Subroutines & Stack

Dr. Farid Farahmand

Updated: 2/18/2019

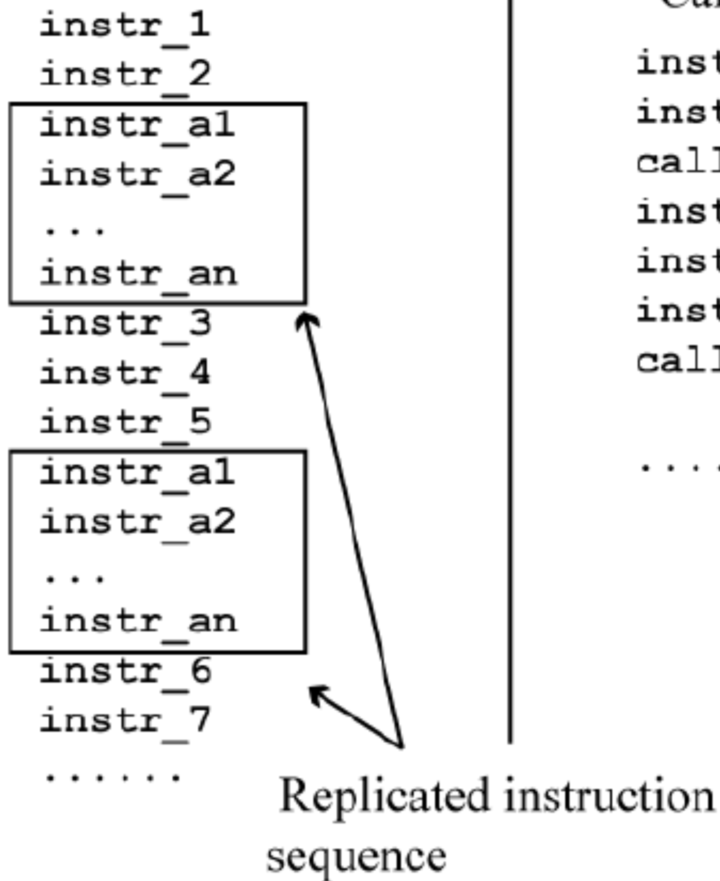
Basic Idea

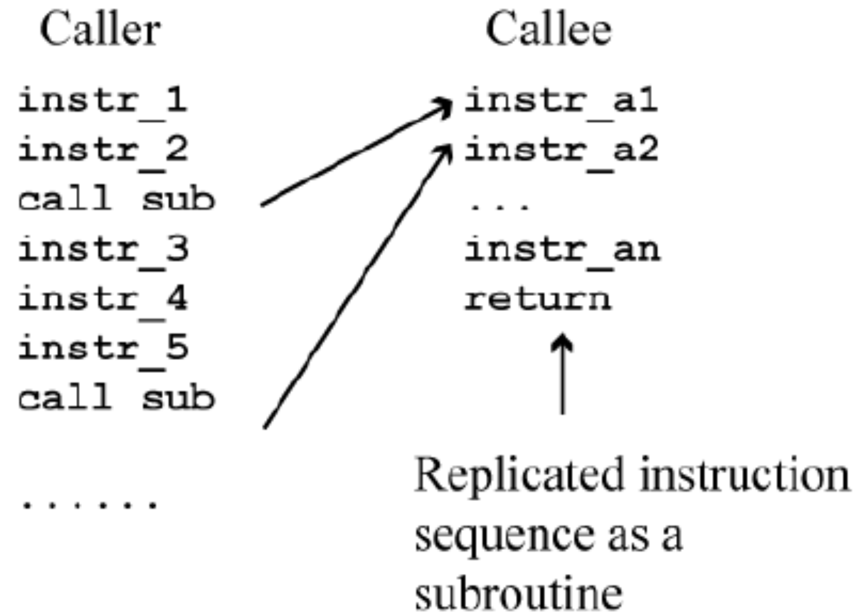
- Large programs are hard to handle
  - We can break them to smaller programs
  - They are called subroutines
- Subroutines are called from the main program
- Writing subroutines
  - When should we jump? (use CALL)
  - Where do we return to? (use RETURN)

