# PIC Assembler 2



## MEMORY USAGE AND SUBROUTINES

# Outline



- Memory on the PIC18 processor
- Addressing modes for accessing memory
- Memory Test Program
- Program Flow: Subroutines and the Stack
- Some example programs for you to write
- Program Design

# Memory in the PIC18

There are Several different types of memory on the PIC which you can use/access

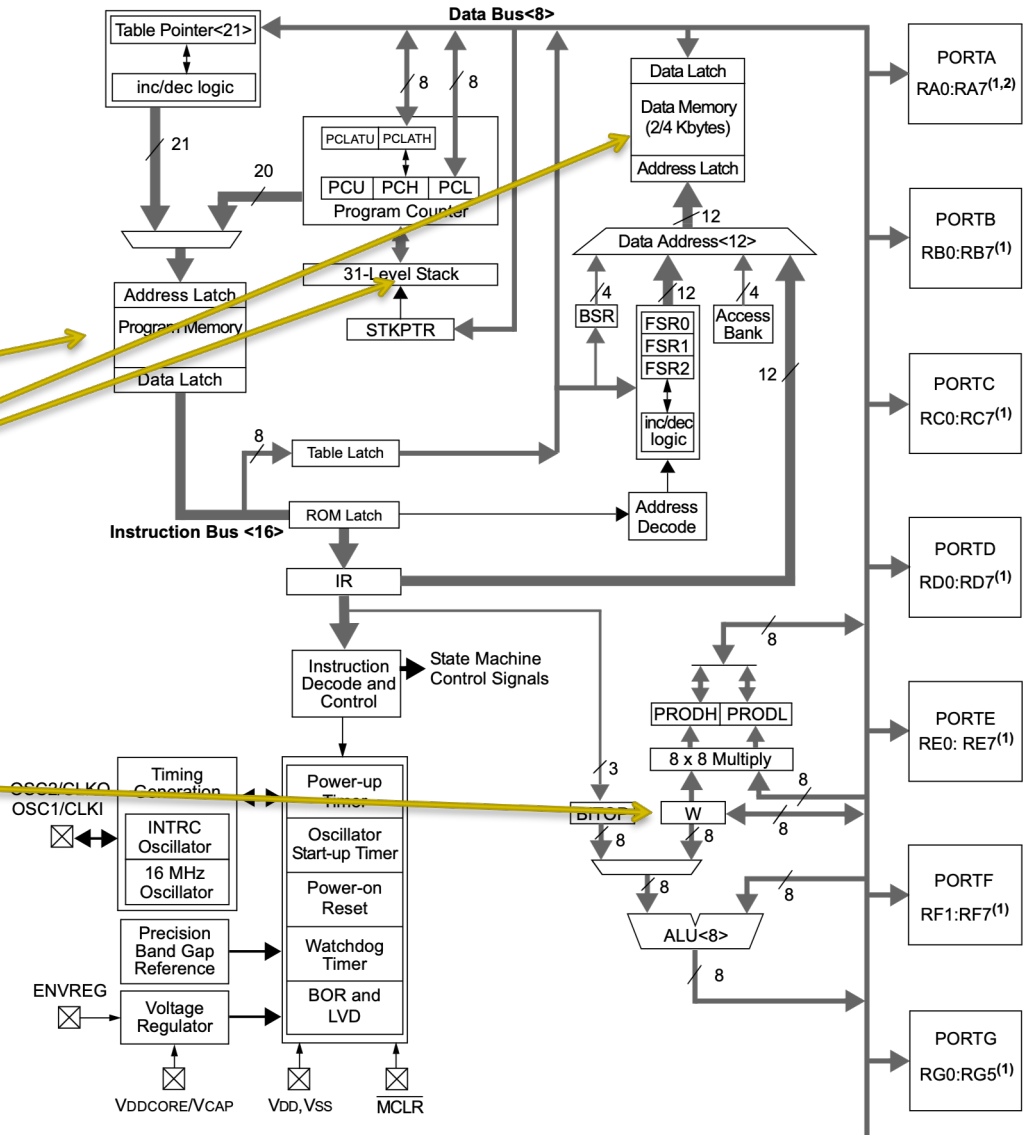
Flash Program Memory

Data SRAM (internal and external) including special function registers

Hardware stack

W register

