Bitchin100 Magazine

The Magazine Devoted to 8-bit Retro Laptop Users

In This Issue

July 2005: Issue 1, Number 1

8085 Assembly Language Programming
Roger Merchberger

Part One of Two

In this series, Roger will introduce us to assembly language programming on the CPU used in the Model 100 and NEC 8201A

► A little personal history: I purchased my first computer in 1984, a Tandy Color Computer 2 with 16K RAM and extended BASIC. I taught myself RSBasic, RS-DOS, OS-9, Basic09 and 6809 Assembly Language before I upgraded to a Tandy Color Computer 3 in 1986. These skills came in handy in college as I was expected to learn 6800 Assembly in a few of my classes. I'd realized Tandy computers to be very powerful, even if they weren't the most popular; so in 1989 when I required a more portable solution, I purchased a Tandy 200.

I'd found that the machine was the most complete and user-friendly machine available at the time. The evidence speaks for itself – it's the only computer I never voided the warranty on! As a matter of fact, I never opened the computer's case for at least a *decade* after I purchased it, and that was not to repair it, but using it as a test-bed to learn about repairing other broken Tandy 200's. 16 years later, and I've finally had to open the case to replace the on-board NiCd battery, and whilst I was in there, I upgraded the memory to 72K RAM. I'm typing this "beginner's foray" into 8085 Assembly Language on this very same machine.

With it's built-in spreadsheet program, larger screen and full keyboard, I found it a most complete solution to portable computing. Amazingly, it was so complete, that it's also the only machine that I've ever owned that I considered more an appliance or tool than a toy. As such, I'd never learned the nuances of the hardware itself, including the processor, as it seems I never really *needed* to. Well, that's about to change.

Although I'm an experienced [albeit rusty] 6809 Assembly Language programmer, I've never delved into the Intel world at all until now. I'm going to take you with me through my first baby steps of programming the 8085 processor and hopefully it will be a pleasant learning experience for us all.

As a preamble to learning any assembly language is knowing how to deal with binary and hexadecimal (or for brevity, hex) numbers. On earlier processors like the DEC PDP/11, octal was also very important, but it's less so on 8-bit microprocessors like the 8085. Binary is base 2 - off or on, 0 or 1, and this is the computer's true numerical encoding. Unfortunately, it's fairly difficult for humans to deal with binary directly, so that's why we use octal (base 8, or 3 binary digits or bits) or hex (base 16, or 4 bits - otherwise called a "nybble"). Decimal is not nearly as useful to the computer itself as it is to humans, and so I encourage you to practice getting used to not only how hex works, but thinking in it directly. If I told you to start your program at memory location 62600, to the computer it's represented as 1111010010001000. You can see that this is not exactly 'human-readable' and converting base 2 to base 10 and vice-versa is difficult, at least in your head. However, if you break down the 16 bit address above into 4-bit chunks, or "nybbles", it's much easier to deal with the location as represented in hex, which is F844. Any CPU with a 16-bit address bus has access to memory locations from 0 to FFFF hex, or in decimal, 0 to 65535.

If you would like a quick example as to why it's easier to deal with hex directly, look at the ASCII codes that your computer uses for character display. It doesn't seem very intuitive that upper case characters start at 65 decimal and lower case characters start at 97 decimal, until you see that those convert to 41 and 61 hex. To change uppercase to lowercase, just add 20 hex to the character, subtract that value to change lowercase to uppercase.

Once you get the basics of computer numbering systems down, the next thing we will need to do is learn the hardware architecture itself. Knowing what registers are available and their purpose is paramount to Assembly language programming, just like knowing the difference between string, integer or floating point variables in Basic.

In the 8085, all registers either store 8 bits or 16 bits of information. 8 bits of information gives a range of values between 0 and 255 decimal or 0 to FF hex. The 16-bit registers can store 0 to 65535 decimal, or 0 to FFFF hex.

There are several registers in the 8085 that we can use - some of which have very specialized uses. I will go over these registers and include a brief synopsis as to their function.

PC - Program Counter: This is a 16-bit register that keeps the CPU from 'getting lost.' It keeps track of the memory location of the next instruction. If it doesn't keep track of where it is in the program, it could never get anything done in an orderly manner.

A - Accumulator: This is an 8-bit register that is used mainly for mathmatical operations, like adding and subtracting 2 operands.

B and C, D and E: Storage registers that can be used as either 4 8-bit registers, or 2 16-bit registers. These can be used to hold temporary values without saving them to RAM, as reading and writing RAM requires many more CPU cycles than working with internal registers.

H and L: These can be used as 2 8-bit or 1 16-bit storage register(s) if you choose, but together these two registers have much more powerful abilities, as this pair of registers can be used as a pointer anywhere in memory, like a stack pointer or index pointer.

PSW, or Program Status Word: An 8-bit register, of which 5 bits are used. These show the 'status' of a mathmatical operation. The five bits are broken down thusly:

Z (Zero) flag - this flag is set if a mathematical operation results in a value equal to Zero.

C (Carry) flag - If an add operation resulted in any carry - as in the result would have been bigger than 255 decimal (FF hex), this bit will be set.

P (Parity) flag - this bit will be set if there are an even number of 1 bits inthe result of the operation.

S (Sign) flag - During a mathmatical operation, if the 7th bit (also called the Most Significant Bit) of the Accumulator is a 1, this bit is set; this is significant when using signed numbers - if the MSB is 0, then the number is positive; if the MSB is 1, then the number is negative. This gives a range of signed numbers (in 8 bits) of -128 to +127; 16-bit numbers from -32768 to +32767.

AC (Auxiliary Carry) If a "half-carry" was performed (if bit 4 is set after an addition) this bit will be set. Chances are, of all the bits in the Program Status Word, you'll use this one the least.

To be honest, I've gotten a bit wordy thus far, and yet, I've barely scratched the surface of this topic - I really do recommend getting a few good books about 8085 programming. If you can't find those, books on 8080A programming are a good bet too, as these processors are very similar, and the 8085 is "backwards compatible" with the 8080. This means that the 8085 can execute all of the instructions that the 8080 can, but there's a few new tricks it can do that were not designed into the 8080.

For learning the processor itself, I recommend "8080A-8085 Assembly Language Programming" by Lance A. Leventhal - it's very in-depth with respect to the instruction set, Program Status Word bits, and the differences between the 8080A and the 8085 processors. Honestly though, it's assembly language programming examples are definitely not "light reading." For better reading about putting the instruction set into good use, I would recommend " insert the name of the book here " I don't currently own this book (yet) but I have it on good authority that it will put everything you learned in the Leventhal book to good practice.

You're going to need to know the actual instruction set for the 8085, and I've listed a couple of good books above that you should consult for deeper understanding of it, but I would be remiss if I didn't at least outline some of the more common instructions, especially those that are used in the assembly language examples that are to follow. (Yes, I *will* get to the good stuff eventually! ;-))

Here are some of the more frequently used assembly language instructions:

ADC, ACI Add A accumulator with carry (immediate)

ADD, ADI Add A accumulator (immediate)

ANA, ANI Logical And (immediate)

CALL Call a subroutine

CMP, CPI Compare register with a value (immediate)

IN, OUT Input value from a Port / Output value to Port

INR, DCR Increment (decrement) a register or memory

INX, DCX Increment (decrement) a 16-bit register pair

JC, JNC Jump on carry (not carry)

JZ, JNZ Jump on zero (not zero)

JMP Jump unconditionally

LDA Load the A accumulator with a value

LXI Load a 16-bit value into a

register pair

MOV, MVI Move data between registers or memory (immediate)

RAL, RAR Rotate with carry Left (Right)

RET Return from a subroutine

STA Store A Accumulator to memory

SUB, SUI Subtract A Accumulator (immediate)

There are also certain "commands" that the assembler uses to perform certain functions - these statements don't actually represent CPU instructions, but set up the environment for your program.

