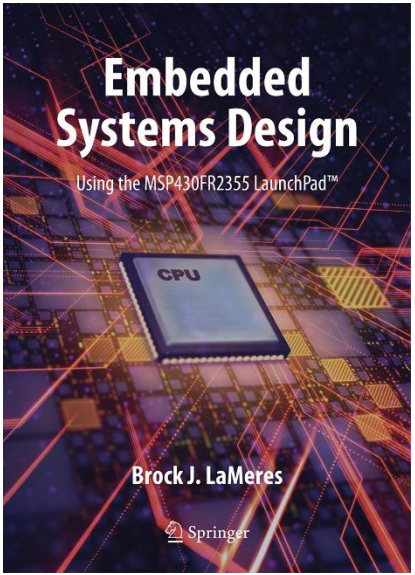
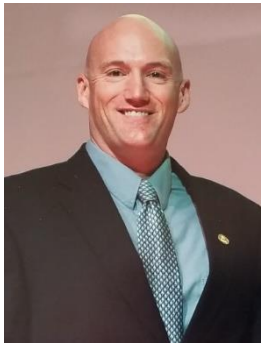
CHAPTER 10: THE STACK AND SUBROUTINES

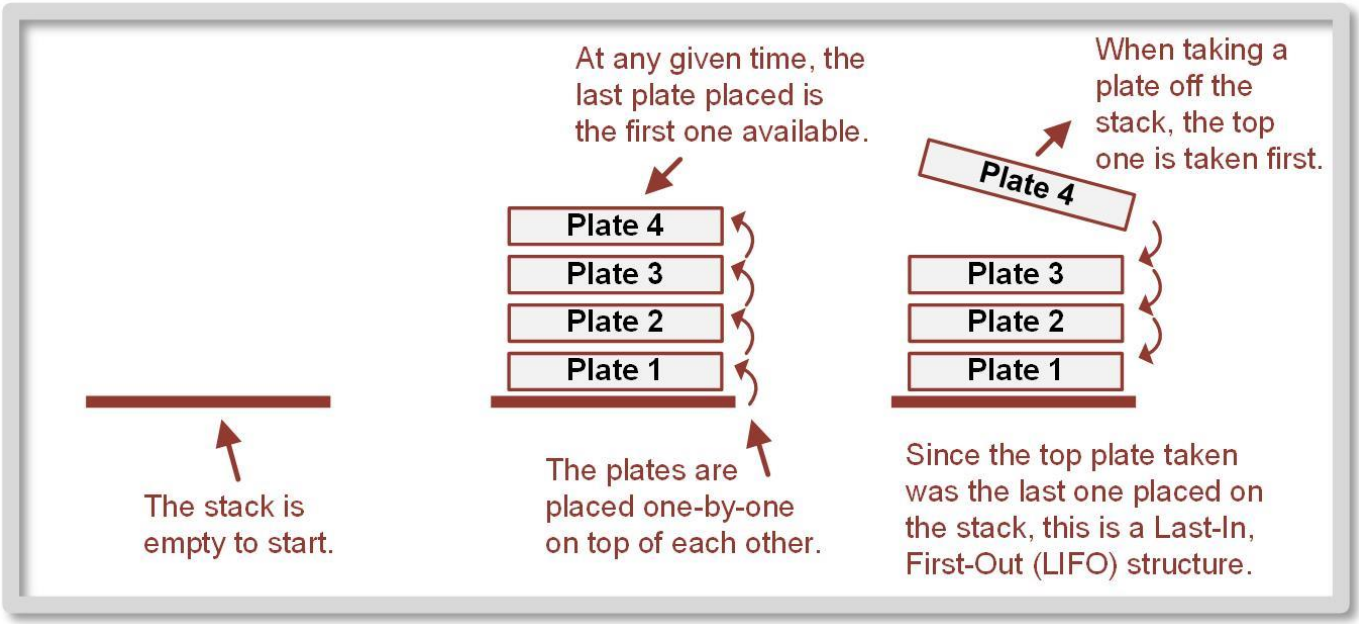
10.1 THE STACK





10.1 THE STACK

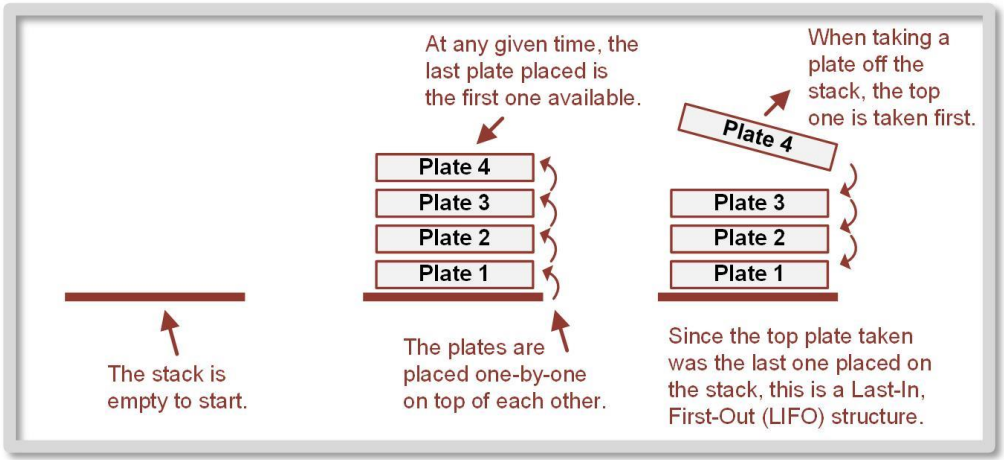
- A stack is a **last-in, first out** (LIFO) storage structure.