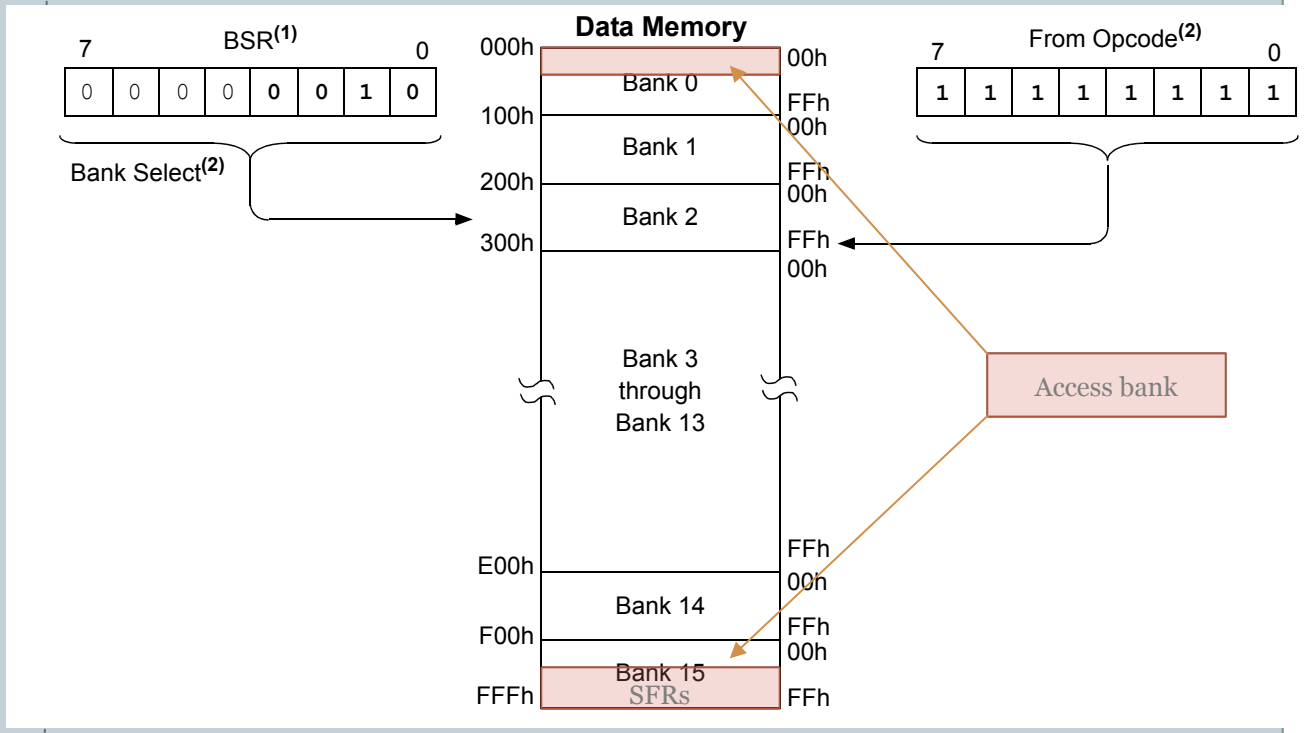
EEPROM also available



# Data Memory



- There is some internal memory (SRAM) on the PIC18 which can be used for storing data
  - Internal SRAM memory is 4K X 8 bits (4 Kilo-Bytes 0x000–0xFFF)
  - The special function registers and I/O ports occupy the last part of this memory
- All of this memory (file registers) can be addressed with single instructions in a single instruction cycle
- There is also the possibility of external memory being installed
  - Not on our boards
  - Can add if your project needs