For example, how do you tell the assembler where in memory you want to put your program? If you don't tell it where, the assembler will assume address 0000 hex - but that's ROM space, so the assembler will generate an error. To tell the assembler where to start in memory, use the ORG (origin) directive. Also, you'll want to use labels when naming loops and data spaces so the assembler can keep track of the addresses; it'll save you a lot of trouble than doing it manually!

Here's an assembly language that doesn't actually *do* anything, but shows how to use several of the assembler's directives:

```
        ORG     62600           ; start the first program address at location 62600

DISPLY: EQU     5A58H           ; location of the ROM routine to print a null-terminated string.

CHRGET: EQU     12CBH           ; location of the ROM routine that reads the keyboard and returns the ASCII equivalent.

BEGIN:  JMP     START           ; jump over the data to follow, more on this later.

; Anything starting with a semicolon is a comment and is ignored by the assembler.

; Next, we're going to define a string, followed a Carriage Return and Line Feed, and ended by a NULL value to denote the end of the line.

STR1:   DB      "Here's a string!",10,13,00

; Next, we're going to just denote a CRLF string, this can be handy for string output.

CRLF:   DB      10,13,00

; If we needed to allocate (for example) 20 bytes of RAM for a buffer or stack, we'd use the DS (Define Storage) directive.

STK1:   DS      20      ; 20 bytes reserved for our nefarious purposes!

; Finally, we're going to get to the real program...

START:  RET             ; We're just going to return back to BASIC or whatever called us...
```

OK, for a program that really does nothing but take up space, there's still a little explaining to do. Firstly, the JMP instruction at the beginning of the program is actually superfluous, and technically increases the size of the program by 3 bytes, so why did I do it? Simple - for an old fart like me, it's called "one less thing to remember." When you go to BASIC and you load a machine language program, it will give you 3 addresses, like this:

Top: 62600   [[Remember this from the ORG statement?]]

End: 62644

Exe: 62600

If we had not put that JMP instruction at the beginning of the program to jump over all the data, we would have had to remember where the START: address was in memory, and set *that* as the Exe: address. By using the JMP, you know that the Top: address of the program and the Exe: address of the program are the same, making it a bit simpler for those of us new to assembly language programming. In my opinion, definitely worth 3 bytes. ;-)

The EQU actually takes up no memory in the program - it's just there to set a human-readable label to a value. It's a lot easier to remember that CHRGET is the location of the routine to read a key instead of 12CBH!

The other directives that I used in the program are pretty well commented in the program, so we're going to move on.

Why reinvent the wheel?

There are a lot of tricks that a fledgling assembly language programmer can use, and probably the biggest timesaver of all would be re-using the machine code that already exists in your computer. Microsoft had to write a lot of routines to get your Model T to do what it can do, including reading the keyboard, printing characters and lines to the screen and printer, and even serial port I/O. Why should you rewrite everything from scratch when Microsoft did most of it for you? You don't! However, you do need to learn how to use the routines that they provided, usually by setting specific values in certain registers before you call the subroutine.

For our next example, which is about the 3rd version of the very first assembly language program I wrote for the Model 10x, we'll be using these routines:

LCDOUT - 4B44H ; Output a single character in A to the display.

CHRGET - 12CBH ; Wait for a keypress from the keyboard, and

; store that value in A

DISPLAY - 5A58H ; Output a Null-terminated string in memory (pointed

; to by HL) to the display.

For the Model 200 you'd need to find the correct entry point addresses.

I've commented each line and the routines pretty well in the program itself, so it should be mostly self-explanetory as to what the program's doing and why. However, if you have questions that I (obviously) cannot forsee here in this document, please email me at z@30below.com and please put '8085' somewhere in the subject. That will make it much easier for me to respond to your email.

A note here about comments: There are good comments and there are bad comments. Most of the time, bad comments are actually worse than no comments at all! For example, if you have this:

LABEL: ADI 12 ; Add 12H to the accumulator

First off, you can tell from the instruction (ADI - Add Immediate to Accumulator) that it's going to add 12 to the A register. The comment really isn't saying anything you don't already know, and thanks to the typo in the comment, at first glance one may not notice that the program actually adds 0BH to the A register. This is a *bad* comment.

OK, so a lot of you are saying "Who'd be stupid enough to do that?" Quite often, we comment the program at the beginning of the code writing phase, so we can keep track of what we were trying to accomplish with the program. However, if the program didn't work as expected, we may make changes to the code and forget to modify the associated comments. The original line could have read:

LABEL: ADI 18 ; Add 12H to the accumulator

at which point the commect would be correct (yet still pointless) but during the debugging phase you found that 18 decimal didn't do what was expected in the program, so it was changed to 12, but the comment was not modified. This error happens quite often (I've done it on many occasions myself) and doesn't make one a bad person per se, but it would be much better to not get into that habit in the first place.

What is much better is to try to state what you *want* to happen in each line, like this:

LABEL: LDI 12 ; Store an ASCII space in the A reg.

Now, altho the code itself is obviously incorrect, the comment itself is right. If we now see that we're trying to work with unprintable characters, we can modify the command thusly:

LABEL: LDI 32 ; Store an ASCII space in the A reg.

It made the program do what we want, and the comment helps inform others (or the author in 6 months) as to *why* that instruction is there.

Also, you should put a small block of comments before each subroutine you write to give an overview of that routine, and also warn anyone looking at the code if it clobbers any registers or memory locations. You will also want to mention any entry and/or exit parameters. Some of the hardest bugs to track down are when registers or memory is modified and those changes are not taken into account. Let's say you have a particularly useful piece of code in a subroutine that you're proud of and you comment it thusly:

; My new subroutine!

; Holy Kukamunga, the darned thing works! And it works *Schweet!*

; d00d, I should get the Nobel Piece-o-code prize for this sucker!!!!!

OK, Now just what good did that do? Let's try it again like this:

; Subroutine: BRKNYB (Break a byte into 2 nybbles)

; Requires register A to be set to the value to be seperated.

; Returns: Location F850H as the Most Significant Nybble

; Location F851H as the Least Significant Nybble

; Clobbers: HL, all other registers preserved.

With the comments above, you have a fair idea as to what the routine does, and you also know that if you have a value you need to keep in register pair HL, you'll need to save it somewhere before calling this routine; but register pairs BC or DE are safe.

In these days of 200 Gig hard drives a lot of people say "Storage is cheap." With 3 GHz processors, they say "Cycles are cheap." In this case: "Comments are cheap." Comments are cheap insurance to jog your memory (or someone else's) about what a line of code or a routine does - but if used incorrectly, they can do more harm than good.

Stay tuned for Part 2!

