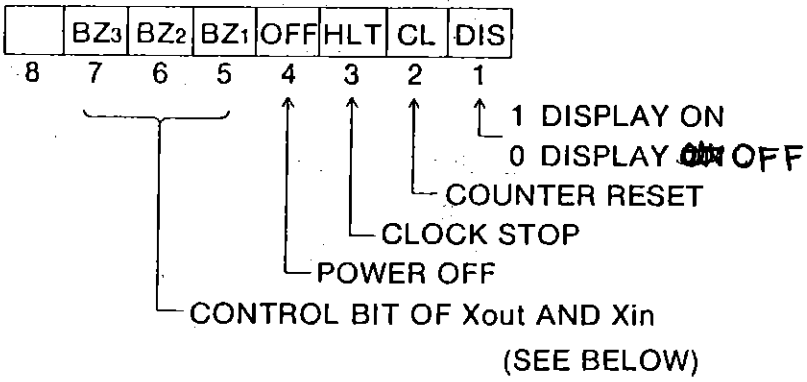
Write the contents of the internal RAM address 5F to the control port. Bit 1 controls the LCD display. (0=DISPLAY OFF, 1=DISPLAY ON). Note: it is important to preserve the rest of this byte when altering bit 1.

(5F)→control port

	Mnem.	Appearance in Memory								Hex. Op Code	Byte
Op Code	OUTC	→	1	1	0	1	1	1	1	1	DF	1
			8	7	6	5	4	3	2	1		

Cycles: 2
Flags: None
Other: None

(5F)



BZ_3	BZ_2	BZ_1	Xout	Xin
0	0	0	LOW	Not Active
0	0	1	HIGH	Not Active
0	1	0	2kHz	Not Active
0	1	1	4kHz	Not Active
1	0	0	LOW	Active
1	0	1	HIGH	Active
1	1	x	Xin→Xout	Active

4.12 OUTA

USATA PER LA TASTIERA

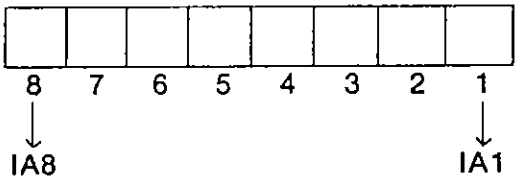
Each bit of (5C) appears to the IA port.

(5C)—IA port

Op Code	Mnem.	Appearance in Memory	Hex. Op Code	Byte
	OUTA →	0 1 0 1 1 1 0 1 (8 7 6 5 4 3 2 1)	5D	1

Cycles: 3
Flags: None
Other: None

(5C)



4.13 OUTB

Each bit of (5D) appears to the IB port.

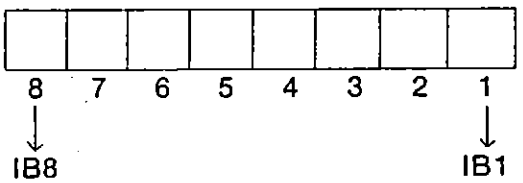
SONO DISPONIBILI LE LINEE DA IB5 A IB8 SUL CONNETTORE DI USCITA

(5D)—IB port

Op Code	Mnem.	Appearance in Memory	Hex. Op Code	Byte
	OUTB →	1 1 0 1 1 1 0 1 (8 7 6 5 4 3 2 1)	DD	1

Cycles: 2
Flags: None
Other: None

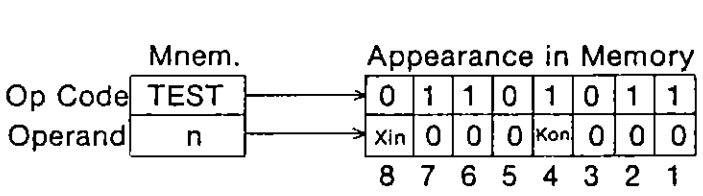
(5D)



4.17 TEST

n SET the bit of the operand which is required to test.

n−TEST



Cycles: 4
Flags: None
Other: None

EX)
```
TEST
80          Judge the input level of Xin
            IF  Xin=1       Z=0
                Xin=0       Z=1
TEST
08          Judge the input level of Kon
            IF  KON(OUT)=1  Z=0
                KON(OUT)=0  Z=1
TEST        2 msec. Counter test
02

TEST        Approx. 0.5 sec. Counter test
01
```