# Addressing modes – Literal and Direct addressing



- Literal addressing
  - The operand is a literal number

```
movlw 0x3f
0x3f → W
```

```
addlw 0x02
W+2 → W
```

- Literal addressing only available to destination W, BSR or FSR registers

- Direct addressing
  - The operand is a file register and its address is embedded in the op-code

```
movf 0x3f,W,A
(0x3f) → W
```

```
addwf 0x02,W,A
(0x02)+W → W
```

- Remember, file registers can be either `Access` or `Banked`
- Destination can be `F` or `W`

# Indirect addressing data RAM using File Select Registers (FSR)



- The file select registers `FSR0`, `FSR1` and `FSR2` can hold a 12-bit address that can point to anywhere in RAM
  - Can load a 12-bit literal into an FSR with the LFSR command eg

    ```
    lfsr 0,0x140
    ```

  - Actually each FSR register is a pair of SFRs, eg `FSR1=FSR1H:FSR1L` that can be written or read independently as normal
- To access the file register at the address pointed to by `FSR0`, use the `INDF0` SFR
- Increment/decrement SFRs also available that let you move through a block of RAM (`POSTINCn`, `PREINCn`, `POSTDECn`)
- `PLUSWn` register allows *indexed addressing* by defining an offset (-128→127) to the FSR* address

```
lfsr  0, 0x140     ;load address 0x140 into FSR0
movlw 0x10         ;load 0x10 into W
addwf INDF0, f, A  ;adds W to contents of address 0x140
                   ;puts result back into address 0x140
```

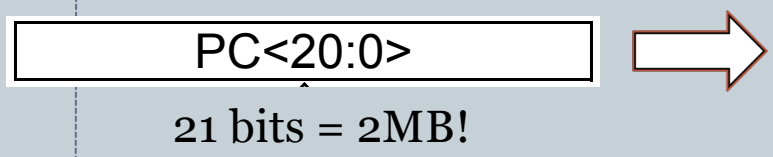
```
lfsr  2, 0x560     ;load address 0x560 into FSR2
clrf  POSTINC2, A  ;clears address 0x560
clrf  POSTINC2, A  ;clears address 0x561
clrf  POSTINC2, A  ;clears address 0x562
clrf  POSTINC2, A  ;clears address 0x563
                   ;FSR2 now contains address 0x564
```

```
lfsr  2, PORTA     ;load address 0xF80 into FSR2
movlw 2            ;loads 2 into W
bcf   PLUSW2, 4, A ;clears bit 4 of 0xF82
                   ;which happens to be PORTC
                   ;FSR2 still contains address 0xF80
```