10.1 THE STACK

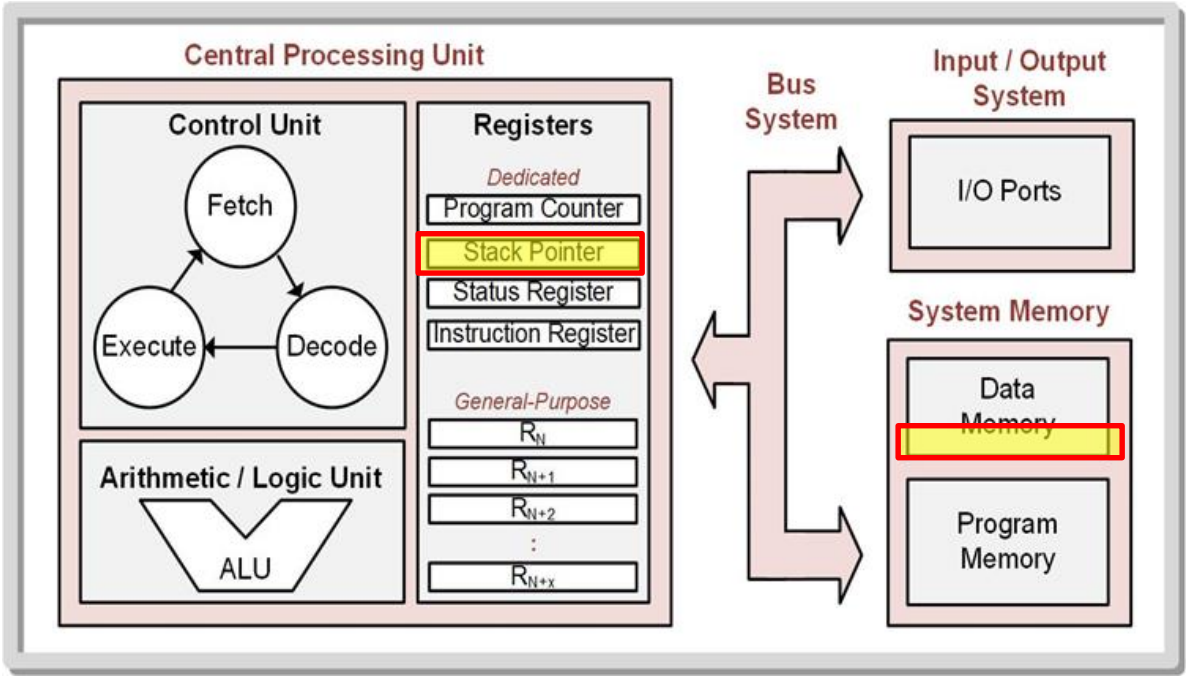
- **Stack** – a system that allows us to dynamically allocate data memory.

- **Dynamically** – we can access memory without initializing it or reserving it using assembler directives such as .short and .space.



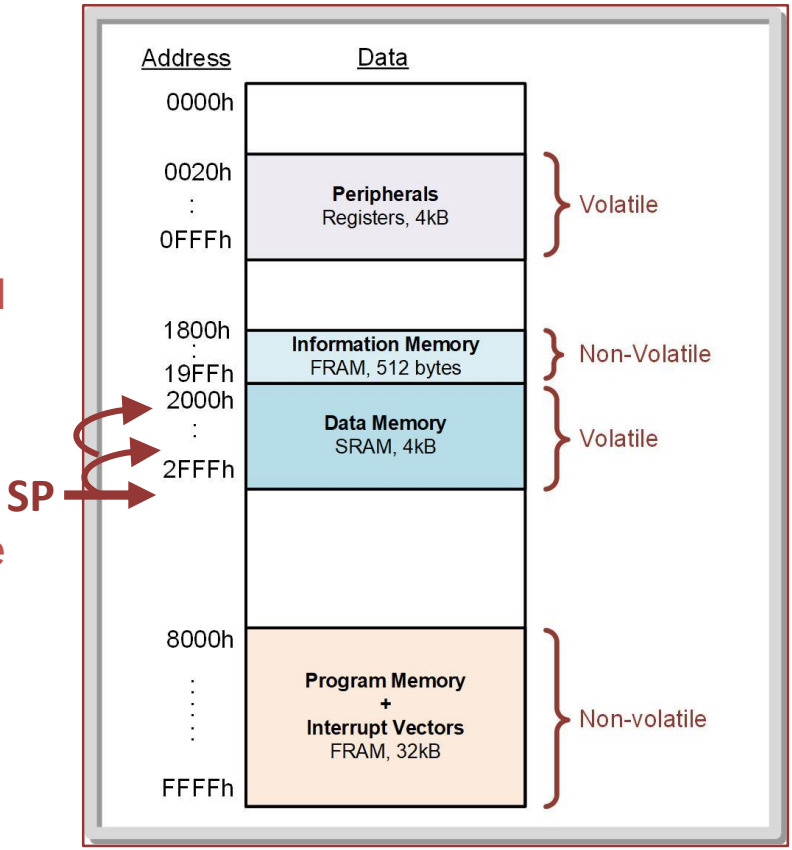
10.1 THE STACK

- **What Physically is the Stack in an MCU?**
  - Storage at the end of data memory & an address pointer.