As Seen On ebay

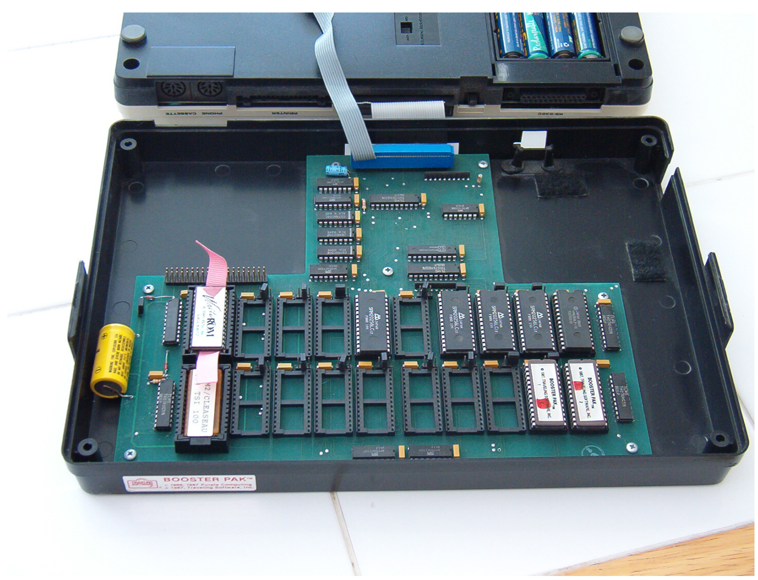
The purpose of the "As Seen on Ebay" column is to reintroduce old technology that (retro geek though you may be) you just haven't heard about yet. Or maybe you did but didn't investigate it.

This time we'll take a look at the Booster Pak, by Traveling Software.

Basically the Booster Pak adds about an inch of thickness to the bottom of the Model 100 or T102. That's a a big burrito, my friend. But what do you get?

- 96K standard RAMDisk file storage (self-powered)
- 2 standard Molex expansion ROMs carriers
- 11 standard DIP sockets for 32K RAM chips or EPROMS
- Built-in TS-DOS to access RAMDisk or external storage
- Xmodem file transfer directly to/from RAMDisk
- Asteroids!
- ROM-based software smart enough to coordinate it all.

More information on the Booster Pak can be found at http://www.geocities.com/m100er/

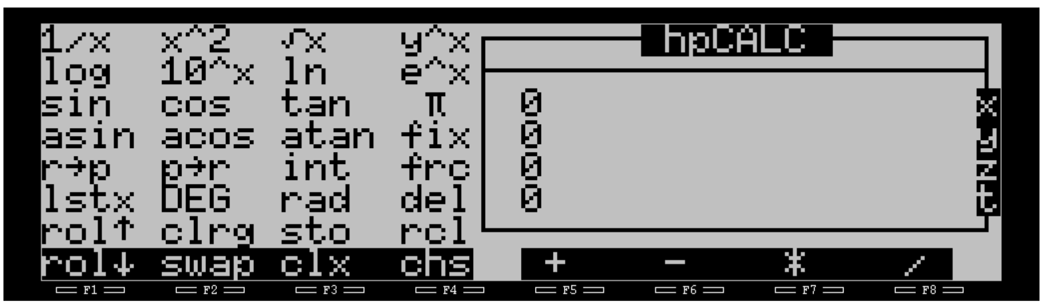


The Vault

HPCalc, HP Calculator Simulator
Scott T. Schad

You may remember this hit program from the Compuserve Model 100 SIG
Scott has permitted us to bring this gem back from The Vault.

hpCALC is an easy-to-use RPN (reverse-polish-notation) emulator for Tandy model 100/102 laptops. It operates identically with Hewlett-Packard calculators, but is not programmable. This documentation assumes you are sufficiently familiar with HP calculators to be interested in this program, so it concentrates on the unique aspects of hpCALC instead of teaching you RPN.



hpCALC's best feature is its simple user interface. All options appear on the screen, where they are easily accessed by a sliding-bar menu. You don't need to memorize cryptic keyboard combinations, although several keyboard shortcuts are provided.

Numeric entry can be from the embedded keypad or top row of keys. Use an "E" or "e" to enter powers of ten. Any calculation option shown to the left of the stack window can be selected by using the up and down cursor keys to slide a highlighted bar to the desired row. Hit the F1-F4 function keys to execute the calculation option when its row is selected. The four math function keys (+-*/) below the stack window keep their functions regardless of the menu bar position.

Keyboard shortcuts; "r"=roll down stack; "R"=roll up stack; "c"=clear x; "C"=clear registers; "D" or "d"=delete x; "s" or "S"=swap x&y; ESC=exit program. You can execute all of these commands (except ESC) by sliding the menu bar and pressing the indicated function key.

Commands: The "DEG" and "rad" function keys toggle capitalization on and off to indicate the current trig mode. The "Pi" key returns that number up to 14 digits. The "fix" key takes the integer value of the current number in the x- register and trims decimals to that number of displayed digits. You can set from 0-14 digits, with a default of 5. Attempting to use a number out of this range will reset the digits back to 5. The "p-r" and "r-p" are polar/rectangular conversions: put x in the x register and y in the y register, hit "r-p", and the radius will be left in x with the angle (in deg or rad) left in y. "p-r" reverses the calculation.

Storage: the "lstx" key will bring up the last x-register value which was entered or used in a calcualtion. "sto" and "rcl" provide access to a single data storage register.

Calculation errors: most errors are self-recovering. If you try to divide by zero for example, an "ERROR" message is briefly displayed in the x register, then x is redisplayed.