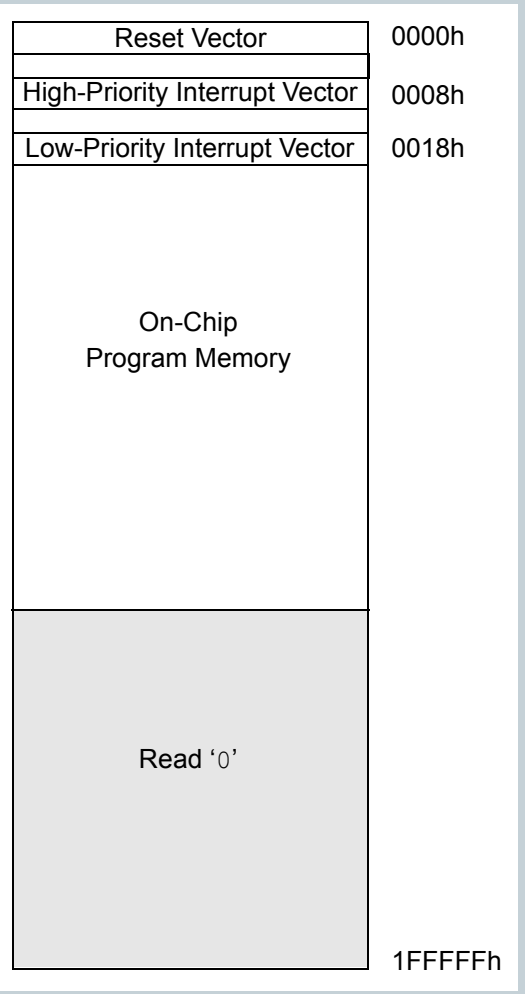
# Program Memory



- Organized as 64K 16bit words
  - 128 Kilo-Bytes of memory
  - Addressed `0x00000-0x1FFFF`
  - Instructions always start on an even byte address
  - The processor architecture can address up to 2MB of program memory
- This is where the program which is executing is stored
  - The Program Counter (PC) holds the address in program memory of the current instruction.
  - To move to the next instruction the PC must increment by 2 – its least significant bit is always a zero
- Data can be stored in program memory
  - The data stored in program memory needs to be read out using SFRs and special instructions before it can be used




21 bits = 2MB!



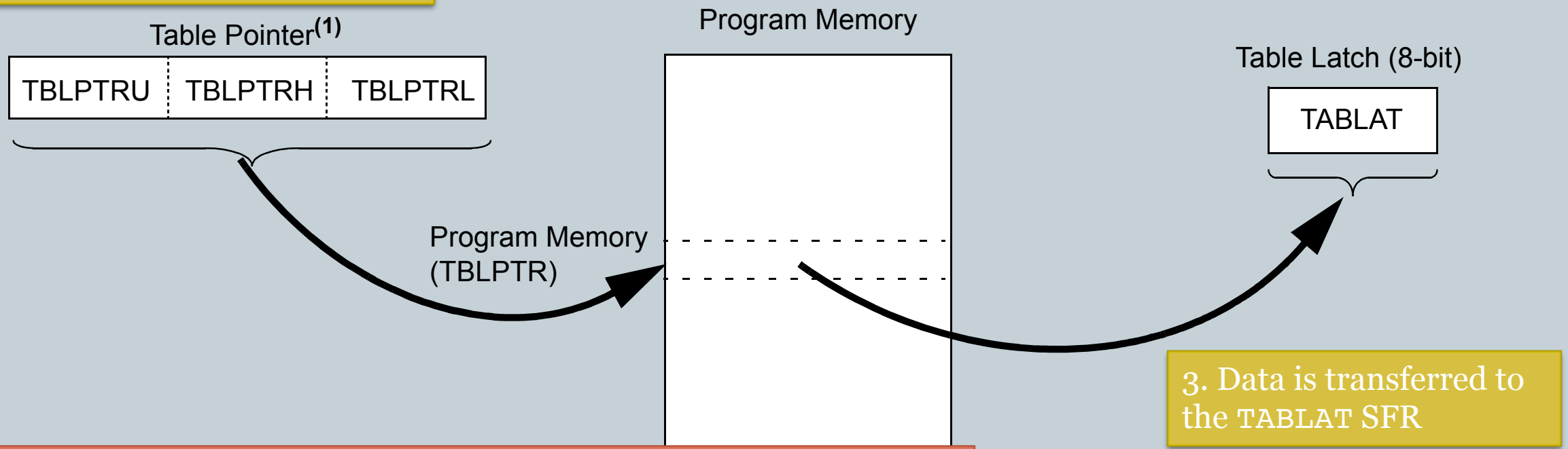
# Reading data from program memory by indirect addressing



1. Load address of location you want to read into `TBLPTRU`, `TBLPTRH`, `TBLPTRL` SFRs

Instruction: `TBLRD*`

2. Execute a `TBLRD*` instruction




3. Data is transferred to the `TABLAT` SFR

```
; ******* Programme FLASH read Setup Code ****
BCF CFGS      ; point to Flash program memory
BSF EEPGD     ; access Flash program memory
```