Subroutine

- A **subroutine** is a block of code that is **called** from different places from within a main program or other subroutines.
- Saves code space in that the subroutine code does not have to be repeated in the program areas that need it;
  - Only the code for the subroutine call is repeated.
- A subroutine can have
  - **parameters** that control its operation
  - **local variables** for computation.
- A subroutine may pass a **return value** back to the caller.
- Space in data memory must be reserved for parameters, local variables, and the return value.

Subroutine



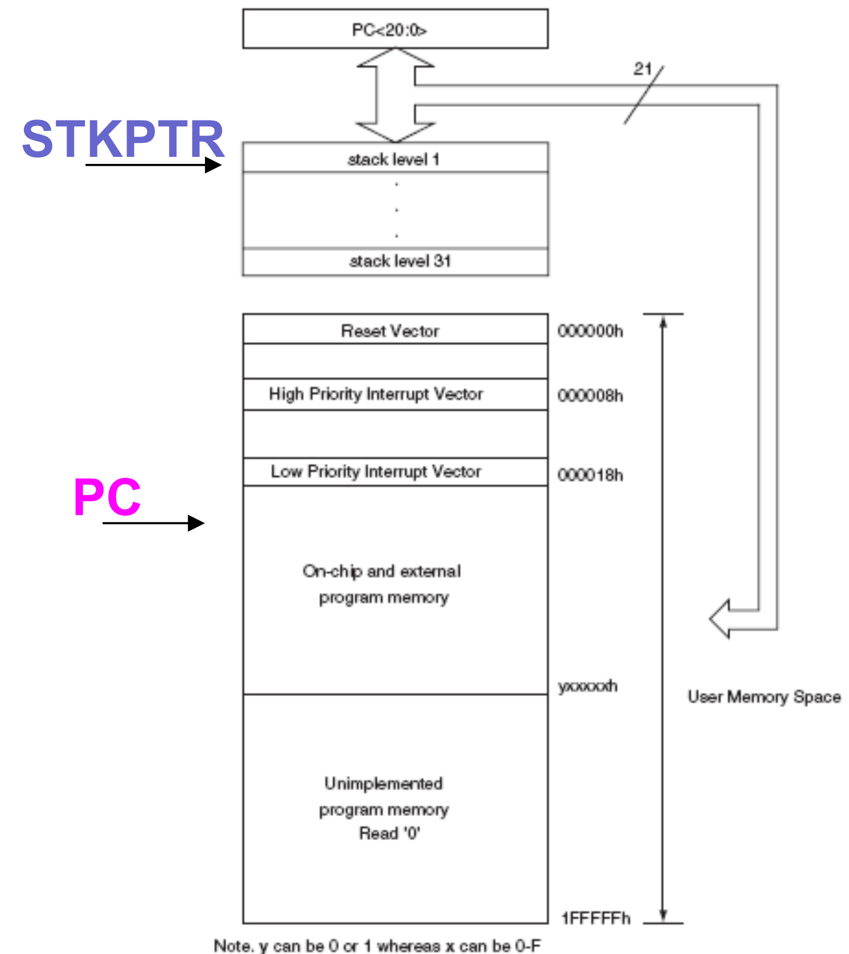


Using Subroutines

- When using subroutines we need to know the following:
  - Where is the NEXT instruction's address
  - How to remember the RETURN address
- Subroutines are based on MPU instructions and use STACK

