Lab Report

LAB 13_ STACK

I/- OBJECTIVE:

Understand the usage and operation of Stack and Stack Pointer (LIFO)
Using PUSH & POP instructions.

II/- TOOL – Software :

Download Attached Files : Lab 13 (Stack).zip (20.688 KB)
From : **Lab 13 (Stack)** in Blackboard .
Open file Stack type IAR IDE Workspace
Edit file asm.s43

- The window of file asm.s43 and lines of Instruction.

asm.s43

```
1  #include "msp430.h"                    ; #define controlled include file
2
3          ORG     0F800h
4
5  init:   MOV.W   #0280h, SP             ; set up stack
6          MOV.W   #WDTPW+WDTHOLD,&WDTCTL  ; Stop watchdog timer
7          MOV.W   #0000h, R5
8          MOV.W   #000h, R4               ; loop counter for main loop
9
10 main:   NOP                             ; main program
11
12         CMP.W #005, R5
13         JHS POPTIME
14
15         ADD.W   #001h, R4              ; Increment loop counter
16         PUSH  R4
17
18         ADD.W   #001h, R5
19
20         JMP   main
21
22 POPTIME:
23         POP   R14
24         POP   R14
25         POP   R14
26         POP   R14
27         POP   R14
28         MOV.W #00, R5
29
30         JMP     main                       ; jump to current location
31                                           ; (endless loop)
```

III/- PRACTICE

1)- Start simulating the program by clicking the "Download and Debug" button on the toolbar.

- Using single step through the loop one time. This is accomplished by clicking on the "Step Into" button or pressing F11

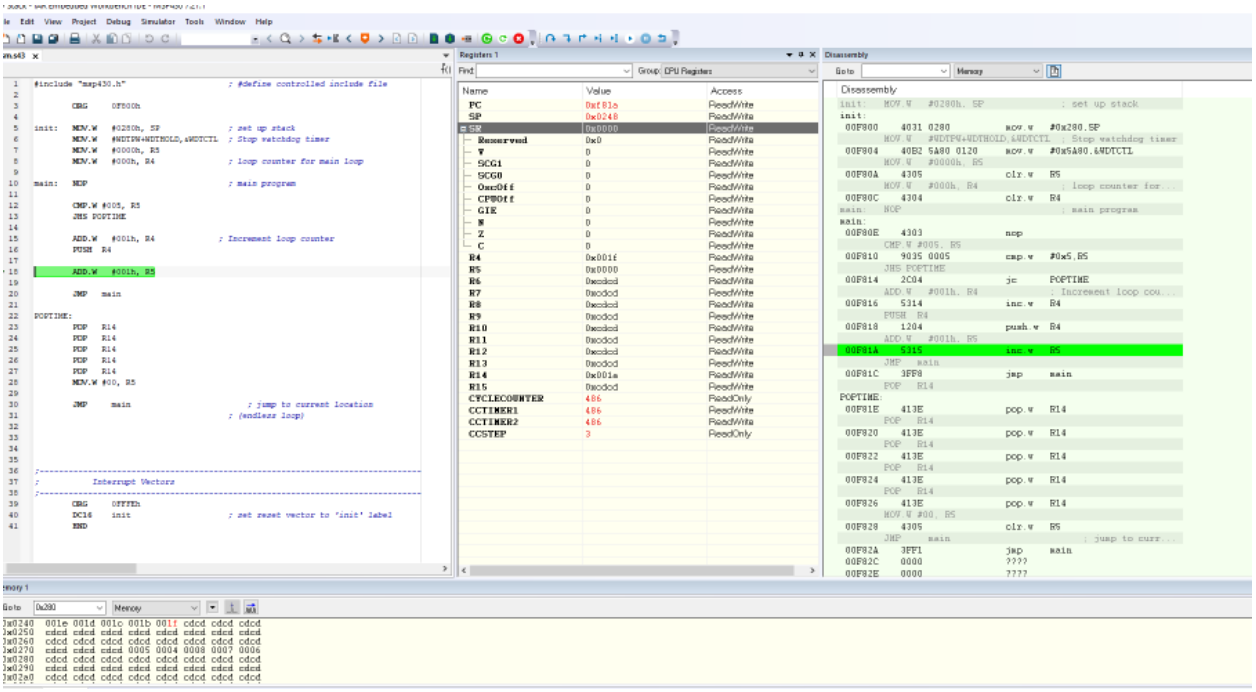
2)- Single step through assembly code in the program and observations Include what happens to the SP and the stack itself in memory with each stack related operation.