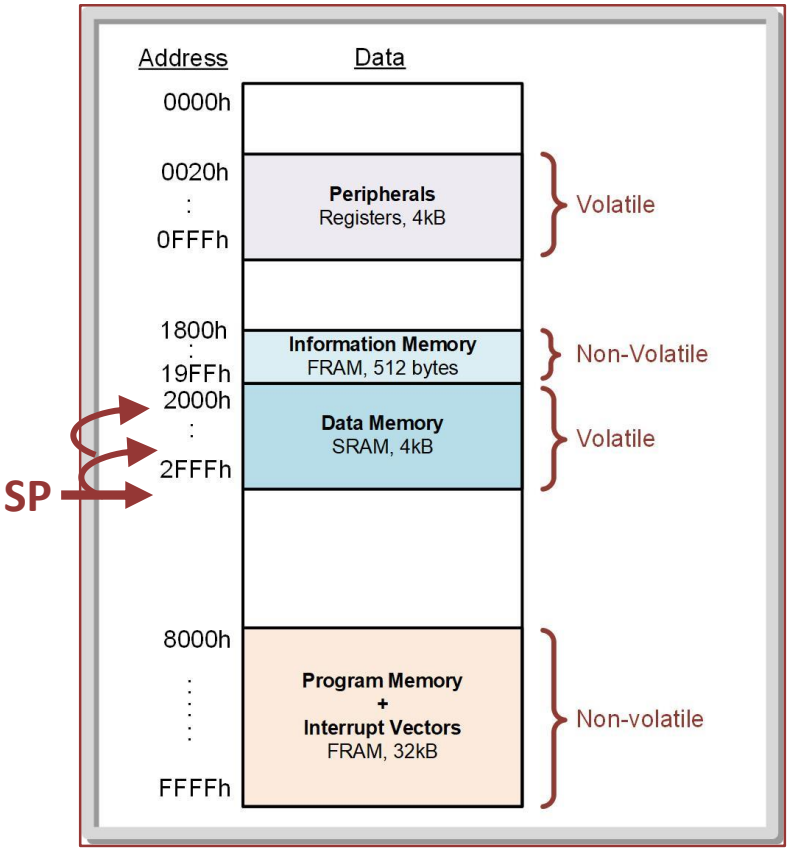
10.1 THE STACK

- The data memory range in the MSP430FR2355 is from 2000h → 2FFFh.
- The stack resides at the end of data memory to allow the maximum potential size of the stack and also avoids overriding reserved locations in memory that are placed at the beginning address of data memory (i.e., 2000h).



10.1 THE STACK

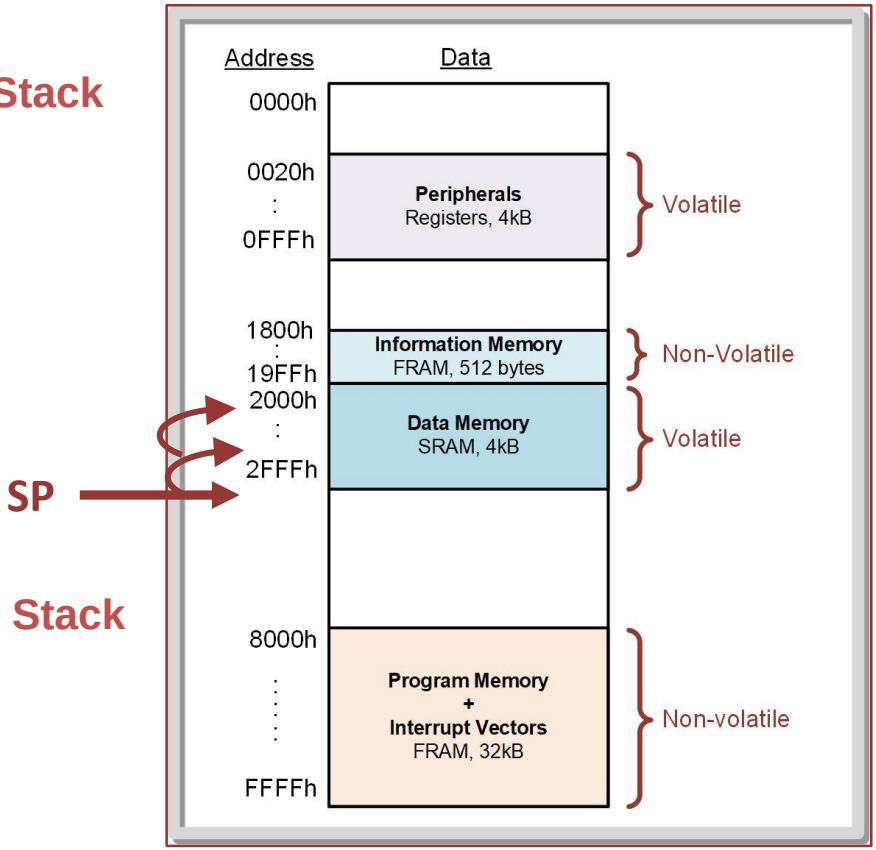
- SP is initialized to 3000h.
- This means that the first 16-bit word of information pushed will be stored at address 2FFEh.
- This is accomplished using a move instruction and a global constant called __STACK_END.