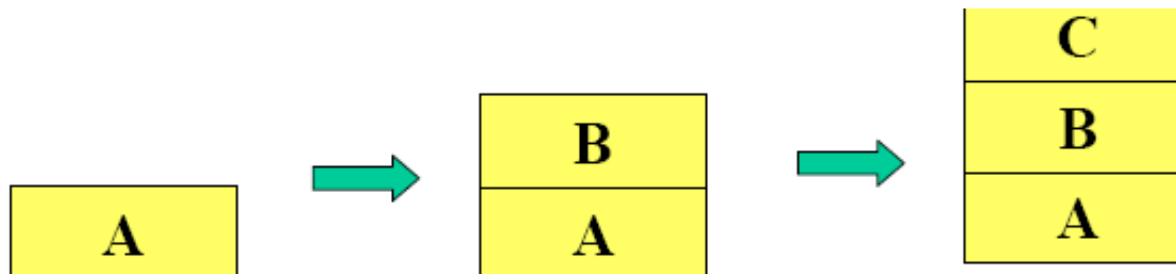
Stack

- Temporary memory storage space used during the execution of a program
- Used by MPU
- Stack Pointer (SP)
  - The MPU uses a register called the stack pointer, similar to the program counter (PC), to keep track of available stack locations.



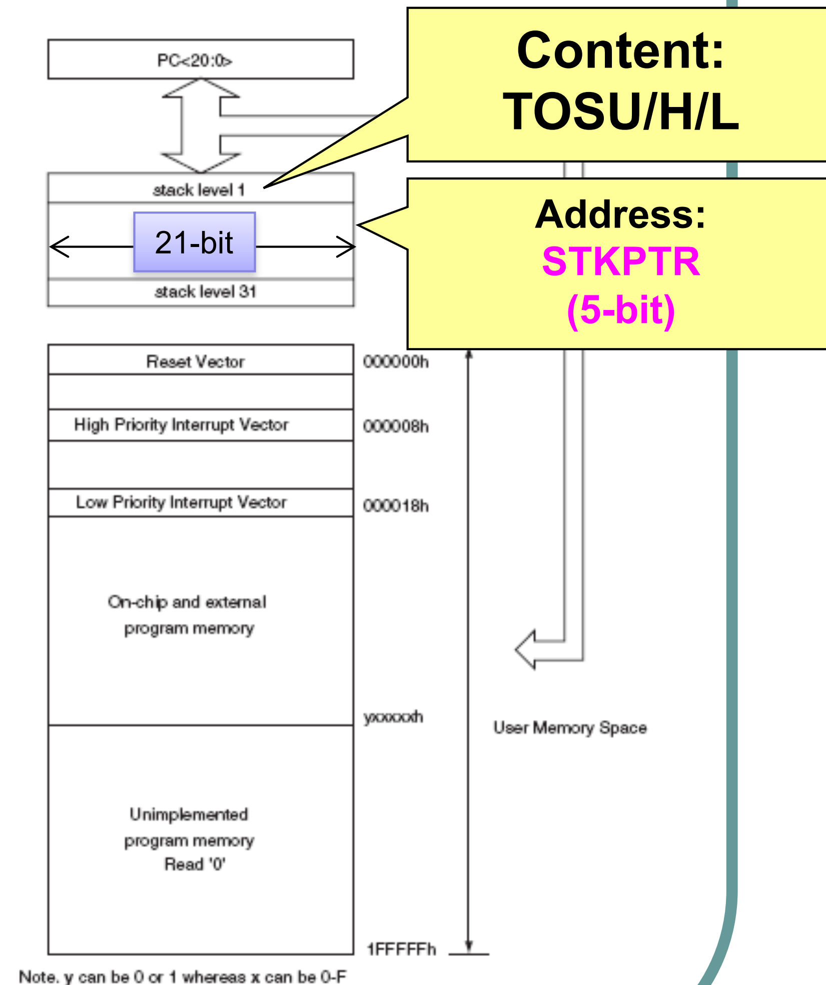
Data Storage via the Stack

- The word ‘stack’ is used because storage/retrieval of words in the stack memory area is the same as accessing items from a stack of items.
- Visualize a stack of boxes. To build a stack, you place box A, then box B, then box C
  - Notice that you only have access to the last item placed on the stack (the Top of Stack –TOS). You retrieve the boxes from the stack in reverse order (C then B then A). A stack is also called a LIFO (last-in-first-out) buffer (similar to a Queue)