3)- Try changing the SP to a value 40 bytes lower using the watch window after executing *line 28* to simulate a new "frame" of operation.

a)- Include what happens to the data in the original stack frame.

b)- Could we use the memory area on the stack between the original stack farm and the new stack frame for temporary variables?

- The Screen of asm.s43 file in Stack operations, and Push and Pop Instruction .



IV/- RESULT

2)- After loop 1 : We see

- The Program Counter (PC) is incremented (+2) after execute each Instruction .
  (from 0xF804 to 0xf81C)

- The Status Register (SR) decreased (-2) from 0x0280 to 0x027E by Push R4 instruction (line 16)
- The Register R4 increment (+1) from 0x0000 to 0x0001 by **ADD.W #001h, R4** instruction (line 15)
- The Register R5 increment (+1) from 0x0000 to 0x0001 by **ADD.W #001h, R5** instruction (line 18)

After loop 5:

- The Register R5 increment (+5) and reached 0x005.
- Thus, CMP.W #005, R5 is the same dst = src (line 12)

=> **JHS POPTIME** Instruction will execute jump to poptime: (line 13)

- The Program Counter (PC) continuous incremented (+2) from 0xF820 to 0xf82A
- The Stack Pointer (SP) is incremented (+2) when run though each POP instruction,
  from 0x0278 to 0x0280.
- The Register R5 = 0x00 by **ADD.W #00h, R5** instruction (line 28)
- The Register R14 in decreased from 0x0005 to 0x0001 (after executed 5 POP instructions)

3)- When changing the SP to a value 40 bytes lower we have value of SP = 0x0240.

And after executing line 28 to simulate a new "frame" of operation, we have :

MOV.W #00, R5 **(line 28)**

a)- What happens to the data in the original stack frame?

b)- Yes, we could use the memory in the region between new stack frame (0x0240) and original stack farm (0x0280) for temporary variables.

Since the SP no longer points to this area, and as long as no other parts of the program attempt to use the stack in this area, we can safely store temporary variables here.

However, doing so could lead to stack management issues and potential bugs if the SP is reset or if any interruptions occur, which could overwrite or disrupt data in this region.

	Loop 1								
Line		Instruction	PC	SP	SR	R4	R5	R14	Note
1	# include "msp430.h"								; # difinene controlled include file
2									
3		ORG 0F800h							
4									
5	init :	MOV.W # 0280h, SP	0xF804	0x0280					; Set up Stack
6		MOV.W # WDPTW+WDTHOLD, &WDTCTL	0xF80A						; Stop Watchdog timer
7		MOV.W # 0000h, R5	0xF80C				0x0000		
8		MOV.W # 0000h, R4	0xF80E			0x0000			; Loop counter for main loop
9									
10	Main :	NOP	0xF810						; Main program
11									
12		CMP.W #005, R5	0xF814		0x0004				; (N = 1)
13		JHS POPTIME	0xF816						
14									
15		ADD.W #001h, R4	0xF818		0x0000	0x0001			; Increment Loop counter
16		PUSH R4	0xF81A	0x027E					
17									
18		ADD.W #001h, R5	0xF81C				0x0001		
19									
20		JMP main							
21	Loop 5	; After each loop then (Add.W #001h, R5) the value of R5 will incremen more +1. So After 5 main loop then R5 = 0x0005 , and JHS POPTIME will execute instruction CMP.W #005, R5 (Jump if higher or the same) and jumo to POPTIME							
22	POPTIME:								
23		POP R14	0xF820	0x0278				0x0005	
24		POP R14	0xF822	0x027A				0x0004	
25		POP R14	0xF824	0x027C				0x0003	
26		POP R14	0xF826	0x027E				0x0002	
27		POP R14	0xF828	0x0280				0x0001	
28		MOV.W #00, R5	0xF82A				0x0000		
29									
30		JMP main					; Jump to current location		
31							; (enless loop)		