Available in electronic form on the web at http://bitchin100.com/hpcalc

Model 100/102/200 Program Listing:

```
0 'hpCALC (c) 1987 Scott Schad--REGISTER FOR $10: 3943 S. Delaware Pl., Tulsa, OK 74105
1 CLEAR1000:SCREEN0,0:CLS:KEYON:ONERRORGOTO83:LX$="0":X$(1)="0":X$(2)="0":X$(3)="0":X$(4)="0":K6=57.29577951308232:ONKEYGOSUB31,41,51,61,71,72,73,74:K$(1)="1/x x^2  "+CHR$(137)+"x   y^x":E$=CHR$(27)+"p":F$=CHR$(27)+"q
2 R$(1)=CHR$(240)+STRING$(6,CHR$(241))+E$+" hpCALC "+F$+STRING$(6,CHR$(241))+CHR$(242):K$(2)="log  10^x ln   e^x":R$(3)=CHR$(245)+STRING$(20," ")+E$+"x"+F$:K$(3)="sin  cos  tan   "+CHR$(136)+" ":R$(2)=CHR$(244)+STRING$(20,CHR$(241))+CHR$(249)
3 K$(4)="asin acos atan fix":R$(4)=CHR$(245)+STRING$(20," ")+E$+"y"+F$:K$(5)="r"+CHR$(154)+"p  p"+CHR$(154)+"r  int  frc ":JF=1:FX=5:R$(5)=CHR$(245)+STRING$(20," ")+E$+"z"+F$:K$(6)="lstx DEG  rad  del":R$(6)=CHR$(245)+STRING$(20," ")+E$+"t"+F$
4 K$(7)="rol"+CHR$(152)+" clrg sto  rcl":R$(7)=CHR$(246)+STRING$(20,CHR$(241))+CHR$(247):K$(8)="rol"+CHR$(153)+" swap clx  chs":R$(8)=" +    -    *    / ":FORI=0TO7:IFI<7THENPRINT@I*40,K$(I+1);:GOTO6
5 PRINT@I*40,E$+K$(I+1)+F$;
6 IFI<7THENPRINT@I*40+18,R$(I+1);:GOTO8
7 PRINT@I*40+20,E$+R$(I+1)+F$;
8 PRINT@I*40+18,H$(I+1);:NEXTI:P=7:NP=7:GOSUB75
9 ER=0:PRINT@99,X$(1):I$=INKEY$:IFI$=""THEN9
10 IFI$="c"THENP1=P:P=7:GOSUB51:P=P1:GOTO9
11 IFI$="C"THENP1=P:P=6:GOSUB42:P=P1:GOTO9
12 IFI$="D"ORI$="d"THENP1=P:P=5:GOSUB63:P=P1:GOTO9
13 IFI$="S"ORI$="s"THENP1=P:P=7:GOSUB41:P=P1:GOTO9
14 IFI$="r"THENP1=P:P=7:GOSUB32:P=P1:GOTO9
15 IFI$="R"THENP1=P:P=6:GOSUB33:P=P1:GOTO9
16 J=ASC(I$):IFJ<32THEN23
17 K=INSTR(1,"-0123456789.Ee",I$):IFK>0ANDJF=1THENGOSUB79:X$(1)="
18 IFK=0THEN22
19 IFK>0ANDX$(1)="0"ORKANDX$(1)=""ORK>0ANDCR=1THENX$(1)=I$:CR=0:GOSUB75:GOTO22
20 IFI$="-"ANDVAL(X$(1))=0THENCR=1:GOTO22
21 IFK>0THENX$(1)=X$(1)+I$:PRINT@99,X$(1)
22 JF=0:GOTO9
23 IFJ=27THENMENU
24 L=LEN(X$(1)):IFL>0ANDJ=8THENX$(1)=LEFT$(X$(1),L-1):GOSUB75:GOTO9
25 IFLEN(X$(1))=0THENX$(1)="0":GOSUB75:GOTO9
26 IFJ=13THENGOSUB79:X$(1)=X$(2):GOSUB75:CR=1:GOTO9
27 IFJ=30THENNP=P-1:IFNP<0THENNP=7
28 IFJ=31THENNP=P+1:IFNP>7THENNP=0
29 IFNP<>PTHENPRINT@P*40,K$(P+1);
30 PRINT@NP*40,E$+K$(NP+1)+F$;:P=NP:GOTO9
31 IFP<>5THENLX$=X$(1)
32 IFP=7THENLX$=X$(1):X$(1)=X$(2):X$(2)=X$(3):X$(3)=X$(4):X$(4)=LX$:JF=1:CR=1:GOSUB75:RETURN
33 IFP=6THENLX$=X$(1):X$(1)=X$(4):X$(4)=X$(3):X$(3)=X$(2):X$(2)=LX$:JF=1:CR=1:GOSUB75:RETURN
34 IFP=5THENGOSUB79:X$(1)=LX$:JF=1:GOSUB75:RETURN
35 IFP<>4THEN37
36 R=SQR(VAL(X$(1))^2+VAL(X$(2))^2):RA=(ATN(VAL(X$(2))/VAL(X$(1)))*K6):X$(1)=STR$(R):X$(2)=STR$(RA):JF=1:GOSUB75:RETURN
37 IFP=3THENGOSUB82:JF=1:GOSUB75:RETURN
38 IFP=2THENX$(1)=STR$(SIN(VAL(X$(1))/K6)):JF=1:GOSUB75:RETURN
39 IFP=1THENX$(1)=STR$(LOG(VAL(X$(1)))*.4342945):JF=1:GOSUB75:RETURN
40 IFP=0THENJF=1:X$(1)=STR$(1/VAL(X$(1))):GOSUB75:RETURN
41 LX$=X$(1):IFP=7THENX$(1)=X$(2):X$(2)=LX$:JF=1:CR=1:GOSUB75:RETURN
42 IFP=6THENX$(1)="0":X$(2)="0":X$(3)="0":X$(4)="0":ST$="0":JF=1:CR=1:GOSUB75:RETURN
43 IFP=5THENK6=57.29577951308232:GOSUB80:RETURN
44 IFP<>4THEN46
45 Y=VAL(X$(1))*(SIN(VAL(X$(2))/K6)):X=VAL(X$(1))*(COS(VAL(X$(2))/K6)):X$(1)=STR
```

```
$(X):X$(2)=STR$(Y):JF=1:GOSUB75:RETURN

46 IFP<>3THEN48
47 GOSUB82:JF=1:X$(1)=STR$(K6*(3.1415926
535898/2)-(VAL(X$(1)))):GOSUB75:RETURN

48 IFP=2THENX$(1)=STR$(COS(VAL(X$(1))/K6
)):JF=1:GOSUB75:RETURN
49 IFP=1THENX$(1)=STR$(10^VAL(X$(1))):JF
=1:GOSUB75:RETURN
50 IFP=0THENX$(1)=STR$(VAL(X$(1))^2):JF=
1:GOSUB75:RETURN
51 LX$=X$(1):IFP=7THENX$(1)="0":GOSUB75:
X$(1)="":RETURN
52 IFP=6THENJF=1:ST$=X$(1):GOSUB75:RETUR
N
53 IFP=5THENK6=1:GOSUB80:RETURN
54 IFP<>4THEN57
55 IFINSTR(1,X$(1),".")=0THENRETURN
56 DP=INSTR(1,X$(1),"."):X$(1)=LEFT$(X$(
1),DP):JF=1:GOSUB75:RETURN
57 IFP=3THENX$(1)=STR$(ATN(VAL(X$(1)))*K
6):JF=1:GOSUB75:RETURN
58 IFP=2THENX$(1)=STR$(TAN(VAL(X$(1))/K6
)):JF=1:GOSUB75:RETURN
59 IFP=1THENX$(1)=STR$(LOG(VAL(X$(1)))):
JF=1:GOSUB75:RETURN
60 IFP=0THENX$(1)=STR$(SQR(VAL(X$(1)))):
JF=1:GOSUB75:RETURN
61 LX$=X$(1):IFP=7THENX$(1)=STR$(-VAL(X$
(1))):GOSUB75:RETURN
62 IFP=6THENGOSUB79:JF=1:X$(1)=ST$:GOSUB
75:RETURN
63 IFP=5THENX$(1)=X$(2):GOSUB78:GOSUB75:
JF=1:RETURN
64 IFP<>4THEN67
65 IFINSTR(1,X$(1),".")=0THENRETURN
66 DP=LEN(X$(1))-INSTR(1,X$(1),"."):X$(1
)=RIGHT$(X$(1),DP+1):JF=1:GOSUB75:RETURN

67 IFP=3THENFX=INT(VAL(X$(1))):IFFX<0ORF
X>14THENFX=5:RETURNELSEX$(1)=X$(2):GOSUB
78:GOSUB75:JF=1:RETURN
68 IFP=2THENJF=1:GOSUB79:X$(1)="3.141592
6535898":GOSUB75:RETURN
69 IFP=1THENX$(1)=STR$(2.7182818284590^V
AL(X$(1))):JF=1:GOSUB75:RETURN
70 IFP=0THENJF=1:X$(1)=STR$(VAL(X$(2))^V
AL(X$(1))):GOSUB78:GOSUB75:RETURN
71 LX$=X$(1):JF=1:X$(1)=STR$(VAL(X$(2))+
VAL(X$(1))):GOSUB78:GOTO75
72 LX$=X$(1):JF=1:X$(1)=STR$(VAL(X$(2))-
VAL(X$(1))):GOSUB78:GOTO75
73 LX$=X$(1):JF=1:X$(1)=STR$(VAL(X$(2))*
VAL(X$(1))):GOSUB78:GOTO75
74 LX$=X$(1):JF=1:X$(1)=STR$(VAL(X$(2))/
VAL(X$(1))):GOSUB78:GOTO75
75 PV=VAL("0."+STRING$(FX,"0")+"5"):BA=V
AL("1"+STRING$(FX,"0")):FORI=1TO4:PRINT@
(I+1)*40+19,STRING$(20," "):IFX$(I)="-"O
RX$(I)="."THEN77
76 XX=VAL(X$(I)):XX=FIX((XX+PV*SGN(XX))*
BA)/BA:X$(I)=STR$(XX)
77 PRINT@(I+1)*40+19,X$(I):NEXTI:IFER=1T
HENER=0:RETURNELSERETURN
78 IFER=1THENER=0:RETURNELSEX$(2)=X$(3):
X$(3)=X$(4):RETURN
79 IFER=1THENER=0:RETURNELSEX$(4)=X$(3):
X$(3)=X$(2):X$(2)=X$(1):RETURN
80 IFK6=1THENK$(6)="lstx deg  RAD  del"E
LSEK$(6)="lstx DEG  rad  del
81 PRINT@5*40,E$+K$(6)+F$:RETURN
82 X$(1)=STR$(2*ATN(VAL(X$(1))/(1+SQR(AB
S(1-VAL(X$(1))^2))))*K6):RETURN
83 BEEP:PRINT@99,"ERROR":ER=1:RESUMENEXT
```