10.1 THE STACK

- **Push = Put Data on Stack**
  - decrement SP
  - dst → @SP

- **Pop = Get Data from Stack**
  - increment SP
  - mov src, @SP



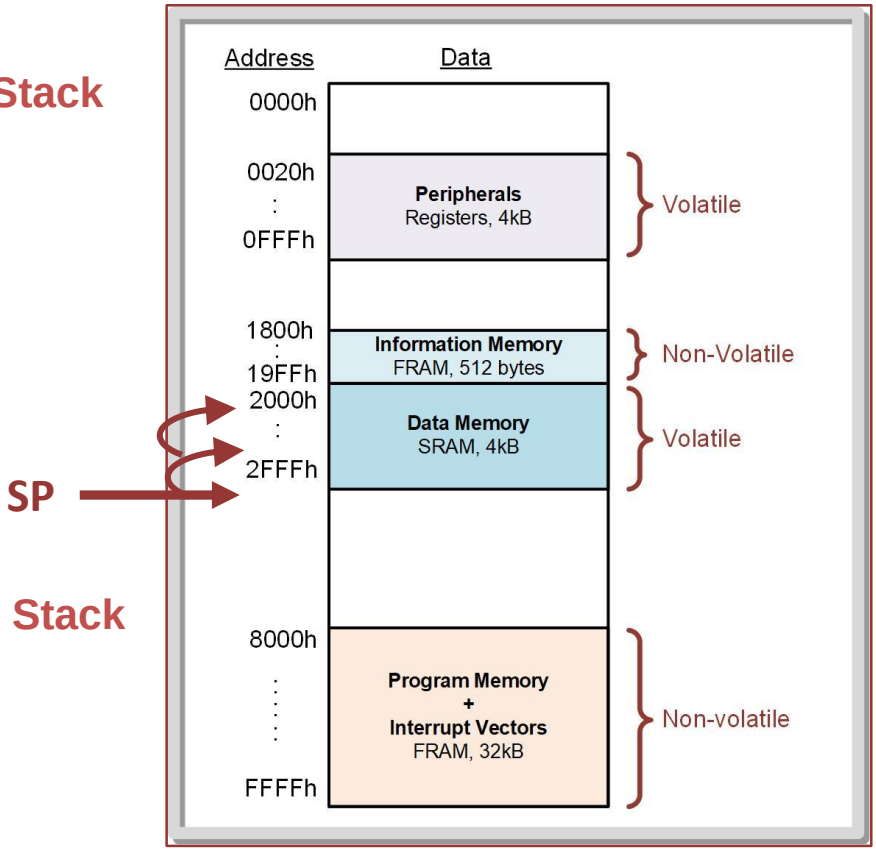
10.1 THE STACK

- **Push = Put Data on Stack**
  - decrement SP
  - dst → @SP

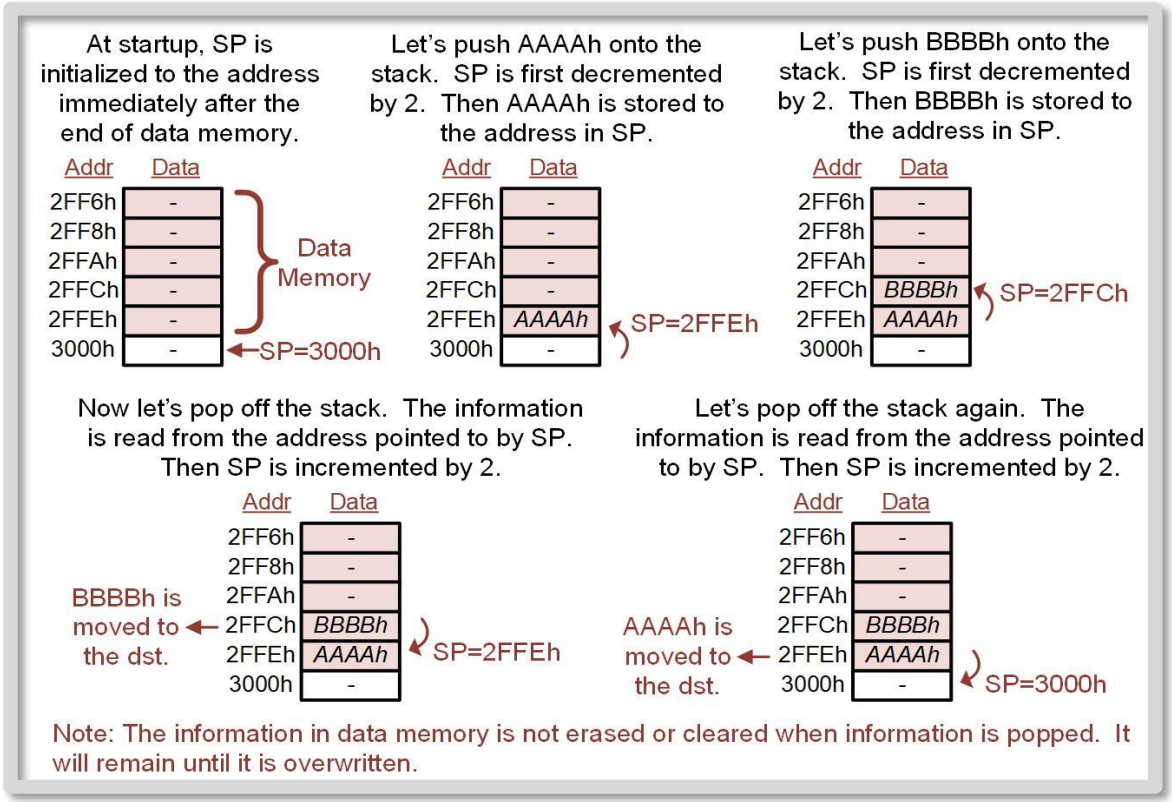
  - .w: SP – 2 → SP

- **Pop = Get Data from Stack**
  - increment SP
  - mov src, @SP

  - .w: SP + 2 → SP



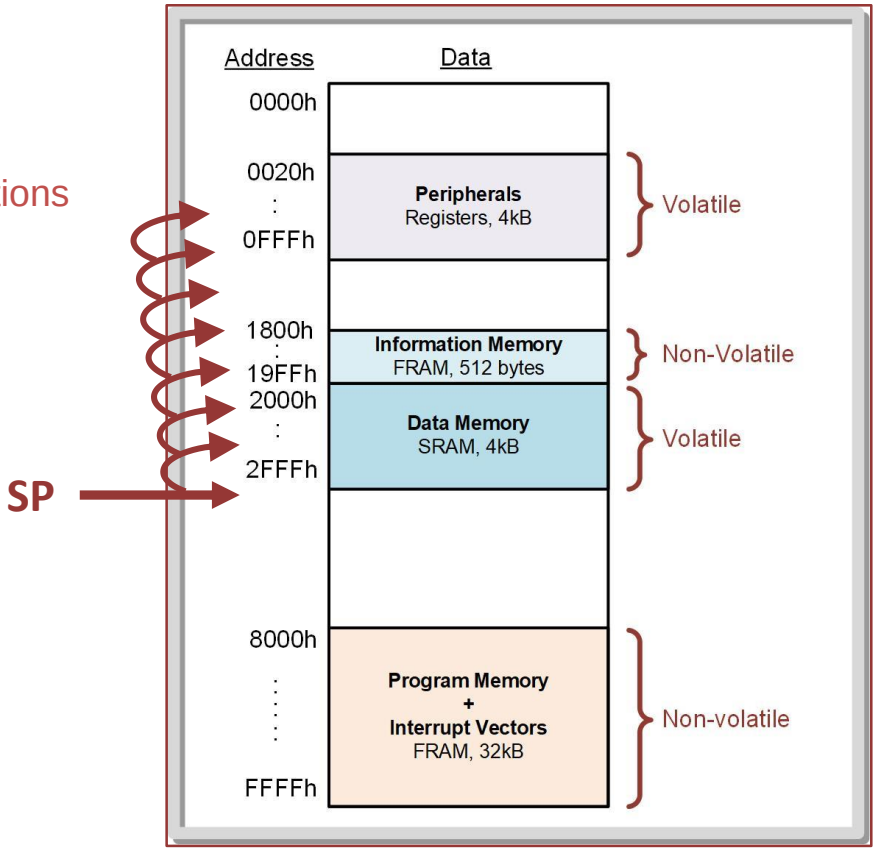
10.1 THE STACK



Note: The information in data memory is not erased or cleared when information is popped. It will remain until it is overwritten.

10.1 THE STACK

- **STACK Overflow**
  - When pushes start overwriting other locations in memory.



EXAMPLE: USING THE STACK

```
main:
    mov.w   #0AAAAh, R4
    mov.w   #0BBBBh, R5

    push    R4
    push    R5

    pop     R6
    pop     R7


    mov.w   #0CCCCh, R4
    mov.w   #0DDDDh, R5

    push    R4
    push    R5

    pop     R6
    pop     R7

    jmp     main
```

First we'll put values into R4 and R5.

