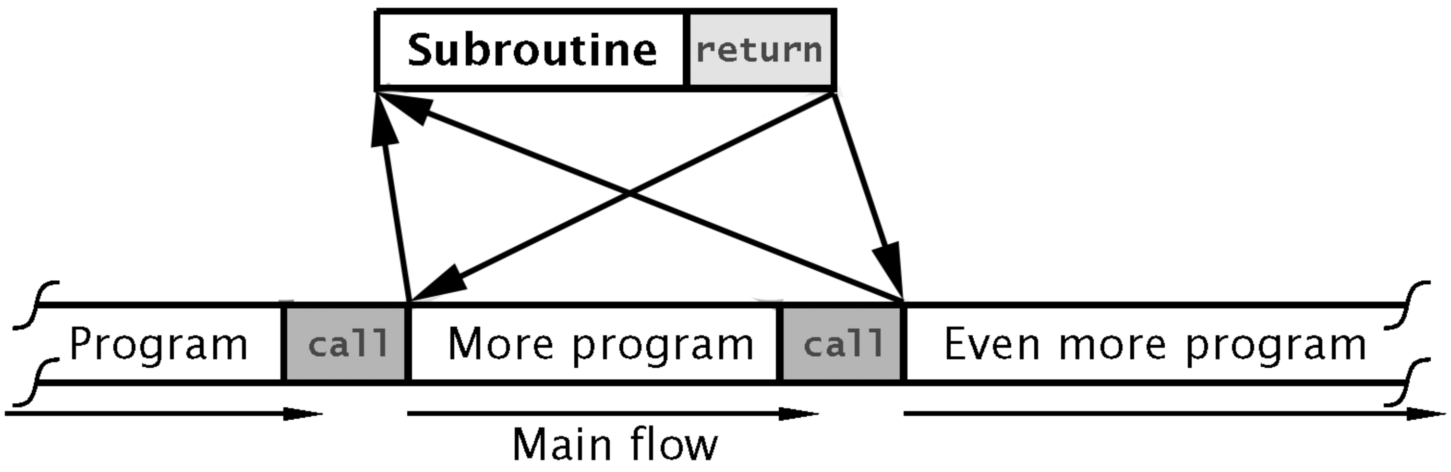
# 4.4 Subroutine and Stack

# Subroutine

- A *subroutine* is a sequence of instructions that can be called from different places in a program.
- Two reasons for creating subroutines:
  - The problem is too big: Easier to divide the problem into smaller sub-problems
  - There are several places in a program that need to perform the same operation.