# TBLRD* instructions



| Mnemonic, Operands | Description | Cycles | 16-Bit Instruction Word MSb LSb | Status Affected | Notes |
|---|---|---|---|---|---|
| **DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS** | | | | | |
| TBLRD* | Table Read | 2 | 0000 0000 0000 1000 | None | |
| TBLRD*+ | Table Read with Post-Increment | | 0000 0000 0000 1001 | None | |
| TBLRD*- | Table Read with Post-Decrement | | 0000 0000 0000 1010 | None | |
| TBLRD+* | Table Read with Pre-Increment | | 0000 0000 0000 1011 | None | |
| TBLWT* | Table Write | 2 | 0000 0000 0000 1100 | None | |
| TBLWT*+ | Table Write with Post-Increment | | 0000 0000 0000 1101 | None | |
| TBLWT*- | Table Write with Post-Decrement | | 0000 0000 0000 1110 | None | |
| TBLWT+* | Table Write with Pre-Increment | | 0000 0000 0000 1111 | None | |

- There are different versions of `TBLRD*` instructions that either increment or decrement the `TBLPTR` registers before or after they have read the data
  - Useful for reading blocks of data
- `TBLWT*` instructions are also available, but its not quite that simple with FLASH memory!

# Copying data from program memory to data memory

Don't forget setup code

**db** defines a table of bytes in program memory

**EQU** defines a constant number for use in the assembler at compile time

**Low highword**, **high** and **low** are assembler functions to help you do simple calculations at compile time

See help documents in MPLAB for more info on other assembler directives

```
        ; ******* Programme FLASH read Setup Code ****
setup:  bcf     CFGS                ; point to Flash program memory
        bsf     EEPGD               ; access Flash program memory
        goto    start
        ; ******* My data and where to put it in RAM *
myTable: db     'T','h','i','s',' ','i','s',' ','j','u','s','t',
        db      ' ','s','o','m','e',' ','d','a','t','a'
        myArray EQU 0x400           ; Address in RAM for data
        counter EQU 0x10            ; Address of counter variable
        align   2         ; ensure alignment of subsequent instructions
        ; ******* Main programme *********************
start:  lfsr    0, myArray          ; Load FSR0 with address in RAM
        movlw   low highword(myTable)     ; address of data in PM
        movwf   TBLPTRU, A          ; load upper bits to TBLPTRU
        movlw   high(myTable)       ; address of data in PM
        movwf   TBLPTRH, A          ; load high byte to TBLPTRH
        movlw   low(myTable)        ; address of data in PM
        movwf   TBLPTRL, A          ; load low byte to TBLPTRL
        movlw   22                  ; 22 bytes to read
        movwf   counter, A          ; our counter register
loop:   tblrd*+                     ; move one byte from PM to TABLAT,
                                            ; increment TBLPRT
        movff   TABLAT, POSTINC0    ; move read data from TABLAT to
                                            ; (FSR0), increment FSR0
        decfsz  counter, A          ; count down to zero
        bra     loop                ; keep going until finished
        goto    0
```

# Checkout the code using Git



- Commit any recent changes you have made to your repository
- Under “Team” menu choose “Branch/Tag” then “Switch to Branch…”
- In the dialog select the “Table_read_V5.4” branch
  - You may need to tick the “Checkout as new branch” box to copy this branch to your local repository, you then need to make sure that is has the same name Table_read_V5.4 locally
- Code in your window should change to the example that reads program memory into RAM
- Step through the program in simulator or ICD mode and check that it works and that you understand what is going on
  - Watch what happens to the FSR and TBLPTR registers as you loop through the table