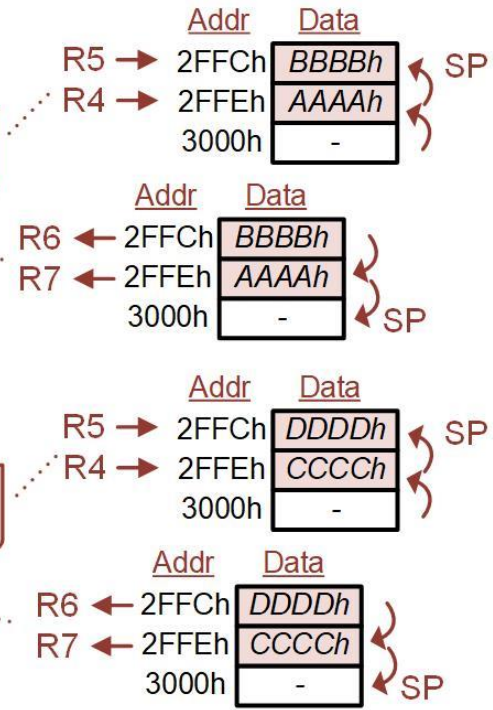
Then we'll push them onto the stack.

Then we'll pop them into R6 and R7.

Next we'll put new values into R4 and R5.

Then we'll push them onto the stack.

Then we'll pop them into R6 and R7.



EXAMPLE: USING THE STACK

Step 1: Create a new Empty Assembly-only CCS project titled:

Asm_Stack

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

EXAMPLE: USING THE STACK

Step 3: Debug your program.

Step 4: Run your program to the breakpoint.

Step 5: Open the Register Viewer so that you can see SP and R4 → R7. Open the Memory Browser and go to 0x3000. Then scroll up so you can see the values 2FFEh and 2FFCh.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

EXAMPLE: USING THE STACK

Step 6: Step your program. As you step, look at the values of SP and the values in data memory before 0x3000. In the Memory Browser you should see the following as the values are pushed.

First two pushes.

Second two pushes.

Did it work? Did you see SP decrement from 3000h → 2FFEh → 2FFECh as values were pushed?

