Assembly Language and the Stack on the IMSAI 8080

by Bruce E. Hall, W8BH

"When one door closes, another door opens," I thought as I finished my most recent article. One of the assembly language programs was behaving badly. It worked fine but as soon as it returned to CP/M, things got *weird*. Sometimes my system froze. Sometimes I got strange system messages. Sometimes it was OK – for a while. Intermittent errors can be hard to fix. Certainly, a new door had opened, and it was time to find the error lurking behind it.

Stack Basics

My problem was a stack overflow. For me, the stack was just one of those things that is supposed to work: call a routine, and the program counter is put on the stack. Finish the routine, the program counter is pulled from the stack, and execution resumes. You can push data onto the stack and POP is back off, last-in first-out (LIFO). The stack sits somewhere high in the memory address space, pointed to by the stack pointer register, and it grows downward as it's used. That was the extent of my understanding. A quick review of stack basics is provided here: stack basics video. The rest of this article will look at the stack as it applies to the IMSAI 8080 and CP/M, and describe several ways to avoid a stack overflow error.

Clearly, CALL/RET and PUSH/PULL instructions change the value of the stack pointer register. But there are a lot of additional Z80 instructions that directly use the stack pointer.

```
LD SP, nn     ; Load the stack pointer with 16 bit number NN
LD SP, IX     ; Load the stack pointer with IX (or IY)
LD SP, HL     ; Load the stack pointer with HL
LD SP, (aa)   ; Load the stack pointer with value at location AA
LD (aa), SP   ; Store stack pointer at location AA
INC SP        ; Increment the stack pointer
DEC SP        ; Decrement the stack pointer
```

```
ADD HL, SP    ; Add the stack pointer value to HL
ADD IX, SP    ; Add the stack pointer to IX (or IY)
ADC HL, SP    ; Add the stack pointer value to HL
SBC HL, SP    ; Subtract stack pointer from HL
EX (SP), HL   ; Exchange the value at SP with HL
EX (SP), IX   ; Exchange the value at SP with IX (or IY)
```

From the instructions above, we see that the stack pointer can be set in several different ways. Interestingly, you cannot directly copy the stack pointer to another register. You'll need at least two instructions, like this:

```
LD  HL,0      ; HL = 0
ADD HL,SP     ; now HL = SP
```

I started wondering about the stack, as it applied to CP/M. My first question was...

Where is the stack when a CP/M program starts?

To find out, I wrote a small program that prints the stack pointer:

```
LD     HL,0
ADD    HL,SP         ;get SP into HL
CALL   printHex16    ;and display value on console
RET                  ;return to CP/M
```

The answer is C3A9 on my 54K system and E3A9 on my 62K system. It is located near the end of the CCP, just before the start of BDOS. Why there? I think the answer lies in the transient nature of CCP itself. The memory space it occupies may be used by other programs, and then CCP reloaded afterwards. But how does CP/M know when the CCP needs to be reloaded?

Answer: it doesn't. That's on you. You decide when to use that space and reload CCP.

How big is the stack?

This is the 64K question. If the stack is located within CP/M, how many entries can it hold before it starts corrupting CP/M? The answer is found in the CP/M Operating Manual, chapter 5:

> *"Upon entry to a transient program, the CCP leaves the stack pointer set to an **eight-level stack area** with the CCP return address pushed onto the stack, leaving **seven levels before overflow occurs**. Although this stack is usually not used by a transient program (most transients return to the CCP through a jump to location 0000H) it is large enough to make CP/M system calls because the FDOS switches to a local stack at system entry."*

Look carefully at the CCP source code (CCP source code). The stack is located near the end, as expected, in the data section. Only 16 bytes are reserved for the stack, which allows for 8 two-byte addresses. CCP pushes its return address on the stack, leaving 7 stack levels for your code. Any stack entries beyond the 7th will overwrite and corrupt the CCP.

For simple programs without subroutines, a 7-level stack is often enough. Add a few subroutines, however, and you can quickly run out of stack space. Consider the following:

```
Routine1:
    PUSH   BC
    PUSH   DE
    PUSH   HL           ;save registers
    CALL   Routine2     ;do something
    POP    HL           ;restore registers
    POP    DE
    POP    BC
    RET
```

This small routine uses 5 level of stack: one to call it, 3 to push registers onto the stack, and one more to call Routine2.

There is no helpful message that tells you, "Oops, you just ran out of stack space!" Instead, your system may freeze or act strangely. You may end up spending hours poring over your code, looking for the error. Stack overflow is not an easy error to find. To see what happens when your stack overflows, check out sample program stack0.asm.

Bottom Line: *your assembly language program should never cause a stack overflow. Practice good stack hygiene with one of the methods below.*

→ **Method 1: Don't return; jump to 0000 instead.**

The easiest way to warm boot the system is a jump to address 0000h. There are two options: RST 00 and JP 0. The reset instruction is shorter (it is a one byte instruction). The jump instruction, although 3 bytes length, is easily read and comes with conditional variants. A third alternative is calling BDOS function 0. All three warm boot CP/M. See example stack1.asm.

```
<code here>          <code here>          ; your program
JP 0000h      -or-   RST 00               ; return to CP/M via warm boot
```

Method 2: Push a zero on the stack

For existing programs, there is no need to find and change all return instruction(s) with a JP 0000 or RST 00. Instead, push the return address of 0000h onto the existing stack. When the program encounters the return instruction, it will continue execution at 0000h and trigger a warm boot. See example stack2.asm.

```
LD HL,0
PUSH HL              ; push 0000 onto the stack
< . . . >            ; all your code here
RET                  ; return to CP/M via warm boot
```

The methods above are super-simple and keep your stack in high memory, far away from your program. Little or no additional coding is required. Their primary disadvantage is that CCP must be reloaded from disk at the termination of your program.