# Storing data in program memory



- You can create initial values for data in your program code
  - These will be stored in the program memory
    - Accessed using the labels you define
  - See the examples below for the assembler directives needed to create data in program memory
- You can copy the data into RAM (as we did on the previous pages) or use them directly from program memory
  - Access will be simpler and quicker from RAM and values can be changed

```
MyByteTable:
db 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07    ;Table loading with 8 bytes
db “a”,”s”,”c”,”i”,”i”                ;Table loading with 5 ascii characters
MyWordTable:
dw 0x0908h,0x0B0Ah,0x0D0Ch,0x0F0Eh   ; Load 4 16-bit words in the table
dw 0x1514h,0x1617h,0x1918h,0x1B1Ah   ; Next 4 16-bit words in the table
```

- Remember, words are stored little-endian, (least significant byte first)

# Program flow and subroutines



- You will want to break your program up into small pieces or *subroutines*
- Use the `call` operations to jump to these subroutines
- A special separate area of memory called the `STACK` is used to store the program address of where you came from
- When you `return` from the subroutine the microprocessor reads the address from the top of the stack back into the program counter

- PIC18 implements a HARDWARE stack that can contain up to 31 addresses
- You have to implement your own software stack if your program is more than 32 levels deep…
- …or if it has recursive subroutines!

```
        ; main code
        movlw   0x10
        movwf   0x20, A  ; store 0x10 in FR 0x20
        call    delay
        goto 0

        ; a delay subroutine
delay:  decfsz  0x20, A  ; decrement until zero
        bra delay
        return
```

# Stack example



```
013C   0E10      MOVLW 0x10
013E   6E20      MOVWF 0x20, ACCESS
0140   ECA4      CALL 0x148, 0
0142   F000      NOP
0144   EF00      GOTO 0x0
0146   F000      NOP

   ; a delay subroutine
0148   2E20      DECFSZ 0x20, F, ACCESS
014A   D7FE      BRA 0x148
014C   0012      RETURN 0
```

| W | 0x20 |
|---|---|
| 0x?? | 0x?? |

| Stack location | Contents |
|---|---|
| 00* | ---- |
| 01 | 0x00000 |
| 02 | 0x00000 |
| 03 | 0x00000 |
| 04 | 0x00000 |
| 05 | 0x00000 |
| 06 | 0x00000 |

| STKPTR | PC |
|---|---|
| 0x00 | 0x0013c |

# Stack example



```
013C   0E10      MOVLW 0x10
013E   6E20      MOVWF 0x20, ACCESS
0140   ECA4      CALL 0x148, 0
0142   F000      NOP
0144   EF00      GOTO 0x0
0146   F000      NOP

   ; a delay subroutine
0148   2E20      DECFSZ 0x20, F, ACCESS
014A   D7FE      BRA 0x148
014C   0012      RETURN 0
```

| W | 0x20 |
|---|---|
| 0x10 | 0x?? |

| Stack location | Contents |
|---|---|
| 00* | ---- |
| 01 | 0x00000 |
| 02 | 0x00000 |
| 03 | 0x00000 |
| 04 | 0x00000 |
| 05 | 0x00000 |
| 06 | 0x00000 |

| STKPTR | PC |
|---|---|
| 0x00 | 0x0013e |

# Stack example



```
013C   0E10      MOVLW 0x10
013E   6E20      MOVWF 0x20, ACCESS
0140   ECA4      CALL 0x148, 0
0142   F000      NOP
0144   EF00      GOTO 0x0
0146   F000      NOP

   ; a delay subroutine
0148   2E20      DECFSZ 0x20, F, ACCESS
014A   D7FE      BRA 0x148
014C   0012      RETURN 0
```

| W | 0x20 |
|---|---|
| 0x10 | 0x10 |

| Stack location | Contents |
|---|---|
| 00* | ---- |
| 01 | 0x00000 |
| 02 | 0x00000 |
| 03 | 0x00000 |
| 04 | 0x00000 |
| 05 | 0x00000 |
| 06 | 0x00000 |

