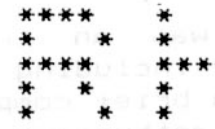

GAZETTE

The Official Newsletter of the
Richmond Heath Users Group

Volume II Issue 1
JANUARY 1984

Subscriptions: $12.00 per calendar year.
Editor: Dave Harrington, 10813 Lunswood Rd, Chester VA, 23831
Next editorial deadline: February 6, 1984.
Note to other HUGs: If you want to trade newsletters with us, please mail yours to Harold Lanna, 600 Watch Hill Rd, Midlothian, VA 23113

MEETING NOTICE

The next meeting will be Monday, January 16, at 7:30. The meeting location is Alpha Audio's third floor conference room, at 2049 West Broad Street. The night-time phone number there is 358-3853. The front door has a touch-pad combination lock, and the combination for the night will be 5082 (five zero eight two).

Everyone is welcome!

MINUTES (Meeting of December 19, 1983)

Present: Carlos Chafin, Dave Harrington, Harold Lanna, John Purcell, Jim Scott, Ron Stauffer, Nelson Trinkle and Parks Watson.

ELECTION OF OFFICERS. There were no additional nominations from the floor and the following officers nominated at the November meeting were elected unanimously:

Harold Lanna . . . . . President
John Purcell . . . . . Vice-President
Carlos Chafin. . . . . Secretary/Treasurer
Nelson Trinkle . . . . Software Librarian
Dave Harrington. . . . Newsletter Editor

At Jim Scott's prompting, Watson stated that the treasury was always in the receiving mode, with the result that 7 of the eight members present coughed up $12.00 each for 1984 dues. Also one member, Hank Steigleder had mailed in his check for a total of eight members paid for the new year.

A discussion of the question of moving up future nominations and elections of officers to meet the REMark publication deadline indicated that it should be deferred to a later date.

The program for the evening was an introduction to the C programming language by Carlos Chafin, including the general overview of the design of the language, and a brief comparison of C compilers from three different sources. Carlos' enthusiasm for C was evident, but tempered by the warning that tutorial type publications on the subject were practically non-existent.

The meeting adjourned at 10:00 PM.

Parks Watson
Secretary/Treasurer

****

NEWS

MAILING OF NEWSLETTERS

This month is the last one in which this newsletter will be mailed to persons who have not paid dues. If you have not paid your dues, you will no longer receive this newsletter. New (prospective) members will still have an opportunity to receive one issue, without payment of dues. This is a simple matter of economics. No Tickee??--No Shirtee!! (If you can't attend the next meeting and want to pay your dues by mail, send a check to Carlos or Harold Lanna.)

****
****

ASSEMBLY LANGUAGE PROGRAMMING - PART 7

by Jim Scott

INTRODUCTION

This is the seventh of a series of articles which parallel and summarize the discussions about assembly language at our meetings. The purpose of the discussions and the articles is to present enough information about assembly language programming so that someone who knows how to program in a higher-level language, and is willing to use the proper manuals for reference, will at least have some idea how to get started at programming in assembly language.

The previous four articles (August, September, October, and December issues of the Gazette) described the Data Transfer, Arithmetic, Logical, and Branch groups of instructions for the 8080 CPU. Instructions in these groups move data between registers and memory, perform arithmetic and logical operations on data in registers and memory, and alter the normal sequence of program flow. This time we will discuss the final group of instructions, the Stack, I/O, and Machine Control Group.

STACK, I/O, AND MACHINE CONTROL GROUP

Instructions in this group manipulate the stack, perform I/O (input and output), and alter internal control flags.

Before considering the individual instructions, we need to learn something about what a stack is. (I've put this off as long as I could but now we've got to do it. Oh well, it won't be so bad - you'll see.)

Article 2 in this series (see the July issue of the Gazette) defined the Stack Pointer register pair (SP) as follows:

SP Stack Pointer — Contains the address of the memory location which is the current "stack" location. The stack is a set of consecutive memory locations which are used for temporarily saving data so that it can be retrieved later. Instructions such as PUSH, POP, CALL, and RET make special use of the stack pointer.

The stack may be thought of as a stack of cafeteria trays with a spring under the bottom of the stack to keep the top of the stack more or less level with the top of the counter. This is sometimes called a "push-down stack" or a "last-in-first-out (LIFO) queue". When a cafeteria employee adds a tray (supposedly clean) to the stack, it is added to the top; the rest of the stack is pushed down by one tray-thickness. This is called a "push" operation (by us programmers, that is). When a customer takes a tray, it is taken from the top; the spring makes the rest of the stack rise by one tray-thickness. This is called a "pop" operation; a tray has been popped off of the stack. The last tray in is the first one out.