# Calling subroutine: Is it just branching?

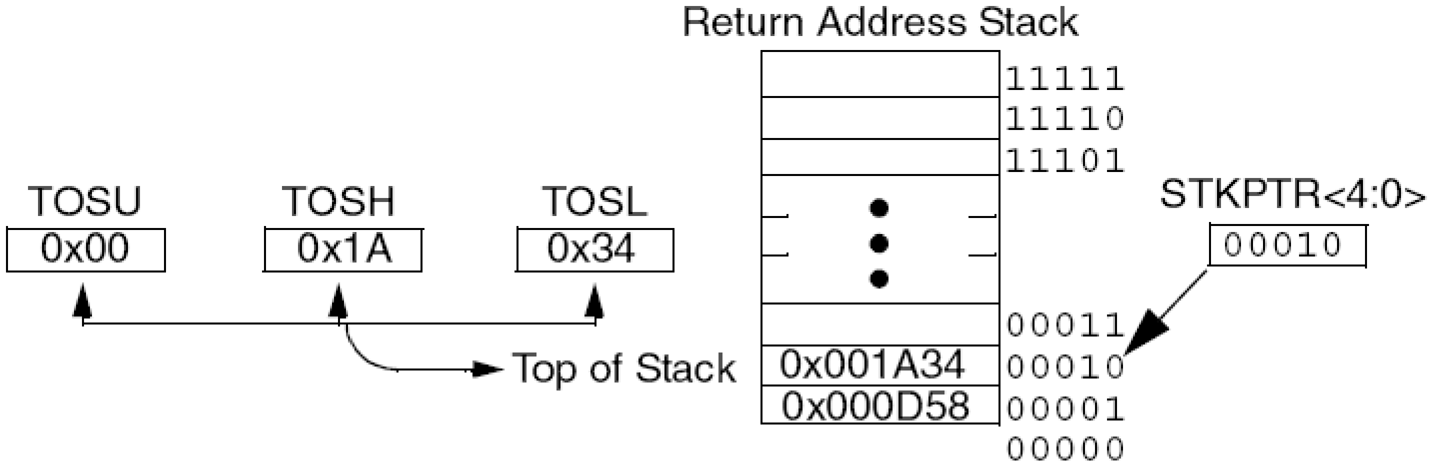



- Calling subroutine involves the “jumping” part, which is the same as branching.
- But we need to get back to the main program after the subroutine returns.
- Thus, we need a location in which the return address can be stored.

Image courtesy of S. Katzen, The essential PIC18 Microcontroller, Springer

# Return Address Stack

- The program counter is pushed to the top of the return address stack when calling a subroutine.
- 31-word-by-21-bit memory
- 5-bit stack pointer (*STKPTR*) pointing to the top of stack (*TOS*)



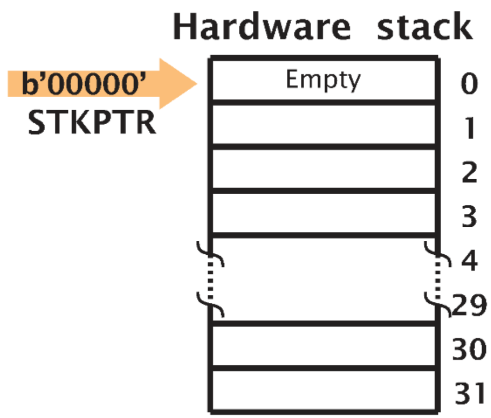
# Return Address Stack Pointer (STKPTR)

- STKPTR is 0 when powered up
- STKPTR is 1 upon the first subroutine call.
- Thus, the stack has 31 spaces for address storage.
  - Useful for nested subroutine calls.
- The TOS address is readable and writable through three registers: TOSU, TOSH, TOSL.

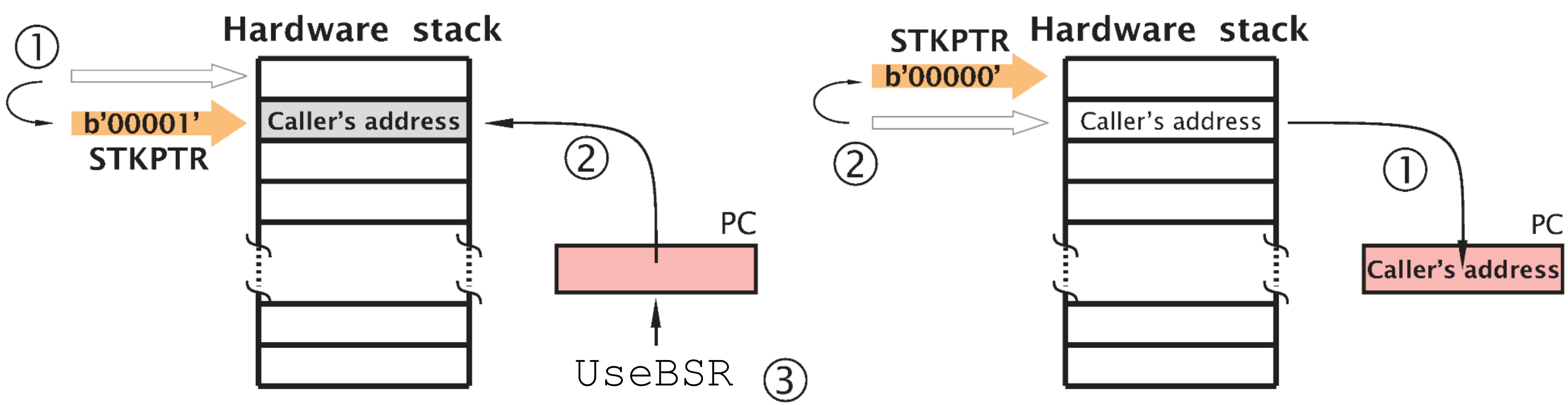
# Demonstration: `SubroutineDemo.asm`

- You can run this demonstration yourself at home by following these instructions:
  - Download the `SubroutineDemo.asm` file available at Canvas.
  - Create a new project in MPLAB. (For instructions on how to do this, refer to Tutorial Week 3).
  - Step through the program and stop when the green arrow is pointing to the call instruction.
  - Open the Hardware Stack window by selecting View → Hardware Stack.
  - In the Watch window, type STKPTR and TOS in the Symbol column.
  - Continue stepping through the program and see what happens.