APPENDIX

OP CODES IN ALPHABETICAL ORDER

mnemonic	function	bytes	cycles	hex. op. code
ADB	[P+1, P]+[BA]→[P+1, P]	1	5	14
ADCM	(P)+A+C→(P)	1	3	C4
ADIA n	A+n→A	2	4	74
ADIM n	(P)+n→(P)	2	4	70
ADM	(P)+A→(P)	1	3	44
ADN	(P)+A→(P) (BCD)	1	7+3d	0C
ADW	(P)+(Q)→(P) (BCD)	1	7+3d	0E
ANIA	A∧n→A	2	4	64
ANID	(DP)∧n→(DP)	2	6	D4
ANIM n	(P)∧n→(P)	2	4	60
ANMA	(P)∧A→(P)	1	3	46
CAL In	PC+2→(R−1, R−2) R−2→R In→PC	2	7	E0+I
CALL nm	(PC+3)→(R−1, R−2) R−3→R nm→PC	3	8	78
CPIA n	A−n C,Z	2	4	67
CPIM n	(P)−n C,Z	2	4	63
CPMA	(P)−A C,Z	1	3	C7
DECA	A−1→A	1	4	43
DECB	B−1→B	1	4	C3
DECI	I−1→I	1	4	41
DECJ	J−1→J	1	4	C1
DECK	K−1→K	1	4	49
DECL	L−1→L	1	4	C9
DECP	P−1→P	1	2	51
DX	X−1→X, X→DP	1	6	05
DXL	X−1→X, X→DP, (DP)→A	1	7	25
DY	Y−1→Y, Y→DP	1	6	07
DYS	Y−1→Y, Y→DP, A→(DP)	1	6	27
EXAB	A↔B	1	3	DA
EXAM	A↔(P)	1	3	DB
EXB	(P)↔(Q)	1	6+3d	0B
EXBD	(P)↔(DP)	1	7+6d	1B
EXW	(P)↔(Q)	1	6+3d	09
EXWD	(P)↔(DP)	1	7+6d	19
FILD	A→(DP), DP+1→DP	1	4+3d	1F
FILM	A→(P), P+1→P	1	5+d	1E
INA	IA port→A	1	2	4C
INB	IB port→A	1	2	CC
INCA	A+1→A	1	4	42

mnemonic	function	bytes	cycles	hex. op. code	Dec
INCB	B+1→B	1	4	C2	194
INCI	I+1→I	1	4	40	64
INCJ	J+1→J	1	4	C0	192
INCK	K+1→K	1	4	48	72
INCL	L+1→L	1	4	C8	200
INCP	P+1→P	1	2	50	80
IY	Y+1→Y, Y→DP	1	6	06	6
IYS	Y+1→Y, Y→DP A→(DP)	1	6	26	38
IX	X+1→X X→DP	1	6	04	4
IXL	X+1→X, X→DP (DP)→A	1	7	24	36
JP nm	n→PCH m→PCL	3	6	79	121
JPC nm	IF C=1 THEN n→PCH m→PCL IF C=0 THEN PC+3→PC	3	6	7F	127
JPNC nm	IF C=0 THEN n→PCH m→PCL IF C=1 THEN PC+3→PC	3	6	7D	125
JPNZ nm	IF Z=0 THEN n→PCH m→PCL IF Z=1 THEN PC+3→PC	3	6	7C	124
JPZ nm	IF Z=1 THEN n→PCH m→PCL IF Z=0 THEN PC+3→PC	3	6	7E	126
JRCM n	IF C=1 THEN PC+1−n→PC IF C=0 THEN PC+2→PC	2	7/4	3B	59
JRCP n	IF C=1 THEN PC+1+n→PC IF C=0 THEN PC+2→PC	2	7/4	3A	58
JRM n	PC+1−n→PC	2	7	2D	45
JRNCM n	IF C=0 THEN PC+1−n→PC IF C=1 THEN PC+2→PC	2	7/4	2B	43
JRNCP n	IF C=0 THEN PC+1+n→PC IF C=1 THEN PC+2→PC	2	7/4	2A	42
JRNZM n	IF Z=0 THEN PC+1−n→PC IF Z=1 THEN PC+2→PC	2	7/4	29	41
JRNZP n	IF Z=0 THEN PC+1+n→PC IF Z=1 THEN PC+2→PC	2	7/4	28	40
JRP n	PC+1+n→PC	2	7	2C	44
JRZM n	IF Z=1 THEN PC+1−n→PC IF Z=0 THEN PC+2→PC	2	7/4	39	57
JRZP n	IF Z=1 THEN PC+1+n→PC IF Z=0 THEN PC+2→PC	2	7/4	38	56