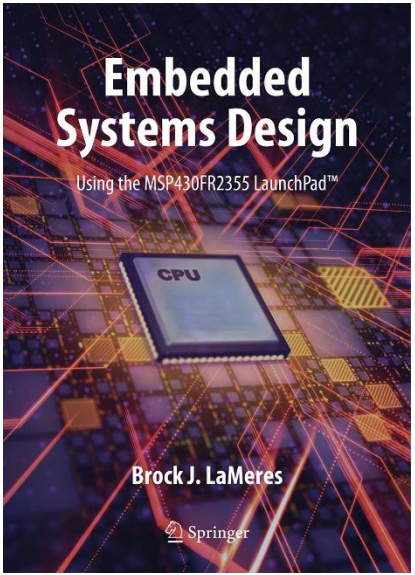
Did you see SP increment from 2FFCh → 2FFEh → 3000h as values were popped?

Did you see the values in data memory at addresses 2FFCh and 2FFEh change as values were pushed?

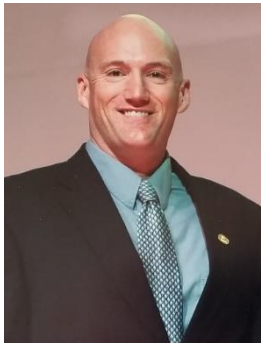
Did you see the values of R6 and R7 change as values were popped?

CHAPTER 10: THE STACK AND SUBROUTINES

10.1 THE STACK

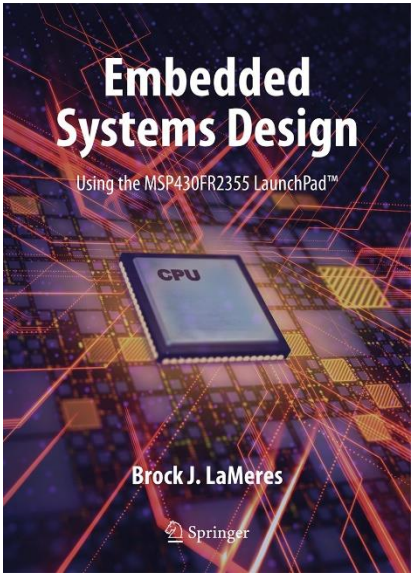


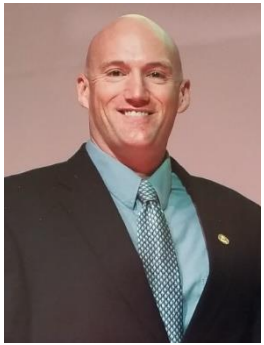
www.youtube.com/c/DigitalLogicProgramming_LaMeres



CHAPTER 10: THE STACK AND SUBROUTINES

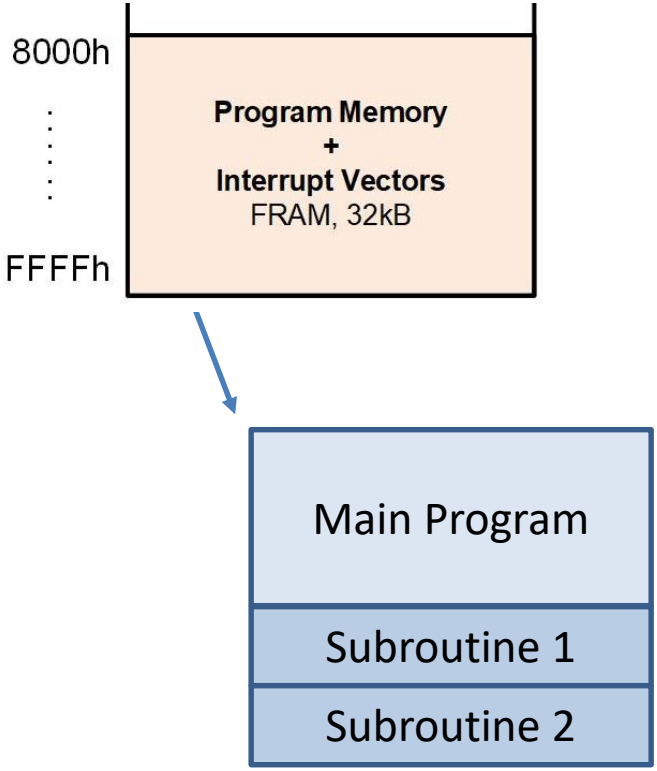
10.2 SUBROUTINES





10.2 SUBROUTINES

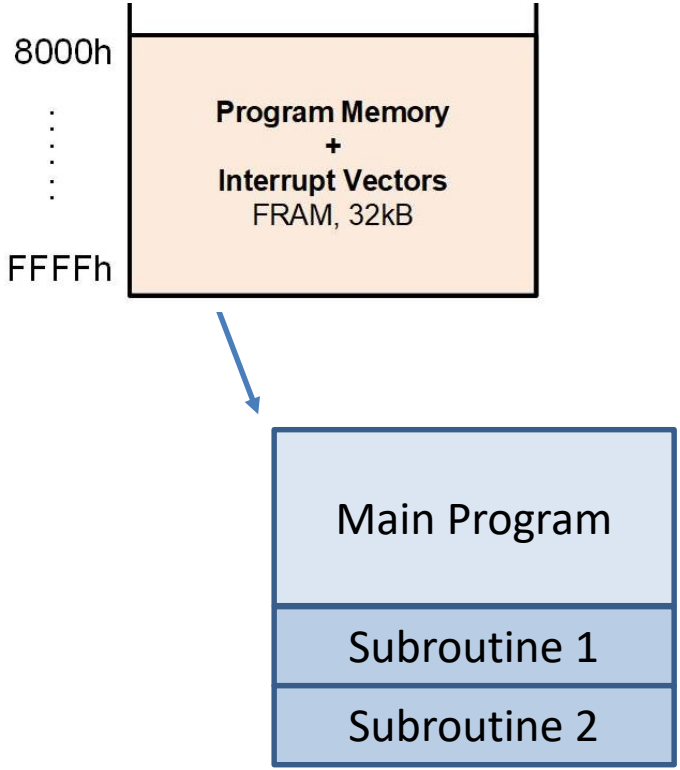
- **Subroutine** – a piece of code that will be used repeatedly in a program; typically accomplishes a very specific task.
- The subroutine code is implemented only once outside the main program loop. This creates a more efficient and simple program to read.
- Other names for subroutines: *procedures, functions, routines, methods, subprograms.*