# What happens when calling and returning from a subroutine?




(a) Initial state




(b) *Calling* (call UseBSR )

(c) *Returning* (return)

Image courtesy of S. Katzen, The essential PIC18 Microcontroller, Springer

# What happens when *calling* and *returning* from a subroutine?

Calling: [Figure (b)]

1. Increment Stack Pointer (STKPTR).
2. Copy the 21-bit contents of the Program Counter (PC) into the stack at the location pointed to by the STKPTR (i.e., this stored item is the address of the instruction following the call instruction)
3. The address of the instruction with label UseBSR (entry point of the subroutine) overwrites the original state of the PC (i.e., program jumps to subroutine).

Returning: [Figure (c)]

1. Copy the 21-bit address in the stack pointed to by the STKPTR into the PC.
2. Decrement the STKPTR.

# `call` instruction

- Encodes subroutine address by absolute address (similar to goto).
- Increment stack pointer (STKPTR)
- Top of Return Address Stack (TOS) = address of the instruction following the call instruction.

**CALL** **Subroutine Call**

| | |
|---|---|
| Syntax: | CALL k {,s} |
| Operands: | $0 \le k \le 1048575$<br>$s \in [0,1]$ |
| Operation: | (PC) + 4 → TOS,<br>k → PC<20:1>;<br>if s = 1,<br>(W) → WS,<br>(STATUS) → STATUSS,<br>(BSR) → BSRS |
| Status Affected: | None |

Encoding:

| | | | | |
|---|---|---|---|---|
| 1st word (k<7:0>) | 1110 | 110s | $k_7$kkk | kkkk$_0$ |
| 2nd word(k<19:8>) | 1111 | $k_{19}$kkk | kkkk | kkkk$_8$ |

| | |
|---|---|
| Description: | Subroutine call of entire 2-Mbyte memory range. First, return address (PC + 4) is pushed onto the return stack. If 's' = 1, the W, STATUS and BSR registers are also pushed into their respective shadow registers, WS, STATUSS and BSRS. If 's' = 0, no update occurs (default). Then, the 20-bit value 'k' is loaded into PC<20:1>. CALL is a two-cycle instruction. |
| Words: | 2 |
| Cycles: | 2 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'k'<7:0>, | PUSH PC to stack | Read literal 'k'<19:8>, Write to PC |
| No operation | No operation | No operation | No operation |

# `return` instruction

- Put Top of Return Address Stack (TOS) to Program Counter (PC).
- Decrement stack pointer (STKPTR)

| **RETURN** | **Return from Subroutine** |
|---|---|
| Syntax: | RETURN {s} |
| Operands: | $s \in [0,1]$ |
| Operation: | (TOS) → PC;<br>if s = 1,<br>(WS) → W,<br>(STATUSS) → STATUS,<br>(BSRS) → BSR,<br>PCLATU, PCLATH are unchanged |
| Status Affected: | None |
| Encoding: | `0000` `0000` `0001` `001s` |
| Description: | Return from subroutine. The stack is popped and the top of the stack (TOS) is loaded into the program counter. If 's'= 1, the contents of the shadow registers, WS, STATUSS and BSRS, are loaded into their corresponding registers, W, STATUS and BSR. If 's' = 0, no update of these registers occurs (default). |
| Words: | 1 |
| Cycles: | 2 |

# `rcall` instruction

- Encodes subroutine address by relative address (similar to bra).
- 11 bits are used to store the relative address of the targeting instruction → `rcall` can jump forward for a max. of 1023 instructions and backward for a max. of 1024 instructions.
- Store address in stack and increment STKPTR the same way as the call instruction.