mnemonic	function	bytes	cycles	hex. op. code
LEAVE	0→(R)	1	2	D8
LDD	(DP)→A	1	3	57
LDM	(P)→A	1	2	59
LDP	P→A	1	2	20
LDQ	Q→A	1	2	21
LDR	R→A	1	2	22
LIA n	n→A	2	4	02
LIB n	n→B	2	4	03
LIDL n	n→DPL	2	5	11
LIDP nm	n→DPH, m→DPL	3	8	10
LII n	n→I	2	4	00
LIJ n	n→J	2	4	01
LIP n	n→P	2	4	12
LIQ n	n→Q	2	4	13
LOOP	(R)−1→(R) IF C=0 THEN PC+1→n→PC IF C=1 THEN PC+2→PC	2	10/7	2F
LP ℓ	I→p	1	2	80+ℓ
MVB	(Q)→(P)	1	5+2d	0A
MVBD	(DP)→(P)	1	5+4d	1A
MVDM	(P)→(DP)	1	3	53
MVMD	(DP)→(P)	1	3	55
MVW	(Q)→(P)	1	5+2d	08
MVWD	(DP)→(P)	1	5+4d	18
NOPT	NOP	1	3	CE
NOPW	NOP	1	2	4D
ORIA n	A∨n→A	2	4	65
ORID n	(DP)∨n→(DP)	2	6	D5
ORIM n	(P)∨n→(P)	2	4	61
ORMA	(P)∨A→(P)	1	3	47
OUTA	(5C)→IA port	1	3	5D
OUTB	(5D)→IB port	1	2	DD
OUTF	(5E)→F0 port	1	3	5F
OUTC	(5F)→CONTROL	1	2	DF
POP	(R)→A R+1→R	1	2	5B
PUSH (DO)	R−1→R A→(R)	1	3	34
RC	0→C 1→Z	1	2	D1
RTN	(R)→PCL, (R+1)→PC R+2→R	1	4	37

mnemonic	function	bytes	cycles	hex. op. code	De
SBB	[P+1, P]−[BA]−[P+1, P]	1	5	15	21
SBCM	(P)−A−C−(P)	1	3	C5	19
SBIA n	A−n−A	2	4	75	11
SBIM n	(P)−n−(P)	2	4	71	11
SBM	(P)−A−(P)	1	3	45	69
SBN	(P)−A−(P) (BCD)	1	7+3d	0D	13
SBW	(P)−(Q)−(P) (BCD)	1	7+3d	0F	15
SC	1−C, 1−Z	1	2	D0	208
SL	1 bit shift left	1	2	5A	90
SLW	4 bit shift left	1	5+d	1D	29
SR	1 bit shift right	1	2	D2	210
SRW	4 bit shift right	1	5+d	1C	28
STD	A−(DP)	1	2	52	82
STP	A−P	1	2	30	48
STQ	A−Q	1	2	31	49
STR	A−R	1	2	32	50
SWP	A1−4−A5−8	1	2	58	88
TEST n	n−TEST	2	4	6B	107
TSIA n	A∧n Z	2	4	66	102
TSID n	(DP)∧n Z	2	6	D6	214
TSIM n	(P)∧n Z	2	4	62	98
WAIT n	NOP	2	6+n	4E	78