V/- CONCLUSION

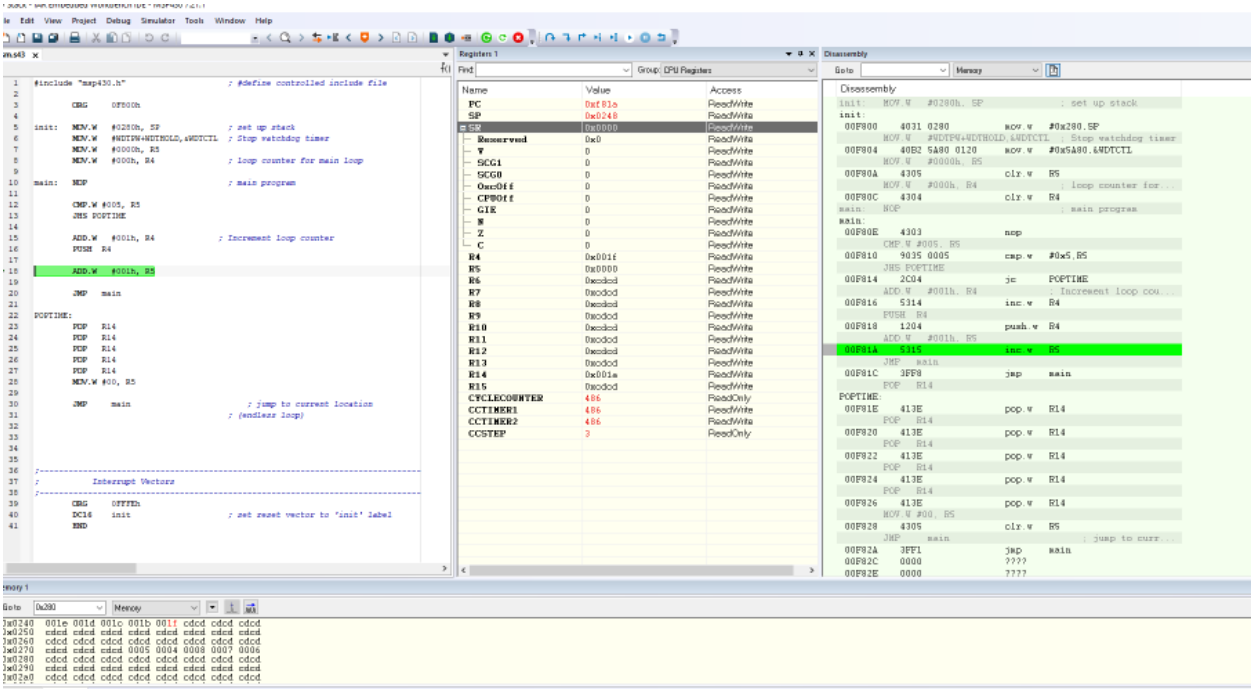
From each step in the program we observe: they work in accordance with the theory of instruction.