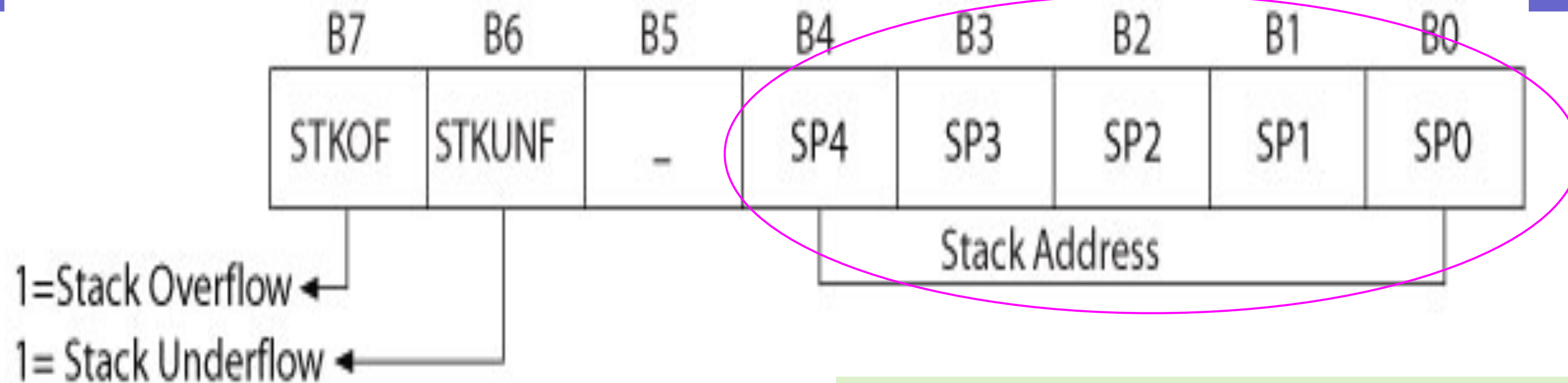


PIC18 Microcontroller Stack

- Consists of 31 registers-21-bit wide, called the hardware stack
  - Starting with 1 to 31
  - Stack is neither a part of program memory or data registers.
  - To identify these 31 registers, 5-bit address is needed
  - PIC18 uses one of the special function registers called STKPTR (Stack Pointer) to keep track of the available stack locations (registers).



STKPTR (Stack Pointer) Register



Find this register in the data sheet:
http://ww1.microchip.com/downloads/en/DeviceDoc/41303D.pdf

- SP4-SP0: Stack Address
- STKOF: Stack overflow
  - When the user attempts to use more than 31 registers to store information (data bytes) on the stack, BIT7 in the STKPTR register is set to indicate an overflow.
- STKUNF: Stack underflow
  - When the user attempts to retrieve more information than what is stored previously on the stack, BIT6 in the STKPTR register is set to indicate an underflow.