8085 Instruction Mnemonic Meanings

Data Transfer Group

Instruction		Mnemonic Meaning	Flags Zf	Cf	Pf	Sf
MOV	dreg, sreg	MOVe	.	.	.	.
MVI	reg, byte	MoVe Immediate	.	.	.	.
MVX	drp, srp	MoVe eXtended-register (pseudo for high & low MOVs)	.	.	.	.
LXI	rp, word	Load eXtended-register Immediate	.	.	.	.
XCHG		eXCHanGe hl with de	.	.	.	.
LDA	addr	LoaD Accumulator direct	.	.	.	.
STA	addr	STore Accumulator direct	.	.	.	.
LDAX	B	LoaD Accumulator indirect via eXtended-register Bc	.	.	.	.
STAX	B	Store Accumulator indirect via eXtended-register Bc	.	.	.	.
LDAX	D	LoaD Accumulator indirect via eXtended-register De	.	.	.	.
STAX	D	Store Accumulator indirect via eXtended-register De	.	.	.	.
LHLD	addr	Load HL Direct	.	.	.	.
SHLD	addr	Store HL Direct	.	.	.	.
LHLI		Load HL Indirect via extended register de	.	.	.	.
SHLI		Store HL Indirect via extended register de	.	.	.	.

Arithmetic Group

Instruction		Mnemonic Meaning	Flags Zf	Cf	Pf	Sf
ADD	reg	ADD	x	x	x	x
ADI	byte	ADd Immediate	x	x	x	x
ADC	reg	ADd with Carry	x	x	x	x
ACI	byte	Add with Carry Immediate	x	x	x	x
SUB	reg	SUBtract	x	x	x	x
SUI	byte	SUbtract Immediate	x	x	x	x
SBB	reg	SuBtract with Borrow	x	x	x	x
SBI	byte	Subtract with Borrow Immediate	x	x	x	x
DAA		Decimal Adjust Accumulator	x	x	x	x
INR	reg	INcrement Register	x	.	x	x
INX	rp	INcrement eXtended-register	.	.	.	.
DCR	reg	DeCrement Register	x	.	x	x
DCX	rp	DeCrement eXtended-register	.	.	.	.
DAD	rp	Dual-register ADd to hl	.	x	.	.
HLMBC		HL Minus BC	x	x	h	x
DEHL	byte	DE from HL plus byte	.	.	.	.
DESP	byte	DE from SP plus byte	.	.	.	.

Logical Group

Instruction		Mnemonic Meaning	Flags Zf	Cf	Pf	Sf
CMP	reg	CoMPare	x	x	x	x
CPI	byte	ComPare Immediate	x	x	x	x
CMA		CoMplement Accumulator	.	.	.	.
CMC		CoMplement Carry	.	x	.	.
STC		SeT Carry	.	1	.	.
SHLR		Shift HL Right	.	x	.	.
ANA	reg	ANd Accumulator	x	0	x	x
ANI	byte	ANd Immediate	x	0	x	x
ORA	reg	OR Accumulator	x	0	x	x
ORI	byte	OR Immediate	x	0	x	x
XRA	reg	eXclusive oR Accumulator	x	0	x	x
XRI	byte	eXclusive oR Immediate	x	0	x	x
RAL		Rotate Accumulator Left through carry	.	x	.	.
RAR		Rotate Accumulator Right through carry	.	x	.	.
RLC		Rotate accumulator Left Circular	.	x	.	.
RRC		Rotate accumulator Right Circular	.	x	.	.
RDEL		Rotate DE Left through carry	.	x	.	.

Stack, Input/Output, & Machine Control Group

Instruction		Mnemonic Meaning	Flags Zf	Cf	Pf	Sf
PUSH	rp	PUSH on stack	.	.	.	.
POP	rp	POP off stack	.	.	.	.
SPHL		Stack Pointer from HL	.	.	.	.
XTHL		eXchange Top of stack with HL	.	.	.	.
IN	port	INput from port	.	.	.	.
OUT	port	OUTput to port	.	.	.	.
DI		Disable Interrupts	.	.	.	.
EI		Enable Interrupts	.	.	.	.
RIM		Read Interrupt Mask	.	.	.	.
SIM		Set Interrupt Mask	.	.	.	.
NOP		No OPeration	.	.	.	.
HLT		HaLT	.	.	.	.

Branch Group

Instruction		Mnemonic Meaning	Flags Zf	Cf	Pf	Sf
JMP	label	JuMP unconditional	.	.	.	.
JZ	label	Jump if Zero	.	.	.	.
JNZ	label	Jump if No Zero	.	.	.	.
JP	label	Jump if Positive	.	.	.	.
JM	label	Jump if Minus	.	.	.	.
JC	label	Jump if Carry	.	.	.	.
JNC	label	Jump if No Carry	.	.	.	.
JTM	label	Jump if True sign Minus	.	.	.	.
JTP	label	Jump if True sign Positive	.	.	.	.
JPE	label	Jump if Parity Even	.	.	.	.
JPO	label	Jump if Parity Odd	.	.	.	.
CALL	label	CALL unconditioanl	.	.	.	.
CZ	label	Call if Zero	.	.	.	.
CNZ	label	Call if No Zero	.	.	.	.
CP	label	Call if Positive	.	.	.	.
CM	label	Call if Minus	.	.	.	.
CC	label	Call if Carry	.	.	.	.
CNC	label	Call if No Carry	.	.	.	.
CPE	label	Call if Parity Even	.	.	.	.
CPO	label	Call if Parity Odd	.	.	.	.
RET		RETurn unconditional	.	.	.	.
RZ		Return if Zero	.	.	.	.
RNZ		Return if No Zero	.	.	.	.
RP		Return if Positive	.	.	.	.
RM		Return if Minus	.	.	.	.
RC		Return if Carry	.	.	.	.
RNC		Return if No Carry	.	.	.	.
RPE		Return if Parity Even	.	.	.	.
RPO		Return if Parity Odd	.	.	.	.
PCHL		Program Counter from HL	.	.	.	.
RST	n	ReSTart	.	.	.	.
RSTV		ReSTart if oVerflow	.	.	.	.