121

Machine Code to Op Code Conversion

First 4 bits

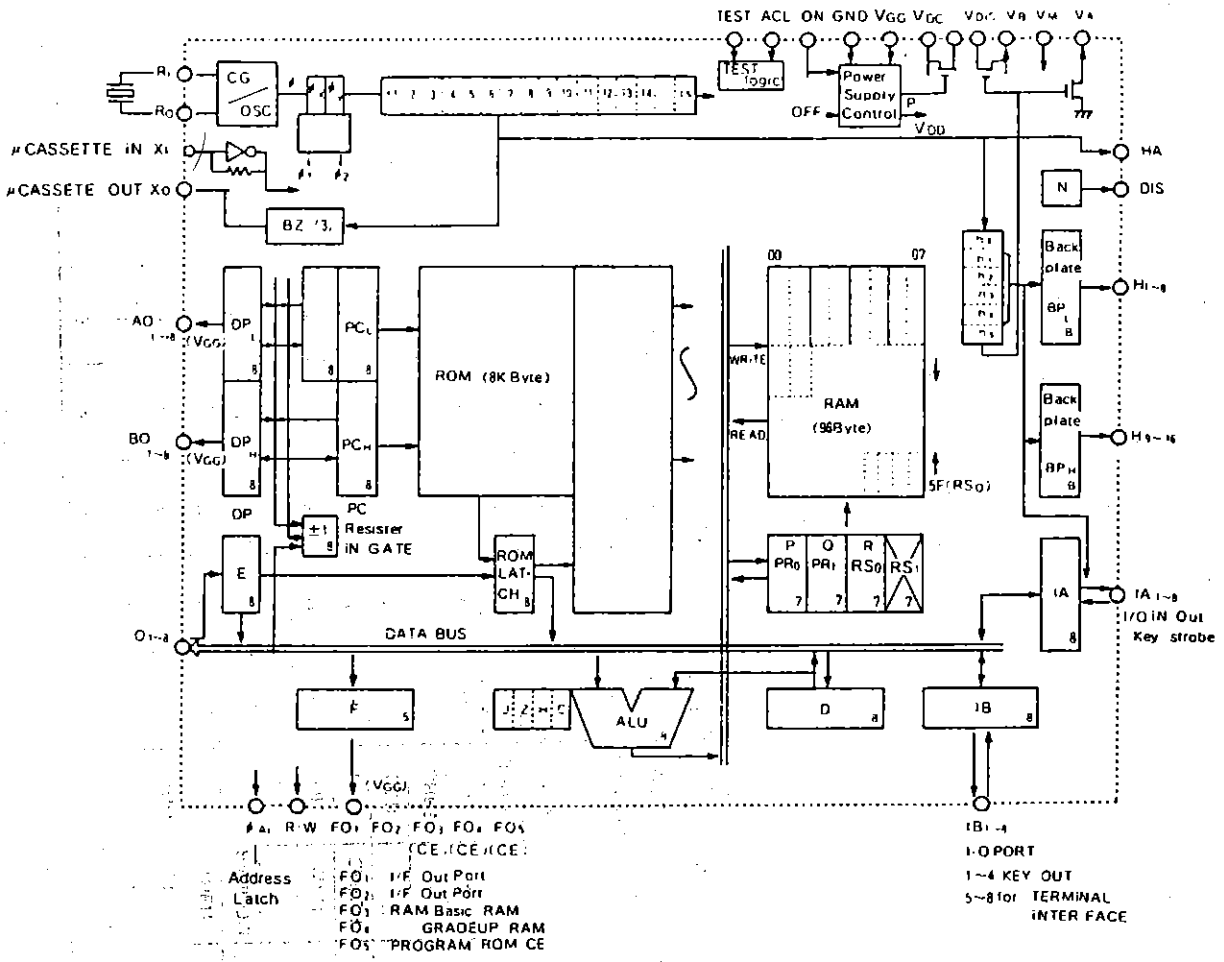
Second 4 bits I_8~I_5 / I_7~I_4	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	LII n	LIDP $\frac{n}{m}$	LDP	STP	INCI	INCP	ANIM n	ADIM n					INCJ	SC		
1	LIJ n	LIDL n	LDQ	STQ	DECI	DECP	ORIM n	SBIM n					DECJ	RC		
2	LIA n	LIP n	LDR	STR	INCA	STD	TSIM n						INCB	SR		
3	LIB n	LIQ n			DECA	MVDM	CPIM n						DECB	WRIT		
4	IX	ADB	IXL	DO PUSH	ADM	READM	ANIA n	ADIA n					ADCM	ANID n		
5	DX	SBB	DXL	DATA	SBM	MVMD	ORIA n	SBIA n		LP	I		SBCM	ORID n		
6	IY		IYS		ANMA	READ	TSIA n							TSID n	CAL	I n
7	DY		DYS	RTN	ORMA	LDD	CPIA n						CPMA			
8	MVW	MVWD	JRNZP n	JRZP n	INCK	SWP		CALL nm					INCL	LEAVE		
9	EXW	EXWD	JRNZM n	JRZM n	DECK	LDM		JP nm					DECL			
A	MVB	MVBD	JRNCP n	JRCP n		SL								EXAB		
B	EXB	EXBD	JRNCM n	JRCM n		POP	TEST n							EXAM		
C	ADN	SRW	JRP n		INA			JPNZ nm					INB			
D	SBN	SLW	JR n JRM n		NOPW	OUTA		JPNC nm						OUTB		
E	ADW	FILM			WAIT n			JPZ nm					NOPT			
F	SBW	FILD	LOOP n			OUTF		JPC nm						OUTC		