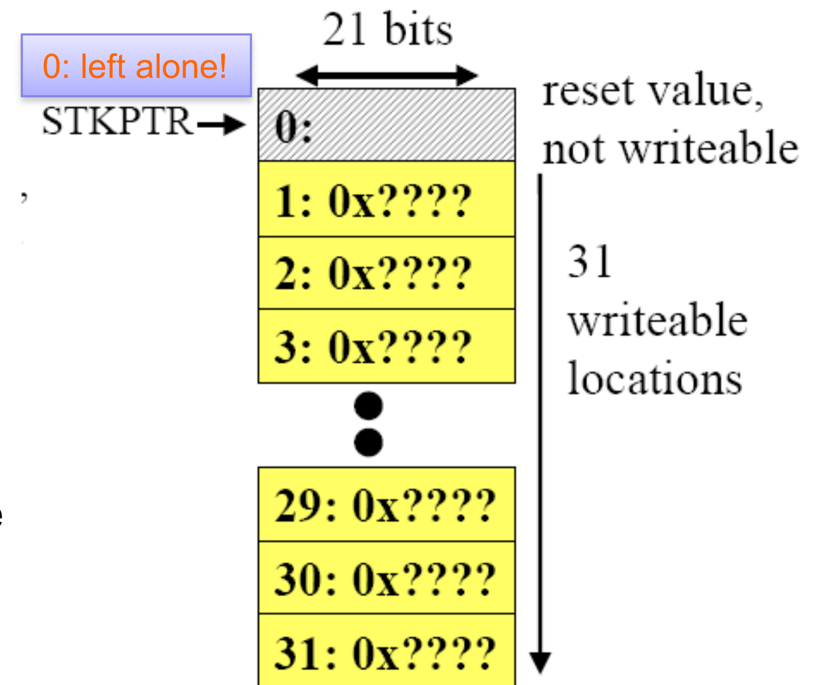
Instructions to Store and Retrieve Information from the Stack

- PUSH
  - Increment the memory address in the stack pointer (by one) and stores the contents of the counter (PC+2) on the top of the stack
- POP
  - Discards the address of the top of the stack and decrement the stack pointer by one
- The contents of the stack (21-bit address), pointed by the stack pointer, are copied into three special function registers
  - TOSU (Top-of-Stack Upper), TOSH (High), and TOSL (Low)

TOSU	TOSH	TOSL

Instructions to Store and Retrieve Information from the Stack

- The PIC18 stack has limited capability compared to other µPs. It resides within its memory, and is limited to 31 locations.
- For a CALL, address of next instruction (nPC) is pushed onto the stack
  - A **push** means to increment STKPTR, then store nPC (Next PC or PC+2) at location [STKPTR].
  - STKPTR++; [STKPTR] ←nPC
- A return instruction pops the PC off the stack.
  - A **pop** means read [STKPTR] and store to the PC, then decrement
  - **STKPTR (PC ←[STKPTR], STKPTR--)**



Example

- What is the value of PC, TOSU/H/L and STKPTR as you execute each line?

nPC	TOS	STKPTR	W	Line	Instruction	Operand
				0001	org	0x20
22	0	0	00	0002	movlw	0x20
24	0	0	20	0003	movwf	0x00
26	26	1	20	0004	push	
28	28	2	20	0005	push	
2A	26	1	20	0006	pop	
2C	0	0	20	0007	pop	

Subroutine Call

- In the PIC18F, the stack is used to store the **return address** of a subroutine call.
- The return address is the place in the calling program that is returned to when subroutine exits.
- On the PIC18Fxx, the return address is PC+4, if PC is the location of the call instruction .
  - Call is a 2-word instruction!
- The return address is PC+2 if it is a rcall instruction.

CALL Instruction

- CALL Label, S (0/1) ;Call subroutine
  ; located at Label
- CALL Label, FAST ;FAST is equivalent to
  ; S = 1
  - If S = 0: Increment the stack pointer and store the contents of the program counter (PC+4) on the top of the stack (TOS) and branch to the subroutine address located at Label.
  - If S = 1: Increment the stack pointer and store the contents of the program counter (PC+4) on the top of the stack (TOS) and the contents of W, STATUS, and BSR registers in their respective **shadow registers**, and branch to the subroutine address located at Label.