8085 Instruction Actions by Functional Group

Data Transfer Group

Instruction		Mnemonic Meaning	Flags			
			Zf	Cf	Pf	Sf
MOV	dreg, sreg	dreg<=sreg	.	.	.	.
MVI	reg, byte	reg<=byte	.	.	.	.
MVX	drp, srp	drp<=srp (pseudo for high & low MOVs)	.	.	.	.
LXI	rp, word	rp<=word	.	.	.	.
XCHG		HL<=DE while DE<=HL	.	.	.	.
LDA	addr	A<=b[addr]	.	.	.	.
STA	addr	b[addr]<=A	.	.	.	.
LDAX	B	A<=b[BC]	.	.	.	.
STAX	B	b[BC]<=A	.	.	.	.
LDAX	D	A<=b[DE]	.	.	.	.
STAX	D	b[DE]<=A	.	.	.	.
LHLD	addr	HL<=w[addr]	.	.	.	.
SHLD	addr	w[addr]<=HL	.	.	.	.
LHLI		HL<=w[DE]	.	.	.	.
SHLI		w[DE]<=HL	.	.	.	.

Arithmetic Group

Instruction		Mnemonic Meaning	Flags			
			Zf	Cf	Pf	Sf
ADD	reg	A<=A+reg	x	x	x	x
ADI	byte	A<=A+byte	x	x	x	x
ADC	reg	A<=A+reg+Cf	x	x	x	x
ACI	byte	A<=A+byte+Cf	x	x	x	x
SUB	reg	A<=A-reg	x	x	x	x
SUI	byte	A<=A-byte	x	x	x	x
SBB	reg	A<=A-reg-Cf	x	x	x	x
SBI	byte	A<=A-byte-Cf	x	x	x	x
DAA		in A3..A0 and A7..A4: if >9 then +6, carry to next	x	x	x	x
INR	reg	reg<=reg+1	x	.	x	x
INX	rp	rp<=rp+1	.	.	.	.
DCR	reg	reg<=reg-1	x	.	x	x
DCX	rp	rp<=rp-1	.	.	.	.
DAD	rp	HL<=HL+rp	.	x	.	.
HLMBC		HL<=HL-BC	x	x	h	x
DEHL	byte	DE<=HL+byte	.	.	.	.
DESP	byte	DE<=SP+byte	.	.	.	.

Branch Group

Instruction		Mnemonic Meaning	Flags			
			Zf	Cf	Pf	Sf
JMP	label	PC<=label	.	.	.	.
JZ	label	if Zf=1 then PC<=label	.	.	.	.
JNZ	label	if Zf=0 then PC<=label	.	.	.	.
JP	label	if Sf=0 then PC<=label	.	.	.	.
JM	label	if Sf=1 then PC<=label	.	.	.	.
JC	label	if Cf=1 then PC<=label	.	.	.	.
JNC	label	if Cf=0 then PC<=label	.	.	.	.
JTM	label	if TSf=1 then PC<=label	.	.	.	.
JTP	label	if TSf=0 then PC<=label	.	.	.	.
JPE	label	if Pf=1 then PC<=label	.	.	.	.
JPO	label	if Pf=0 then PC<=label	.	.	.	.
CALL	label	SP<=SP-2, w[SP]<=PC+3, PC<=label	.	.	.	.
CZ	label	if Zf=1 then SP<=SP-2, w[SP] <=PC+3, PC<=label	.	.	.	.
CNZ	label	if Zf=0 then SP<=SP-2, w[SP] <=PC+3, PC<=label	.	.	.	.
CP	label	if Sf=0 then SP<=SP-2, w[SP] <=PC+3, PC<=label	.	.	.	.
CM	label	if Sf=1 then SP<=SP-2, w[SP] <=PC+3, PC<=label	.	.	.	.
CC	label	if Cf=1 then SP<=SP-2, w[SP] <=PC+3, PC<=label	.	.	.	.
CNC	label	if Cf=0 then SP<=SP-2, w[SP] <=PC+3, PC<=label	.	.	.	.
CPE	label	if Pf=1 then SP<=SP-2, w[SP] <=PC+3, PC<=label	.	.	.	.
CPO	label	if Pf=0 then SP<=SP-2, w[SP] <=PC+3, PC<=label	.	.	.	.
RET		PC<=w[SP], SP<=SP+2	.	.	.	.
RZ		if Zf=1 then PC<=w[SP], SP<=SP+2	.	.	.	.
RNZ		if Zf=0 then PC<=w[SP], SP<=SP+2	.	.	.	.
RP		if Sf=0 then PC<=w[SP], SP<=SP+2	.	.	.	.
RM		if Sf=1 then PC<=w[SP], SP<=SP+2	.	.	.	.
RC		if Cf=1 then PC<=w[SP], SP<=SP+2	.	.	.	.
RNC		if Cf=0 then PC<=w[SP], SP<=SP+2	.	.	.	.
RPE		if Pf=1 then PC<=w[SP], SP<=SP+2	.	.	.	.
RPO		if Pf=0 then PC<=w[SP], SP<=SP+2	.	.	.	.
PCHL		PC<=HL	.	.	.	.
RST	n	SP<=SP-2, w[SP]<=PC+1, PC<=n*8 where n is 0 to 7	.	.	.	.
RSTV		if OVf=1 then SP<=SP-2, w[SP] <=PC+1, PC<=8*8	.	.	.	.

Logical Group

Instruction		Mnemonic Meaning	Flags			
			Zf	Cf	Pf	Sf
CMP	reg	T<=A-reg	x	x	x	x
CPI	byte	T<=A-byte	x	x	x	x
CMA		A<=1's complement of A	.	.	.	.
CMC		Cf<=1's complement of Cf	.	x	.	.
STC		Cf<=1	.	1	.	.
SHLR		HL<=HL/2 while H6<=H7 (extend sign) and Cf<=L0	.	x	.	.
ANA	reg	A<=A AND reg	x	0	x	x
ANI	byte	A<=A AND byte	x	0	x	x
ORA	reg	A<=A Inclusive OR reg	x	0	x	x
ORI	byte	A<=A Inclusive OR byte	x	0	x	x
XRA	reg	A<=A Exclusive OR reg	x	0	x	x
XRI	byte	A<=A Exclusive OR byte	x	0	x	x
RAL		A<=A*2 where Cf<=A7 while A0<=Cf	.	x	.	.
RAR		A<=A/2 where Cf<=A0 while A7<=Cf	.	x	.	.
RLC		A7..A1<=A6..A0 while A0<=A7 and Cf<=A7	.	x	.	.
RRC		A6..A0<=A7..A1 while A7<=A0 and Cf<=A0	.	x	.	.
RDEL		DE<=DE*2 where: Cf<=DE15 while DE00<=Cf	.	x	.	.

Stack, Input/Output, & Machine Control Group

Instruction		Mnemonic Meaning	Flags			
			Zf	Cf	Pf	Sf
PUSH	rp	SP<=SP-2, w[SP]<=rp	.	.	.	.
POP	rp	rp<=w[SP], SP<=SP+2	.	.	.	.
SPHL		SP<=HL	.	.	.	.
XTHL		HL<=w[SP] while w[SP]<=HL	.	.	.	.
IN	port	A<=data from port	.	.	.	.
OUT	port	data to port<=A	.	.	.	.
DI		disable interrupts	.	.	.	.
EI		enable interrupts	.	.	.	.
RIM		A<=interrupt mask	.	.	.	.
SIM		interrupt mask<=A	.	.	.	.
NOP		do nothing	.	.	.	.
HLT		halt 8085 processor	.	.	.	.