Internal Representation of BASIC

High nibble / Low nibble	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	End of line			(	0		P	~	~	TO	SIN	RUN	GRAD	IF		
1		(blank)		)	1	A	Q	~	AND	STEP	COS	NEW	PRINT	FOR		
2		"		>	2	B	R	~	>=	THEN	TAN	MEM	INPUT	LET		
3		?		<	3	C	S	~	<=	RANDOM	ASN	LIST	RADIAN	REM		
4		!		=	4	D	T	~	<>	~	ACS	CONT	DEGREE	END		
5		#		+	5	E	U	~	OR	WAIT	ATN	DEBUG	CLEAR	NEXT		
6		%		-	6	F	V	~	NOT	ERROR	EXP	CSAVE	~	STOP		
7				*	7	G	W	~	SQR	~	LN	CLOAD	~	GOTO		
8		$		/	8	H	X	~	CHR$	~	LOG	MERGE	~	GOSUB		
9		π		^	9	I	Y	~	~	~	INT	TRON	CALL	CHAIN		
A		√			•	J	Z	~	INKEY$	~	ABS	TROFF	DIM	PAUSE		
B					E	K		~	STR$	~	SGN	PASS	DATA	BEEP		
C					■	L		~	LEFT$	~	DEG	LLIST	ON	AREAD		
D						M		ASC	RIGHT$	~	DMS	PI	OFF	USING		
E					~	N		VAL	MID$	~	RND	~	POKE	RETURN		PROGRAM BEG/END
F		&				O		LEN		LPRINT	PEEK	~	READ	RESTORE		

C.P.U. INTERNAL BLOCK DIAGRAM & TERMINAL DESCRIPTIONS

● C.P.U. (SC61860A02) ····· 8 Bits C-MOS C.P.U.



● CPU pin No. and signal description

Pin No.	Signal name	IN/OUT	Description (Standby = power off)
1	AO1	OUT	Address bus, high during standby.
2	R/W	OUT	Write clock, normally high.
3	ØAL	OUT	Low order bits address latch, normally high. The clock used to latch low order 8 bits of 16-bit address signal on the data bus line, when a large capacity ROM is used.
4	TES	IN	Test pin, normally low.
5	ØI	IN	Oscillator input.
6	ØO	OUT	Oscillator output.
7	RES	IN	Reset input, active high. Normally, pulled down to low level.
8,	Xin	IN	Input (MT in) of microcassette signal from the CE-125 option.
9	ON	IN	ON (BREAK) key input, normally pulled down to low level.
10	Xout	OUT	Output (MT out 1) of microcassette signal to the CE-125 option and the buzzer.
11	Dis	OUT	LCD driver control signal.
12	HA	OUT	LCD driver clock, low during standby and in 2 kHz oscillation during display.