RCALL Instruction

- RCALL, n ;Relative call to subroutine

within n = ± 512 ;words (or ± 1 Kbyte)
;Increments the stack pointer and stores the contents of the program counter (PC+2) on the top of the stack (TOS) and branch to the location Label within n = ± 512 words (or ± 1 ;Kbyte)

RETURN Instruction

- RETURN,0 → gets the address from TOS and moves it to PC, decrements stack pointer
- RETURN,1 → gets the address from TOS and moves it to PC, decrements stack pointer; retrieves all shadow registers (WREG, STATUS, BSR)*
- RETLW → gets the address from TOS and moves it to PC ; returns literal to WREG, decrements stack pointer

* 1 or FAST

Example

- Program Listing with Memory Addresses

```
Main Program                    Org 0x20

0020 0EFE        START: MOVLW
0022 6E94               MOVWF
0024 6E01               MOVWF
0026 C001 FF82   ONOFF: MOVFF
002A EC20 F000          CALL
002E 1E01               COMF
0030 D7FA               BRA
                        END  ;endi
```

?

```
Subroutine

DELAY50MC:
0040    0EA6    MOVLW
0042    6E10    MOVWF
0044    0610    DECF
0046    E1FE    BNZ
0048    0012    RETURN
```

?

Can you tell what the complete commands?
Why do we have 0x2A and then 0x2E?
How many instruction cycle is a CALL?
How did we start DELAY subroutine at 0x40?
What happens after executing 0x0048?

Example

- Program Listing with Memory Addresses

Main Program

Org 0x20

```
0020 0EFE    START: MOVLW   B'11111110'
0022 6E94           MOVWF   TRISC
0024 6E01           MOVWF   REG1
0026 C001 FF82 ONOFF: MOVFF REG1,PORTC
002A EC20 F000      CALL    DELAY50MC
002E 1E01           COMF    REG1,1
0030 D7FA           BRA     ONOFF
```

2-Word Instructions

END ;ending directive

Subroutine

Org 0x40

```
DELAY50MC:
0040   0EA6   MOVLW   D'166'
0042   6E10   MOVWF   REG10
0044   0610   DECF    REG10,1
0046   E1FE   BNZ     LOOP1
0048   0012   RETURN
```

Note:
2-Word
→Inst.
PC+4 (2A→2E)

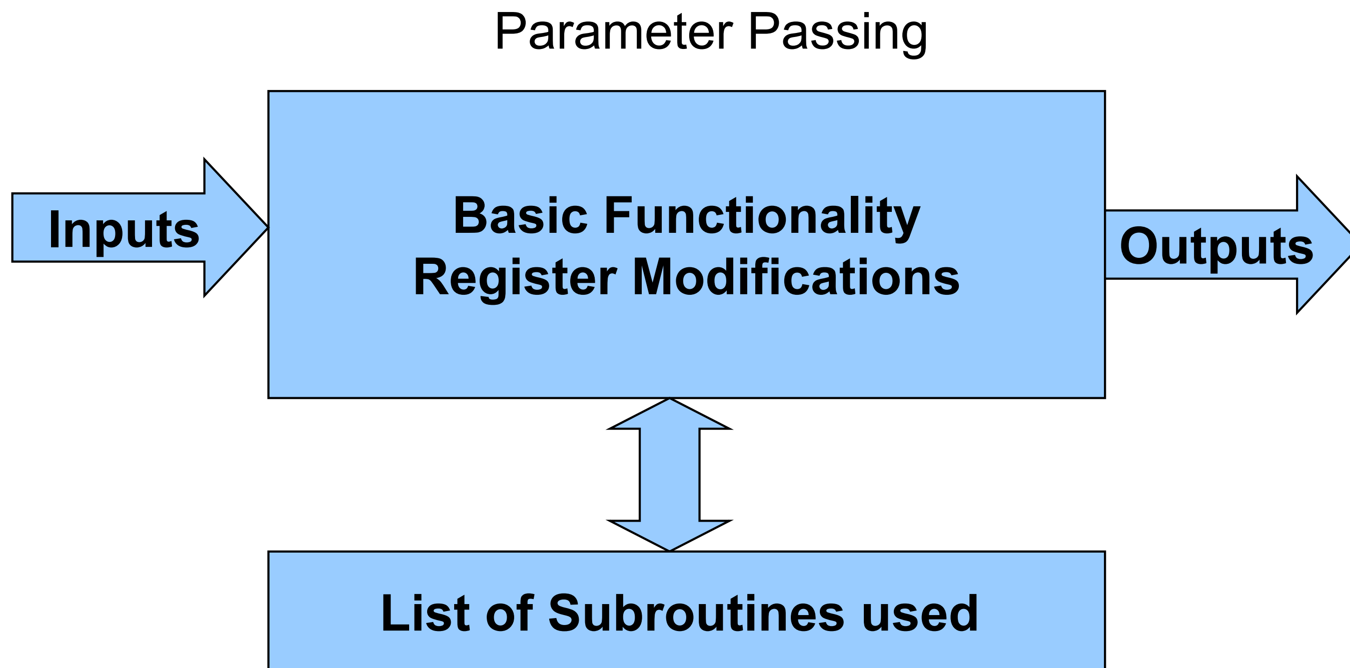
After CALL:
TOS=00 00 2E
PC=00 00 40
STKPTR=01