8085 Instructions by Mnemonic

Instruction		Mnemonic Meaning	Flags			
			Zf	Cf	Pf	Sf
ACI	byte	Add with Carry Immediate	x	x	x	x
ADC	reg	ADd with Carry	x	x	x	x
ADD	reg	ADD	x	x	x	x
ADI	byte	ADd Immediate	x	x	x	x
ANA	reg	ANd Accumulator	x	0	x	x
ANI	byte	ANd Immediate	x	0	x	x
CALL	label	CALL unconditioanl	.	.	.	.
CC	label	Call if Carry	.	.	.	.
CM	label	Call if Minus	.	.	.	.
CMA		CoMplement Accumulator	.	.	.	.
CMC		CoMplement Carry	.	x	.	.
CMP	reg	CoMPare	x	x	x	x
CNC	label	Call if No Carry	.	.	.	.
CNZ	label	Call if No Zero	.	.	.	.
CP	label	Call if Positive	.	.	.	.
CPE	label	Call if Parity Even	.	.	.	.
CPI	byte	ComPare Immediate	x	x	x	x
CPO	label	Call if Parity Odd	.	.	.	.
CZ	label	Call if Zero	.	.	.	.
DAA		Decimal Adjust Accumulator	x	x	x	x
DAD	rp	Dual-register ADd to hl	.	x	.	.
DCR	reg	DeCrement Register	x	.	x	x
DCX	rp	DeCrement eXtended-register	.	.	.	.
DEHL	byte	DE from HL plus byte	.	.	.	.
DESP	byte	DE from SP plus byte	.	.	.	.
DI		Disable Interrupts	.	.	.	.
EI		Enable Interrupts	.	.	.	.
HLMBC		HL Minus BC	x	x	h	x
HLT		HaLT	.	.	.	.
IN	port	INput from port	.	.	.	.
INR	reg	INcrement Register	x	.	x	x
INX	rp	INcrement eXtended-register	.	.	.	.
JC	label	Jump if Carry	.	.	.	.
JM	label	Jump if Minus	.	.	.	.
JMP	label	JuMP unconditional	.	.	.	.
JNC	label	Jump if No Carry	.	.	.	.
JNZ	label	Jump if No Zero	.	.	.	.
JP	label	Jump if Positive	.	.	.	.
JPE	label	Jump if Parity Even	.	.	.	.
JPO	label	Jump if Parity Odd	.	.	.	.
JTM	label	Jump if True sign Minus	.	.	.	.

Instruction		Mnemonic Meaning	Flags			
			Zf	Cf	Pf	Sf
JTP	label	Jump if True sign Positive	.	.	.	.
JZ	label	Jump if Zero	.	.	.	.
LDA	addr	LoaD Accumulator direct	.	.	.	.
LDAX	B	LoaD Accumulator indirect via eXtended-register Bc	.	.	.	.
LDAX	D	LoaD Accumulator indirect via eXtended-register De	.	.	.	.
LHLD	addr	Load HL Direct	.	.	.	.
LHLI		Load HL Indirect via extended register de	.	.	.	.
LXI	rp, word	Load eXtended-register Immediate	.	.	.	.
MOV	dreg, sreg	MOVe				
MVI	reg, byte	MoVe Immediate	.	.	.	.
MVX	drp, srp	MoVe eXtended-register (pseudo for high & low MOVs)	.	.	.	.
NOP		No OPeration	.	.	.	.
ORA	reg	OR Accumulator	x	0	x	x
ORI	byte	OR Immediate	x	0	x	x
OUT	port	OUTput to port	.	.	.	.
PCHL		Program Counter from HL	.	.	.	.
POP	rp	POP off stack	.	.	.	.
PUSH	rp	PUSH on stack	.	.	.	.
RAL		Rotate Accumulator Left through carry	.	x	.	.
RAR		Rotate Accumulator Right through carry	.	x	.	.
RC		Return if Carry	.	.	.	.
RDEL		Rotate DE Left through carry	.	x	.	.
RET		RETurn unconditional	.	.	.	.
RIM		Read Interrupt Mask	.	.	.	.
RLC		Rotate accumulator Left Circular	.	x	.	.
RM		Return if Minus	.	.	.	.
RNC		Return if No Carry	.	.	.	.
RNZ		Return if No Zero	.	.	.	.
RP		Return if Positive	.	.	.	.
RPE		Return if Parity Even	.	.	.	.
RPO		Return if Parity Odd	.	.	.	.
RRC		Rotate accumulator Right Circular	.	x	.	.
RST	n	ReSTart	.	.	.	.
RSTV		ReSTart if oVerflow	.	.	.	.
RZ		Return if Zero	.	.	.	.
SBB	reg	SuBtract with Borrow	x	x	x	x
SBI	byte	Subtract with Borrow Immediate	x	x	x	x

Instruction		Mnemonic Meaning	Flags			
			Zf	Cf	Pf	Sf
SBI	byte	Subtract with Borrow Immediate	x	x	x	x
SHLD	addr	Store HL Direct	.	.	.	.
SHLI		Store HL Indirect via extended register de	.	.	.	.
SHLR		Shift HL Right	.	x	.	.
SIM		Set Interrupt Mask	.	.	.	.
SPHL		Stack Pointer from HL	.	.	.	.
STA	addr	STore Accumulator direct	.	.	.	.
STAX	B	Store Accumulator indirect via eXtended-register Bc	.	.	.	.
STAX	D	Store Accumulator indirect via eXtended-register De	.	.	.	.
STC		SeT Carry	.	1	.	.
SUB	reg	SUBtract	x	x	x	x
SUI	byte	SUbtract Immediate	x	x	x	x
XCHG		eXCHanGe hl with de	.	.	.	.
XRA	reg	eXclusive oR Accumulator	x	0	x	x
XRI	byte	eXclusive oR Immediate	x	0	x	x
XTHL		eXchange Top of stack with HL	.	.	.	.

8085 Machine Cycles by Functional Group

Data Transfer Group

		Essential Cycles +register M involved or condition Met	EC +MM
Instruction		Mnemonic Meaning	Cycles
MOV	dreg, sreg	MOVe	04 +03
MVI	reg, byte	MoVe Immediate	07 +03
MVX	drp, srp	MoVe eXtended-register (pseudo for high & low MOVs)	--
LXI	rp, word	Load eXtended-register Immediate	10
XCHG		eXCHanGe hl with de	04
LDA	addr	LoaD Accumulator direct	13
STA	addr	STore Accumulator direct	13
LDAX	B	LoaD Accumulator indirect via eXtended-register Bc	07
STAX	B	Store Accumulator indirect via eXtended-register Bc	07
LDAX	D	LoaD Accumulator indirect via eXtended-register De	07
STAX	D	Store Accumulator indirect via eXtended-register De	07
LHLD	addr	Load HL Direct	16
SHLD	addr	Store HL Direct	16
LHLI		Load HL Indirect via extended register de	10
SHLI		Store HL Indirect via extended register de	10

Arithmetic Group

		Essential Cycles +register M involved or condition Met	EC +MM
Instruction		Mnemonic Meaning	Cycles
ADD	reg	ADD	04 +03
ADI	byte	ADd Immediate	07
ADC	reg	ADd with Carry	04 +03
ACI	byte	Add with Carry Immediate	07
SUB	reg	SUBtract	04 +03
SUI	byte	SUbtract Immediate	07
SBB	reg	SuBtract with Borrow	04 +03
SBI	byte	Subtract with Borrow Immediate	07
DAA		Decimal Adjust Accumulator	04
INR	reg	INcrement Register	04 +06
INX	rp	INcrement eXtended-register	06
DCR	reg	DeCrement Register	04 +06
DCX	rp	DeCrement eXtended-register	06
DAD	rp	Dual-register ADd to hl	10
HLMBC		HL Minus BC	10
DEHL	byte	DE from HL plus byte	10
DESP	byte	DE from SP plus byte	10