The stack in the 8080 is, as mentioned above, a set of consecutive memory locations. Now, there is no magic place in memory determined by the computer factory to be the stack. Any place in memory will do (as long as that area in memory isn't already being used for something else). When the machine is cold-booted, one of the things that happens is that a stack is set up. This means that an address is moved into the Stack Pointer register pair (SP); the address put into SP at this point should be the address of the byte just above the top of the memory area allocated to serve as the stack. For example, if memory locations F801 to FFFE are to be used as the stack, then SP should initially be set to FFFF.

Each "tray" of the 8080 stack holds 16 bits; i.e., the contents of one register pair, or a sixteen-bit address, can be put onto the stack in one operation. As you may remember from the preceding article in this series, the CALL and Ccc instructions push the address of the next sequential instruction onto the stack; the RET and Rcc instructions pop an address off of the stack and put it into PC (the Program Counter register pair). The PUSH, POP, XTHL, and SPHL instructions, described below, move data 16 bits at a time between the stack and a register pair. When an item is pushed onto the stack, the address in SP is decreased by 2, to show that the current top of the stack is two bytes (16 bits) lower than it used to be. When a pop takes place, 2 is added to SP. Thus SP always points to the current top of the stack, even if this is a lower memory address than the original top of the stack.

It is important to remember that items pushed onto the stack must be popped off in the opposite order. For example, if a subprogram begins by saving the contents of the BC, DE, and HL register pairs

```
PUSH  B
PUSH  D
PUSH  H
```

then it must terminate by restoring them in the opposite order.

```
POP   H
POP   D
POP   B
RET
```

(The RET pops off the return address which was pushed by the CALL that invoked this subprogram.) If the POPs were done in another order, the wrong values would get into the wrong register pairs. If the RET were done without doing any POPs, the return would be to the wrong address (whatever was in HL when the subprogram was entered).

Just as there is no magic place where the stack has to be located, it is also quite possible to have more than one stack (although SP will only point to one stack at a time). If you write an assembly language program that will make use of any instructions that manipulate the stack (and it would be pretty difficult to avoid these instructions), you need to consider what stack you will use. When your program begins execution, SP already contains an address. This will be the current top of the stack set up by the operating system. How much space is available on this stack? There is no guarantee. How much stack space will your program use? That depends on how many CALLs and PUSHes you do, and whether your RETs and POPs are interspersed with them or not. (If you do ten PUSHes before doing a POP, then you need at least 20 bytes for your stack; but if you do PUSH / POP ten times in a row, this will reuse the same two bytes of the stack ten times.) It also depends on what operating system routines you may call (e.g., SCALLs in HDOS, or CALL BDOS in CP/M) and how these system routines use the stack.

Generally, it is safer to set up your own stack area, although many assembly language programmers do not do this, and never have any trouble. You can set up your own stack by using a DS assembler directive to reserve the space for the stack (usually a couple of hundred bytes should be plenty). Near the beginning of the program, store into SP the address of the next byte beyond this area. If you are going to terminate your program by doing a JMP to the operating system BOOT routine, then you don't need to save the operating system's stack address; otherwise, you will need to save the original value of SP (when your program begins execution), and restore SP to this value just before doing the RET instruction that terminates your program.

What marks the bottom of the stack? Nothing! If you push more onto the stack than it will hold, the address in SP keeps getting lower by 2 bytes at a time, and each additional push writes over whatever data or program code happens to lie just below the allocated stack area. This can easily cause what the computer manuals like to refer to as "unpredictable results" (what else could they call it?). If an assembly language program you wrote goes off into never-never land during testing, consider the possibility that it may need a bigger stack.

Now you know everything about stacks. Let's get down to business.

The individual instructions are as follows.

PUSH (Push)

Format: PUSH rp

where rp can be B, D, H, or PSW, representing register pairs BC, DE, HL, and the Processor Status Word, respectively. (The Processor Status Word consists of the accumulator (A) and the flag word (F); the high-order half of PSW is A.) This instruction pushes the register pair onto the stack. It does this as follows: it moves the high-order register to the memory location whose address is one less than the contents of SP; it moves the low-order register to the memory location whose address is two less than the contents of SP; then it subtracts two from the contents of SP.

Example: PUSH D

This pushes the contents of register pair DE onto the top of the stack.