After RETURN:

PC=2E
STKPTR=00

Subroutine Architecture

How do we write a subroutine?



Macros and Software Stack

- Macro
  - Group of assembly language instructions that can be labeled with name
  - Short cut provided by assembler
  - Format includes three parts

```
Push_macro      macro      arg
                movff      arg,POSTINC1
                endm
```

USE: Push_macro WREG

A **push** means to increment STKPTR, then store nPC (Next PC or PC+2) at location [STKPTR].

STKPTR++; [STKPTR] ←nPC

Macro Description - Example

- See how FSR is loaded and POSTDEC works.
- How a MACRO is being called!

BYTE MACRO: Before MAIN

```
        GOTO    MAIN

BYTE    MACRO   REGXX
        LFSR    FSR1,REGXX
        MOVFF   POSTDEC1,REG0
        MOVFF   INDF1,REG0
        ENDM

        ORG     0x020           ;Begin assembly at 0020
MAIN:
        MOVLW   0xAA
        MOVWF   REG10
        BYTE    0x10

        SLEEP                   ;Power down
        END
```

Add...	Symbol Name	Value
FE8	WREG	0xAA
FD8	STATUS	0x00
000	REG0	0xBB
010	REG10	0xAA
	INDF1	Restricted Memory
	POSTDEC1	Restricted Memory

ldress	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
000	BB	00	00	00	00	00	00	00	00	00	00	00	00	00	00	BB
010	AA	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

MACRO Application

- Note COUNT is not defined in the MARCO
  - It is the "arg" of the MACRO
- MACRO is assembled after every instance it is called

```
REG0    EQU     0x00
REG1    EQU     0x01
REG2    EQU     0x02

        GOTO    MAIN

BYTE    MACRO   COUNT
        MOVLW   COUNT
        ADDLW   2
        ENDM

        ORG     0x020
MAIN:
        MOVLW   0x12
        BYTE    05

        SLEEP
        END
```

Add SFR | ADCON0 | Add Symbol | _BOR_OFF_2L

Update	Address	Symbol Name	Value
	FE8	WREG	0x07

Watch 1 | Watch 2 | Watch 3 | Watch 4

Trace

Line	Addr	Op	Label	Instruction	SA	SD	DA	DD
-5	0000	EF10		GOTO 0x20	----	--	----	--
-4	0002	F000		NOP	----	--	----	--
-3	0020	0E12	MAIN	MOVLW 0x12	W	--	W	12
-2	0022	0E05		MOVLW 0x5	W	--	W	05
-1	0024	0F02		ADDLW 0x2	W	--	W	07
0	0026	0003		SLEEP	----	--	----	--

MACRO Application

- So what if MACRO is called multiple times?
  - A MACRO is assembled after every instance it is called

```
MAIN:
        MOVLW   0x12
        BYTE    05
        MOVLW   0x12
        BYTE    05
        SLEEP
        END
```