| STKPTR | PC |
|---|---|
| 0x00 | 0x00140 |

# Stack example



```
013C   0E10      MOVLW 0x10
013E   6E20      MOVWF 0x20, ACCESS
0140   ECA4      CALL 0x148, 0
0142   F000      NOP
0144   EF00      GOTO 0x0
0146   F000      NOP

   ; a delay subroutine
0148   2E20      DECFSZ 0x20, F, ACCESS
014A   D7FE      BRA 0x148
014C   0012      RETURN 0
```

| W | 0x20 |
|---|---|
| 0x10 | 0x10 |

| Stack location | Contents |
|---|---|
| 00 | ---- |
| 01* | 0x00144 |
| 02 | 0x00000 |
| 03 | 0x00000 |
| 04 | 0x00000 |
| 05 | 0x00000 |
| 06 | 0x00000 |

| STKPTR | PC |
|---|---|
| 0x01 | 0x00148 |

# Stack example



```
013C   0E10      MOVLW 0x10
013E   6E20      MOVWF 0x20, ACCESS
0140   ECA4      CALL 0x148, 0
0142   F000      NOP
0144   EF00      GOTO 0x0
0146   F000      NOP

   ; a delay subroutine
0148   2E20      DECFSZ 0x20, F, ACCESS
014A   D7FE      BRA 0x148
014C   0012      RETURN 0
```

| W | 0x20 |
|---|---|
| 0x10 | 0x0f |

| Stack location | Contents |
|---|---|
| 00 | ---- |
| 01* | 0x00144 |
| 02 | 0x00000 |
| 03 | 0x00000 |
| 04 | 0x00000 |
| 05 | 0x00000 |
| 06 | 0x00000 |

| STKPTR | PC |
|---|---|
| 0x01 | 0x0014a |

# Stack example



```
013C   0E10      MOVLW 0x10
013E   6E20      MOVWF 0x20, ACCESS
0140   ECA4      CALL 0x148, 0
0142   F000      NOP
0144   EF00      GOTO 0x0
0146   F000      NOP

   ; a delay subroutine
0148   2E20      DECFSZ 0x20, F, ACCESS
014A   D7FE      BRA 0x148
014C   0012      RETURN 0
```

| W | 0x20 |
|---|---|
| 0x10 | 0x01 |

| Stack location | Contents |
|---|---|
| 00 | ---- |
| 01* | 0x00144 |
| 02 | 0x00000 |
| 03 | 0x00000 |
| 04 | 0x00000 |
| 05 | 0x00000 |
| 06 | 0x00000 |

| STKPTR | PC |
|---|---|
| 0x01 | 0x00148 |

# Stack example



```
013C   0E10      MOVLW 0x10
013E   6E20      MOVWF 0x20, ACCESS
0140   ECA4      CALL 0x148, 0
0142   F000      NOP
0144   EF00      GOTO 0x0
0146   F000      NOP

   ; a delay subroutine
0148   2E20      DECFSZ 0x20, F, ACCESS
014A   D7FE      BRA 0x148
014C   0012      RETURN 0
```

| W | 0x20 |
|---|---|
| 0x10 | 0x00 |

| Stack location | Contents |
|---|---|
| 00 | ---- |
| 01* | 0x00144 |
| 02 | 0x00000 |
| 03 | 0x00000 |
| 04 | 0x00000 |
| 05 | 0x00000 |
| 06 | 0x00000 |

| STKPTR | PC |
|---|---|
| 0x01 | 0x0014c |

# Stack example



```
013C   0E10      MOVLW 0x10
013E   6E20      MOVWF 0x20, ACCESS
0140   ECA4      CALL 0x148, 0
0142   F000      NOP
0144   EF00      GOTO 0x0
0146   F000      NOP

   ; a delay subroutine
0148   2E20      DECFSZ 0x20, F, ACCESS
014A   D7FE      BRA 0x148
014C   0012      RETURN 0
```