10.2 SUBROUTINES

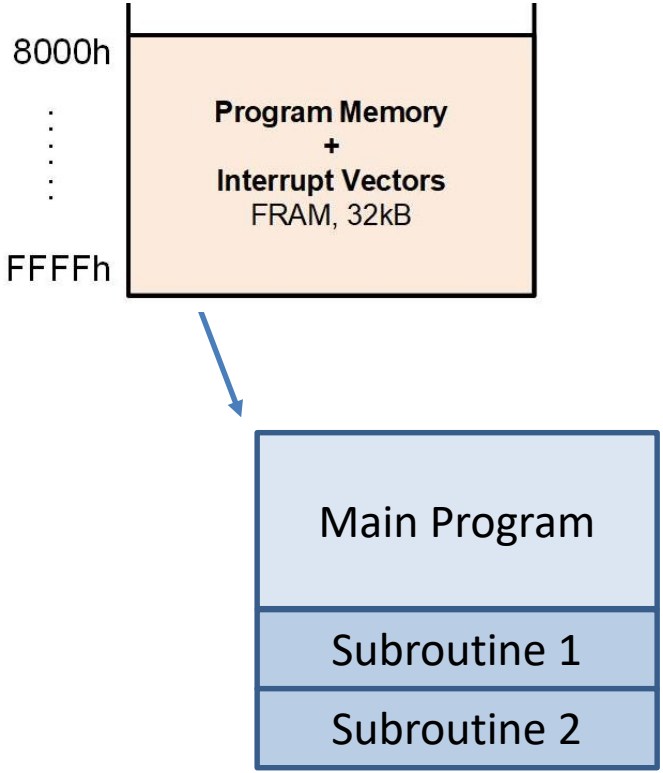
- Whenever the subroutine is needed, it can be executed by jumping to it.

- Once the subroutine completes, a return jump is used to move the PC back to the next location in the main program loop to continue operation.



10.2 SUBROUTINES

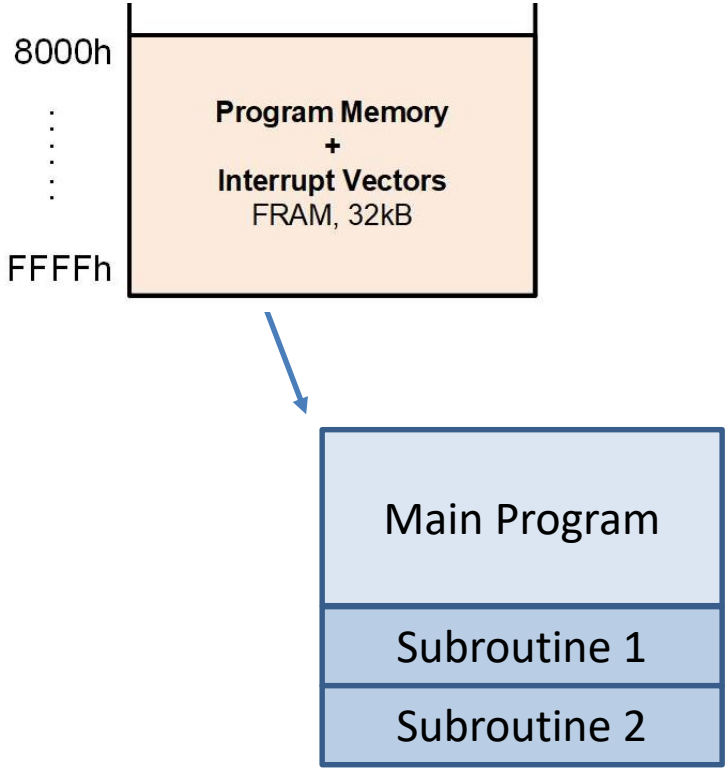
- A subroutine starts with an address label to mark its location in memory.
- Additional steps must be taken when jumping to a subroutine because while the starting address of the subroutine is **always the same**, the return address in the main program **will vary** depending on where in the main program it is called.



10.2 SUBROUTINES

- ***Call*** – instruction that is used to jump to the subroutine address label and handles storing the return address on the stack prior to jumping to the subroutine address.