-8	0000	EF10		GOTO 0x20
-7	0002	F000		NOP
-6	0020	0E12	MAIN	MOVLW 0x12
-5	0022	0E05		MOVLW 0x5
-4	0024	0F02		ADDLW 0x2
-3	0026	0E12		MOVLW 0x12
-2	0028	0E05		MOVLW 0x5
-1	002A	0F02		ADDLW 0x2
0	002C	0003		SLEEP

Subroutine versus Macro

- Subroutine (by MPU)
  - Requires instructions such as CALL and RETURN, and the STACK (overhead)
  - Memory space required by a subroutine does not depend on how many times it is called
  - It is less efficient in terms of execution than that of a macro because it includes overhead instructions such as Call and Return
- Macro (by assembler)
  - Based on assembler
  - Shortcut in writing assembly code
  - Memory space required depends on how many times it is called
  - In terms of execution it is more efficient because it does not have overhead instructions

More about subroutines…

- Remember subroutines can call other subroutines
- This is referred as structured code

Examine this code:

```
;//** I N I T I A L I Z A T I O N ***********/

;//** Constants

BYTECOPY     EQU      0X70
BLOCKNUM     EQU      0x05

ORG 0x60
BUFFER  DB 0X01, 0X02, 0x01, 0x3, 0x00, 0x00

;//** M I C R O C O D E  ***********/

GOTO     MAIN

BYTECP  MACRO   STARTHERE
        LFSR    FSR1, STARTHERE ; SET THE POINTER
        MOVLW   UPPER   BUFFER
        MOVWF   TBLPTRU
        MOVLW   HIGH   BUFFER
        MOVWF   TBLPTRH
        MOVLW   LOW   BUFFER
        MOVWF   TBLPTRL
NEXT_BYTE
        TBLRD*+
        MOVLW   0x0
        XORWF   TABLAT
        BZ      ENDLOOP

        MOVFF   TABLAT,POSTINC1
        GOTO    NEXT_BYTE
ENDLOOP
        ENDM
```

```
;/// Clearing a block of registers
CLEARME    MACRO    STARTCLEAR
           LFSR     FSR1, STARTCLEAR
           MOVLW    0x0
NEXT_CLEAR
           MOVFF    WREG,POSTINC1
           DECF     BLOCKNUM
           BZ       ENDLOOP_CLEAR
          GOTO      NEXT_CLEAR
ENDLOOP_CLEAR
          ENDM

;//** M A I N   C O D E  ***********/
org 0x80
MAIN
    MOVLW    0x0
    MOVLW    0x0
    CLEARME 0x60
    BYTECP BYTECOPY
    MOVLW    0x0
END
```

See next slide…..

Answer the following:

- At what location in the program memory CLEARME is built? Explain.
- What are the contents of register 0x60, 0x61, etc. in the program memory?
- Where is the location of FSR1 when the MARCO is called?
- Where exactly does CLEARME macro does? How many registers are effected?
- Where exactly does BYTECP macro does? How many registers are effected?
- Modify the program using MACROs such that you perform the following tasks:
  - Copy NINE values already stored in PROGRAM MEMORY locations, starting with locations 0x80, into RAM locations starting with register 0x80. Assume the numbers are 1-9.
  - Copy NINE values already stored in PROGRAM MEMORY locations, starting with locations 0x80, into RAM locations starting with register 0x90. Assume the numbers are 1-9.
  - Take the sum of all the values and locate the SUM in RAM location 0x100.
  - Delete all the RAM registers starting with with register 0x90 - 0x09F

Example…..

- Modify program listing here such that you can correctly take the average of any sum.
  - http://web.sonoma.edu/users/f/farahman/sonoma/courses/es310/labs/ip6-5_finding_average_temp.asm
- The number of inputs can be up to 20 non-zero unsigned values