| W | 0x20 |
|---|---|
| 0x10 | 0x00 |

| Stack location | Contents |
|---|---|
| 00* | ---- |
| 01 | 0x00000 |
| 02 | 0x00000 |
| 03 | 0x00000 |
| 04 | 0x00000 |
| 05 | 0x00000 |
| 06 | 0x00000 |

| STKPTR | PC |
|---|---|
| 0x00 | 0x00144 |

# Subroutines and functions



- **`call addr`** - jump to a unit of code which does a given task (2 word instruction)
- **`rcall addr`** - to a unit of code which does a given task (1 word instruction limited distance jump)
- **`return`** jumps back to address at top of stack (1 word)
- **`retlw`** jumps back to return address storing the given literal in W on the way
- **`call addr,fast`** – calls subroutine at `addr`, but stores STATUS, BSR and W registers in shadow registers, return with **`return fast`**
  - One use only, a second `call fast` will overwrite previous contents of shadow registers
  - Useful for *interrupts* – see later

- Functions are just subroutines where you pass data back and forth
- You can use W or file registers for this but…
- … you need to make sure which file registers are being used elsewhere in your programme.

- **`pop`** and **`push`** – let you manipulate the stack pointer and stack contents via the **`TOSU:TOSH:TOSL`** SFR registers so that you can implement your own software stack

- All call and return instructions take 2 cycles because of change in PC

# Exercises:



- Exercise 1 : Last time you made a counter. Now use a subroutine to delay the speed of the counter.
- Exercise 2 : You can delay further by cascading delay subroutines.
- Exercise 3 : Memory Test Program: Write a program that writes incrementing numbers to a block of RAM using an FSR. Read them back using a second FSR, compare with what you expect to see and if it is not correct send a pattern of lights to `PORTC`.
  - What would happen if you started writing above `0xF16`

# LED sequence display



- Write a program that includes a table of data in *program* memory and then reads the contents back sequentially, byte by byte, and outputs them to the LEDs on `PORTC`.
  - Chose the bytes so that they will make a pattern when they are displayed on the LEDs
- Use delays to slow down the program so you can see the different patterns flashing as they light the LEDS on `PORTC`.
  - Allow the user to change the speed of flashing using the switches on `PORTD`
  - Allow the user to start/stop the pattern using switches on `PORTD`
- You might want to create new "Branches" for your code using Git
  - If you have created your own Fork of the lab code then you can store these new branches on Github too
  - Make your Fork private so you can control who has access to your work

# Example: 16 bit delay routine

Sometimes we need to work with numbers larger than 8 bits

There are some operations (multiply commands, `mulwf` and `mullw`) that give a 16 bit result that is stored in SFRs `PRODH:PRODL`

Other wise you need to do it yourself using carry bits and commands like

`addwfc`

`subwfb`

Or you can look at what the carry bit is doing directly.

But be careful, subtraction is effectively addition with 2's complement and a borrow is equivalent to not carry – so logic can seem reversed for sub and dec commands

```
        movlw   high(0xDEAD)    ; load 16bit number into
        movfw   0x10, A         ; FR 0x10
        movlw   low(0xDEAD)
        movwf   0x11, A         ; and FR 0x11
        call    bigdelay
        .
        .
        .

Bigdelay:
        movlw   0x00            ; W=0
Dloop:  decf    0x11, f, A      ; no carry when 0x00 -> 0xff
        subwfb  0x10, f, A      ; no carry when 0x00 -> 0xff
        bc  dloop               ; if carry, then loop again
        return                  ; carry not set so return
```

# Structured Programming



- Design your program into a main control segment, which calls sub-routines
- Subroutines should perform specific tasks or repeated functions
  - Marks will be given on the modularity of your code
  - Use a Top Down Modular Programming approach to design your code
  - Before starting to write code make a simple design diagram outlining the tasks which need to be done
    - Take a few minutes to do this, and show it to a demonstrator
- Put lots of comments in your code

# Top Down Modular Programming