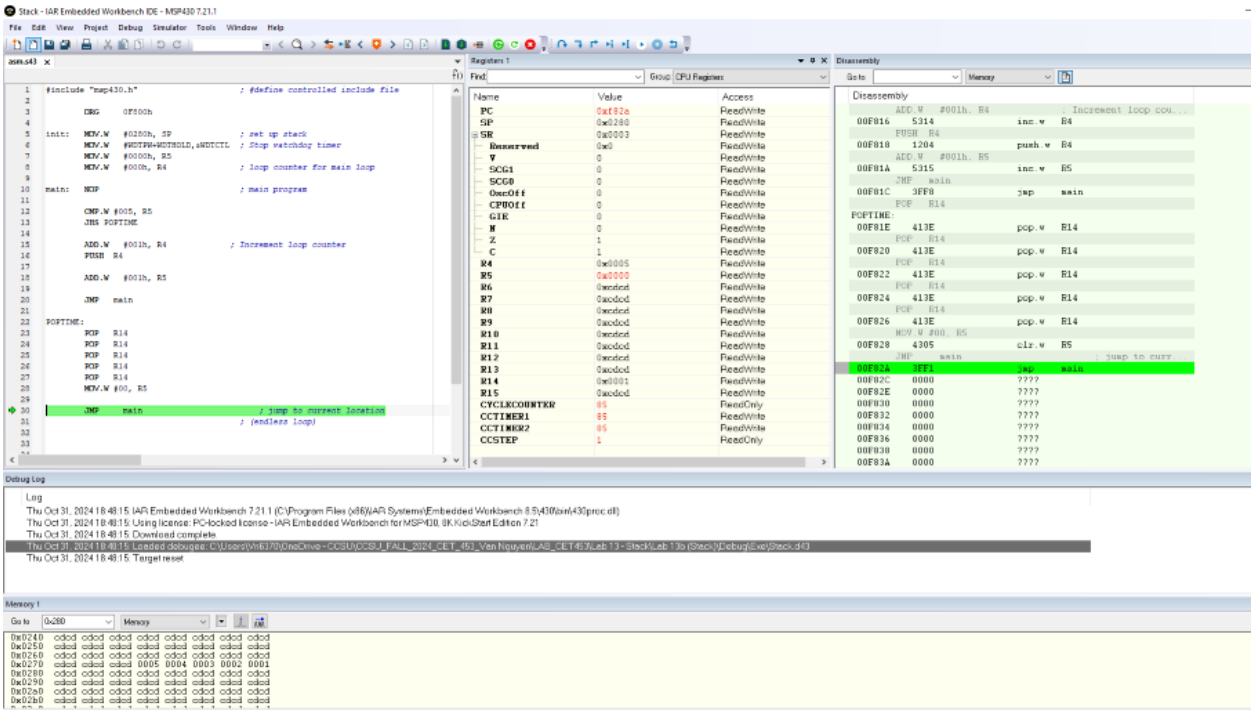
Single step through the program and submit your observations in a lab report.
- Include what happens to the SP and the stack itself in memory with each stack related operation.

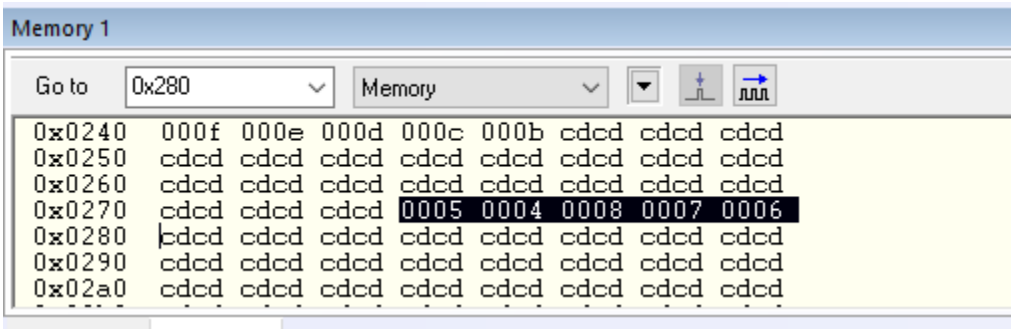
Try changing the SP to a value 40 bytes lower using the watch window after executing line 28 to simulate a new "frame" of operation and submit your observations in a lab report.

- Include what happens to the data in the original stack frame.

- Could you use the memory area on the stack between the original stack farm and the new stack frame for temporary variables?






Memory 1
Go to
0x280
Memory
0x0240 000f 000e 000d 000c 000b cdcd cdcd cdcd
0x0250 cdcd cdcd cdcd cdcd cdcd cdcd cdcd cdcd
0x0260 cdcd cdcd cdcd cdcd cdcd cdcd cdcd cdcd
0x0270 cdcd cdcd cdcd 0005 0004 0008 0007 0006
0x0280 cdcd cdcd cdcd cdcd cdcd cdcd cdcd cdcd
0x0290 cdcd cdcd cdcd cdcd cdcd cdcd cdcd cdcd
0x02a0 cdcd cdcd cdcd cdcd cdcd cdcd cdcd cdcd