Pin No.	Signal name	IN/OUT	Description (Standby = power off)
13	iA8	IN/OUT	Key input/strobe signal, low during standby and key-in pulse is generated when low.
14	iA7	IN/OUT	Key input/strobe signal, low during standby and key-in pulse is generated when low.
15	iA6	IN/OUT	Key input/strobe signal, low during standby and key-in pulse is generated when low.
16	iA5	IN/OUT	Key input/strobe signal, low during standby and key-in pulse is generated when low.
17	iA4	IN/OUT	Key input/strobe signal, low during standby and key-in pulse is generated when low.
18	iA3	IN/OUT	Key input/strobe signal, low during standby and key-in pulse is generated when low.
19	iA2	IN/OUT	Key input/strobe signal, low during standby and key-in pulse is generated when low.
20	iA1	IN/OUT	Key input/strobe signal, low during standby and key-in pulse is generated when low.
21	iB8	IN	ACK signal that enables the CPU to read data through I/O port (PCU).
22	iB7	IN	Data in (Din). Serial data input from PCU (bit by bit serial handshake).
23	iB6	OUT	SEL2 output, P-type open drain.
24	iB5	OUT	SEL1 output, P-type open drain.
25	iB4	IN	Slide switch input.
26	iB3	OUT	Key strobe output, low during standby and key-in pulse is generated when low.
27	iB2	OUT	Key strobe output, low during standby and key-in pulse is generated when low.
28	iB1	OUT	Key strobe output, low during standby and key-in pulse is generated when low.
29	VM	IN	LCD power supply.
30	VA	IN	LCD power supply.
31	GND	IN	Power supply.
32	H1	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
33	H2	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
34	H3	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
35	H4	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
36	H5	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
37	H6	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
38	H7	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
39	H8	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
40	H9	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
41	H10	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
42	H11	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
43	H12	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
44	H13	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
45	H14	OUT	LCD backplate signal, high impedance during standby and 4-level pulse during display.
46	H15	—	Not used. (Display of this unit operates in 1/14 duty.)
47	H16	—	Not used. (Display of this unit operates in 1/14 duty.)
48	V_B	IN	LCD power supply, high during standby and low when the high level clock stops.
49	V_{DIS}	IN	LCD power supply, high during standby and low when the high level clock stops.
50	V_{CC}	IN	LCD power supply, normally low.

Pin No.	Signal name	IN/OUT	Description (Standby = power off)
51	V_{DC}	OUT	LCD power supply, high during standby and low when the high level clock stops.
52	V_{GG}	IN	Power supply normally low.
53	O8	IN/OUT	Data bus, normally high impedance.
54	O7	IN/OUT	Data bus, normally high impedance.
55	O6	IN/OUT	Data bus, normally high impedance.
56	O5	IN/OUT	Data bus, normally high impedance.
57	O4	IN/OUT	Data bus, normally high impedance.
58	O3	IN/OUT	Data bus, normally high impedance.
59	O2	IN/OUT	Data bus, normally high impedance.
60	O1	IN/OUT	Data bus, normally high impedance.
61	Fo5	OUT	External system ROM chip enable (CE1).
62	Fo4	OUT	Expansion RAM (PC-1251 RAM) chip enable (CE2).
63	Fo3	OUT	Basic RAM (PC-1250, PC-1251) chip enable (CE3).
64	Fo2	OUT	Data out (Dout), data output port to peripheral.
65	Fo1	OUT	BUSH I/F output port output.
66	Bo8	OUT	LCD driver LSI chip enable (CE4).
67	Bo7	OUT	A14. Address bus, high during standby.
68	Bo6	OUT	A13. Address bus, high during standby.
69	Bo5	OUT	A12. Address bus, high during standby.
70	Bo4	OUT	A11. Address bus, high during standby.
71	Bo3	OUT	A10. Address bus, high during standby.
72	Bo2	OUT	A9. Address bus, high during standby.
73	Bo1	OUT	A8. Address bus, high during standby.
74	Ao8	OUT	A7. Address bus, high during standby.
75	Ao7	OUT	A6. Address bus, high during standby.
76	Ao6	OUT	A5. Address bus, high during standby.
77	Ao5	OUT	A4. Address bus, high during standby.
78	Ao4	OUT	A3. Address bus, high during standby.
79	Ao3	OUT	A2. Address bus, high during standby.
80	Ao2	OUT	A1. Address bus, high during standby.