Logical Group

		Essential Cycles +register M involved or condition Met	EC +MM
Instruction		Mnemonic Meaning	Cycles
CMP	reg	CoMPare	04 +03
CPI	byte	ComPare Immediate	07
CMA		CoMplement Accumulator	04
CMC		CoMplement Carry	04
STC		SeT Carry	04
SHLR		Shift HL Right	07
ANA	reg	ANd Accumulator	04 +03
ANI	byte	ANd Immediate	07
ORA	reg	OR Accumulator	04 +03
ORI	byte	OR Immediate	07
XRA	reg	eXclusive oR Accumulator	04 +03
XRI	byte	eXclusive oR Immediate	07
RAL		Rotate Accumulator Left through carry	04
RAR		Rotate Accumulator Right through carry	04
RLC		Rotate accumulator Left Circular	04
RRC		Rotate accumulator Right Circular	04
RDEL		Rotate DE Left through carry	10

Stack, Input/Output, & Machine Control Group

		Essential Cycles +register M involved or condition Met	EC +MM
Instruction		Mnemonic Meaning	Cycles
PUSH	rp	PUSH on stack	12
POP	rp	POP off stack	10
SPHL		Stack Pointer from HL	06
XTHL		eXchange Top of stack with HL	16
IN	port	INput from port	10
OUT	port	OUTput to port	10
DI		Disable Interrupts	04
EI		Enable Interrupts	04
RIM		Read Interrupt Mask	04
SIM		Set Interrupt Mask	04
NOP		No OPeration	04
HLT		HaLT	05

Branch Group

		Essential Cycles +register M involved or condition Met	EC +MM
Instruction		Mnemonic Meaning	Cycles
JMP	label	JuMP unconditional	10
JZ	label	Jump if Zero	07 +03
JNZ	label	Jump if No Zero	07 +03
JP	label	Jump if Positive	07 +03
JM	label	Jump if Minus	07 +03
JC	label	Jump if Carry	07 +03
JNC	label	Jump if No Carry	07 +03
JTM	label	Jump if True sign Minus	07 +03
JTP	label	Jump if True sign Positive	07 +03
JPE	label	Jump if Parity Even	07 +03
JPO	label	Jump if Parity Odd	07 +03
CALL	label	CALL unconditional	18
CZ	label	Call if Zero	09 +09
CNZ	label	Call if No Zero	09 +09
CP	label	Call if Positive	09 +09
CM	label	Call if Minus	09 +09
CC	label	Call if Carry	09 +09
CNC	label	Call if No Carry	09 +09
CPE	label	Call if Parity Even	09 +09
CPO	label	Call if Parity Odd	09 +09
RET		RETurn unconditional	10
RZ		Return if Zero	06 +06
RNZ		Return if No Zero	06 +06
RP		Return if Positive	06 +06
RM		Return if Minus	06 +06
RC		Return if Carry	06 +06
RNC		Return if No Carry	06 +06
RPE		Return if Parity Even	06 +06
RPO		Return if Parity Odd	06 +06
PCHL		Program Counter from HL	06
RST	n	ReSTart	12
RSTV		ReSTart if oVerflow	06 +06

8085 Instruction Mnemonics by Op-code

	x0h	x1h	x2h	x3h	x4h	x5h	x6h	x7h	x8h	x9h	xAh	xBh	xCh	xDh	xEh	xFh
00h-0Fh	NOP	LXI B,w	STAX B	INX B	INR B	DCR B	MVI B,b	RLC	HLMBC	DAD B	LDAX B	DCX B	INR C	DCR C	MVI C,b	RRC
10h-1Fh	SHLR	LXI D,w	STAX D	INX D	INR D	DCR D	MVI D,b	RAL	RDEL	DAD D	LDAX D	DCX D	INR E	DCR E	MVI E,b	RAR
20h-2Fh	RIM	LXI H,w	SHLD	INX H	INR H	DCR H	MVI H,b	DAA	DEHL b	DAD H	LHLD	DCX H	INR L	DCR L	MVI L,b	CMA
30h-3Fh	SIM	LXI SP,w	STA @	INX SP	INR M	DCR M	MVI M,b	STC	DESP b	DAD SP	LDA @	DCX SP	INR A	DCR A	MVI A,b	CMC
40h-4Fh	MOV B,B	MOV B,C	MOV B,D	MOV B,E	MOV B,H	MOV B,L	MOV B,M	MOV B,A	MOV C,B	MOV C,C	MOV C,D	MOV C,E	MOV C,H	MOV C,L	MOV C,M	MOV C,A
50h-5Fh	MOV D,B	MOV D,C	MOV D,D	MOV D,E	MOV D,H	MOV D,L	MOV D,M	MOV D,A	MOV E,B	MOV E,C	MOV E,D	MOV E,E	MOV E,H	MOV E,L	MOV E,M	MOV E,A
60h-6Fh	MOV H,B	MOV H,C	MOV H,D	MOV H,E	MOV H,H	MOV H,L	MOV H,M	MOV H,A	MOV L,B	MOV L,C	MOV L,D	MOV L,E	MOV L,H	MOV L,L	MOV L,M	MOV L,A
70h-7Fh	MOV M,B	MOV M,C	MOV M,D	MOV M,E	MOV M,H	MOV M,L	HLT	MOV M,A	MOV A,B	MOV A,C	MOV A,D	MOV A,E	MOV A,H	MOV A,L	MOV A,M	MOV A,A
80h-8Fh	ADD B	ADD C	ADD D	ADD E	ADD H	ADD L	ADD M	ADD A	ADC B	ADC C	ADC D	ADC E	ADC H	ADC L	ADC M	ADC A
90h-9Fh	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB M	SUB A	ANA B	ANA C	ANA D	ANA E	ANA H	ANA L	ANA M	ANA A
A0h-AFh	ANA B	ANA C	ANA D	ANA E	ANA H	ANA L	ANA M	ANA A	XRA B	XRA C	XRA D	XRA E	XRA H	XRA L	XRA M	XRA A
B0h-BFh	ORA B	ORA C	ORA D	ORA E	ORA H	ORA L	ORA M	ORA A	CMP B	CMP C	CMP D	CMP E	CMP H	CMP L	CMP M	CMP A
COh-CFh	RNZ	POP B	JNZ @	JMP @	CNZ @	PUSH B	ADI b	RST 0	RZ	RET	JZ @	RSTV	CZ @	CALL @	ACI b	RST 1
D0h-DFh	RNC	POP D	JNC @	OUT port	CNC @	PUSH D	SUI b	RST 2	RC	SHLI	JC @	IN port	CC @	JTP @	SBI b	RST 3
E0h-EFh	RPO	POP H	JPO @	XTHL	CPO @	PUSH H	ANI b	RST 4	RPE	PCHL	JPE @	XCHG	CPE @	LHLI	XRI b	RST 5
F0h-FFh	RP	POP PSW	JP @	DI	CP @	PUSH PSW	ORI b	RST 6	RM	SPHL	JM @	EI	CM @	JTM @	CPI b	RST 7