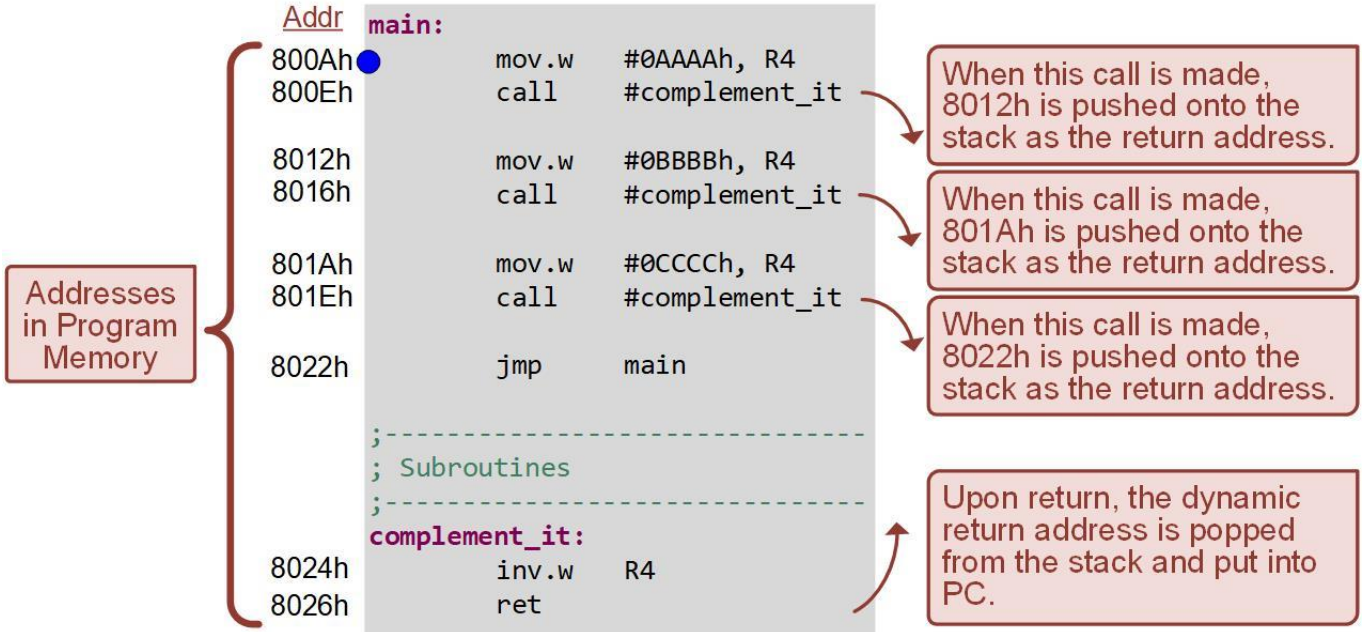
- ***Ret*** – instruction used at the end of the subroutine that pops the return address off the stack and places it into PC to return to the main program.



10.2 SUBROUTINES

- Variables can be passed to subroutines using three different approaches:
  - Using the CPU registers
  - Using the stack
  - Using dedicated variables in data memory

EXAMPLE: USING SUBROUTINES



EXAMPLE: USING SUBROUTINES

Step 1: Create a new Empty Assembly-only CCS project titled:

Asm_Subroutines

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

EXAMPLE: USING SUBROUTINES

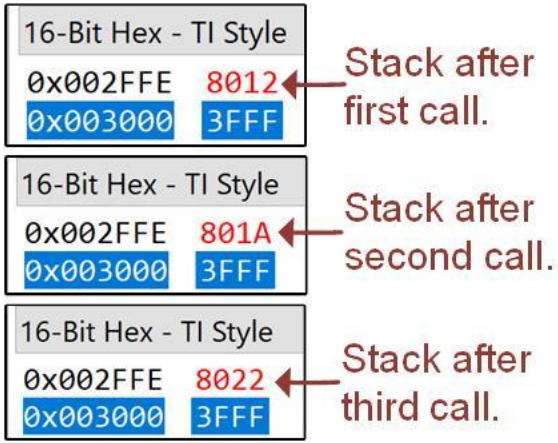
Step 3: Debug your program.

Step 4: Run your program to the breakpoint.

Step 5: Open the Register Viewer so that you can see PC, SP, and R4. Open the Memory Browser and go to 0x3000. Then scroll up so you can see the first location on the stack (address 2FFEh).

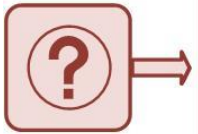
EXAMPLE: USING SUBROUTINES

Step 6: Step your program using **Step Into**. As you step, look at the values of PC, SP, and the values in data memory for the stack. In the Memory Browser you should see the following as the values are pushed.



Step 7: Now step your program using **Step Over**. This time when you step you'll see the program still executes the subroutine, but it doesn't move into the subroutine code. This is the first time we have been able to use *step over*.

EXAMPLE: USING SUBROUTINES



Did it work? Did you see PC jump to the subroutine starting address when it was called and then return to the main program when it returned?

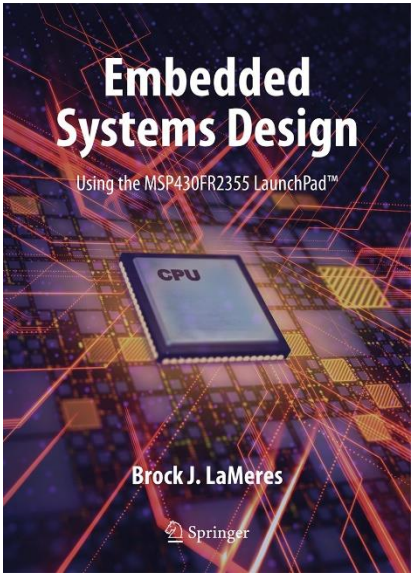
Did you see the stack store the return address when the subroutine was called?

Did you see the difference between Step Over and Step Into? Pretty neat huh?

Image Courtesy of https://neodem.wp.horizon.ac.uk/

CHAPTER 10: THE STACK AND SUBROUTINES

10.2 SUBROUTINES



www.youtube.com/c/DigitalLogicProgramming_LaMeres