This instruction is used for remembering a 16-bit value. The value will presumably be recalled at a later time, by executing a POP D instruction.

POP (Pop)

Format: POP rp

where rp can be B, D, H, or PSW, representing register pairs BC, DE, HL, and the Processor Status Word, respectively. (The Processor Status Word consists of the accumulator (A) and the flag word (F); the high-order half of PSW is A.) This instruction pops the register pair from the stack. It does this as follows: it moves the contents of the memory location, whose address is specified by the contents of SP, to the low-order register of rp; it moves the contents of the next memory location, whose address is one more than the contents of SP, to the high-order register of rp; then it adds two to the contents of SP.

Example: POP B

This pops the contents of the top of the stack into register pair BC.

This instruction is used for recalling a 16-bit value which presumably was remembered at an earlier time, by executing a PUSH B instruction.

XTHL (Exchange Stack Top with HL)

Format: XTHL

This instruction exchanges the contents of HL with the item on the top of the stack. It does this by exchanging the contents of the L register with the contents of the memory location whose address is specified by the contents of SP; it exchanges the contents of the H register with the contents of the memory location whose address is one more than the contents of SP. It does not change the value of SP.

Example: XTHL

This exchanges the contents of HL with whatever was most recently pushed onto the stack.

This instruction is not used very often. It can be used to change the value of the item on the top of the stack without having to do a PUSH. It can also be used to retrieve the value of the item on the top of the stack without having to do a POP, but it is necessary to remember that XTHL changes the stack; the instruction sequence XTHL / XCHG / XTHL could be used to get the value of the item on the top of

the stack into DE without making a permanent change to the stack.

SPHL (Move HL to SP)

Format: SPHL
This instruction moves the contents of HL into SP.

Example: SPHL

This instruction is used to set up a new stack. The address of the byte just above the top of the stack is loaded into HL, then the SPHL instruction is executed. An LXI SP instruction could also be used, but only if the address of the stack is an absolute address. Usually the address is known only at execution time, or is calculated, so the SPHL instruction is used to set SP.

IN (Input)

Format: IN port
This instruction receives an 8-bit value from the port (I/O device) whose number is specified in the second byte of the instruction, and puts the value into the accumulator (register A).

Example: IN 360Q
This reads a byte value from the front panel keypad on the H8 computer.

At the lowest level, this instruction is used to do all input. How it works depends on the device represented by the port. Which port number represents which device depends on a number of hardware considerations.

OUT (Output)

Format: OUT port
This instruction sends an 8-bit value from the accumulator (register A) to the port (I/O device) whose number is specified in the second byte of the instruction.

Example: OUT 371Q
This sends the contents of A to the magnetic tape device, if any, on the H8 computer.

At the lowest level, this instruction is used to do all output. How it works depends on the device represented by the port. Which port number represents which device depends on a number of hardware considerations.

EI (Enable Interrupts)

Format: EI
This instruction enables the computer to take interrupts following the execution of the next instruction.

Example: EI

This instruction undoes the effect of the DI instruction (see below).

DI (Disable Interrupts)

Format: DI

This instruction disables the computer from taking interrupts, immediately following the execution of the DI instruction.

Example: DI

This instruction suspends the ability of the computer to accept interrupts. A complete discussion of interrupts is beyond the scope of this article. But the main thing to know is that all I/O operations require interrupts. This includes disk reads, disk writes, displaying characters on the screen, and accepting input from the terminal keyboard or the computer front panel keypad. While interrupts are disabled, no I/O can take place.

Possible reasons for disabling interrupts include temporarily allowing the system to run faster (by preventing the additional processing necessary to handle interrupts), or allowing time-dependent code to run without being interrupted.

HLT (Halt)

Format: HLT

This instruction stops the computer. It may be started again only by an interrupt or a hardware restart.

Example: HLT

This instruction has very little actual use. Generally, a program will terminate by executing a RET or JMP instruction rather than HLT. The operating system itself, if it has nothing to do, will also not do a HLT; it will go into a loop, waiting for input from the keyboard.

NOP (No Operation)

Format: NOP

This instruction does nothing, and has no affect on any registers or memory locations.

This instruction may be included in an assembly language program to leave room for "patches". It may also be patched into the object code for an existing program, to eliminate other instructions. The object code for NOP is a byte value of zero.

CONCLUSION

This completes the discussion of the 8080 instruction set. The next article will continue the discussion of assembly language programming for the 8080.

****
****