The next two methods create a new stack. The original CCP stack is preserved, allowing us to return to it without having to reload the CCP.

Method 3: Create a new stack in the TPA

```
LD     HL,0
ADD    HL, SP        ; Save original SP in HL
LD     SP, 2000h     ; Establish new stack at 2000
PUSH   HL            ; Push original SP onto new stack

< . . . >            ; all your code here

POP    HL            ; Get original SP off the local stack
LD     SP, HL        ; Restore the original (CCP's) SP
RET                  ; and use it to return to CP/M
```

The *LD SP, nnnn* instruction loads the stack pointer with a specific memory address. Choose a spot well above your code. For example, if your code is 7K in size, the new stack will be safe 1K above it, at 2000h. See example stack3.asm.

Notice how a PUSH/POP pair is used to save the original stack pointer. You could use a memory location instead:

Using a memory location

```
LD HL,0
ADD HL, SP
LD SP, 1000h
LD (oldstack), HL
```

Using the stack (as above)

```
LD HL,0              ; cant load SP directly
ADD HL, SP           ; get old SP into HL
LD SP, 1000h         ; establish new SP
PUSH HL              ; save old SP
```

```
< . . . >                       < . . . >           ; all your code here

LD HL, (oldstack)               POP HL              ; retrieve old SP into HL
LD SP, HL                       LD SP, HL           ; restore old stack
RET                             RET                 ; and return to CP/M
```

Either way is acceptable. I saved the old SP on stack, but you can just as easily save the old SP in memory. Chapter 5 of the CPM 2.2 Manual gives an example of the latter.

→ Method 4: Create a new stack near your code.

This code for is nearly identical to Method 3, except that it keeps the code and stack close together. The assembler reserves space for the stack. In essence, it becomes part of your program. You must decide in advance how much stack space is needed and allow for the required space. In most cases the stack size is not critical. A stack size of 64 allows for 32 stack entries, more than quadrupling the amount of stack CCP provides, and will accommodate most needs. See example stack4.asm.

```
      LD     HL,0
      ADD    HL, SP          ; Temp save original SP in HL
      LD     SP, STACK       ; Point to the new Stack.
      PUSH   HL              ; Push original SP on new stack

      < . . . >              ; all your code here

      POP    HL              ; Get original SP off the local stack
      LD     SP, HL          ; Restore the original (CCP's) SP
      RET                    ; and use it to return to CP/M

      .DS 64                 ; space for 32 stack entries
STACK:                       ; stack grows downward from here
      .END
```

→ Method 5: Mix and Match

You may combine one or more of the techniques above. For example, you can create a local stack (Method 3) ***and*** do a warm reboot (Method 1). See example stack5.asm.

```
      LD     SP, 0           ; Establish a new stack at FFFF
      < . . . >              ; all your code here
      JP     0000            ; return to CP/M via warm boot
```

This method gives you control over your own stack without the hassle of restoring the old one. The code above puts the stack at the top of memory (Note that setting the stack pointer at 0 means that the first entry on the stack will be at FFFE-FFFF). If high memory is used by a video display driver or other software, you can designate a different location via the *LD SP* instruction.

The final two methods below work for CP/M 2.2 but may not be suitable for other CP/M versions or alternatives.

→ Method 6: Create a stack just below the CCP.

This stack maximizes stack space while preserving the current copy of CCP. It assumes that the CCP is 1600h bytes below the BIOS, which is true for CP/M 2.2. See example stack6.asm.

```
LD      IX,0
ADD     IX, SP          ; Temp save original SP in IX
LD      HL, (0001)      ; Get warm boot vector = BIOS+3
LD      BC,1603h
SBC     HL,BC           ; Calculate start of CCP = BIOS-1600h
LD      SP, HL          ; Point to the new Stack.
PUSH    IX              ; Push original SP on new stack

< . . . >               ; all your code here

POP     HL              ; Get original SP off the local stack
LD      SP, HL          ; Restore the original (CCP's) SP
RET                     ; and use it to return to CP/M
```

→ Method 7: Return to CCP without restoring its stack.

That sounds dangerous, doesn't it? But look at the following snippet of code from the CCP in CP/M 2.2:

```
CALL    TPA             ; execute user program at 0100h
LXI     SP, STACK       ; reset stack if user program returns here
```

The stack is immediately reset after returning from the user's program. Therefore, there is no need for our code to preserve the stack. If we return to the correct spot, all will be well. It is very efficient and doesn't require a warm boot. See example stack7.asm.

```
POP     HL              ; Get CCP return address
LD      SP, 2000h       ; Establish new stack at 2000
PUSH    HL              ; Push address onto new stack
< . . . >               ; all your code here
RET                     ; return to CP/M
```

Best Practices

What is the best way to gracefully terminate your assembly language code? It depends. For very small programs, a simple return instruction will do. However, *most programs should return to CP/M via warm boot*. Lesson learned.

It is also considered "best practice" for programs to establish their own stack. CP/M doesn't require it, but what if you are running your code outside CP/M? Or in a debugger? Method 5 shows a simple way of establishing a local stack and restarting CP/M.

Finally, if the time penalty of a warm boot bothers you, a few extra lines of code will provide a swift and safe return to CP/M, keeping the command processor and its stack intact.

Happy Hacking!

Bruce.

Last Updated: 12 July 2025

Other articles in this series: IMSAI 8080 articles