**RCALL** **Relative Call**

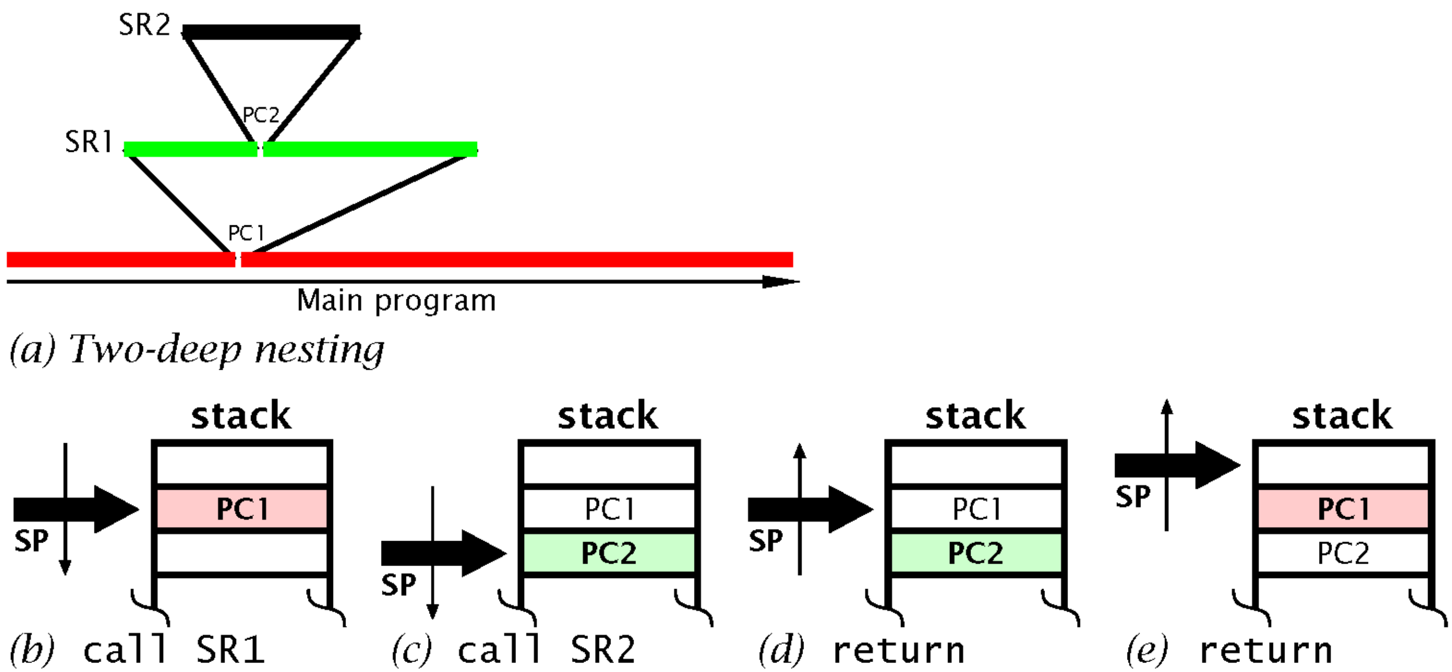
| | |
|---|---|
| Syntax: | RCALL n |
| Operands: | $-1024 \le n \le 1023$ |
| Operation: | (PC) + 2 → TOS,<br>(PC) + 2 + 2n → PC |
| Status Affected: | None |
| Encoding: | `1101` `1nnn` `nnnn` `nnnn` |
| Description: | Subroutine call with a jump up to 1K from the current location. First, return address (PC + 2) is pushed onto the stack. Then, add the 2's complement number '2n' to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is a two-cycle instruction. |
| Words: | 1 |
| Cycles: | 2 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'n'<br>PUSH PC to stack | Process Data | Write to PC |
| No operation | No operation | No operation | No operation |

# Nested subroutines

What happens if I call another subroutine (SR2) within a subroutine (SR1)?




| | | | |
|---|---|---|---|
| TOS = PC1 | TOS = PC2 | PC = PC2 | PC = PC1 |
| STKPTR = 1 | STKPTR = 2 | TOS = PC1 | STKPTR = 0 |
| | | STKPTR = 1 | |

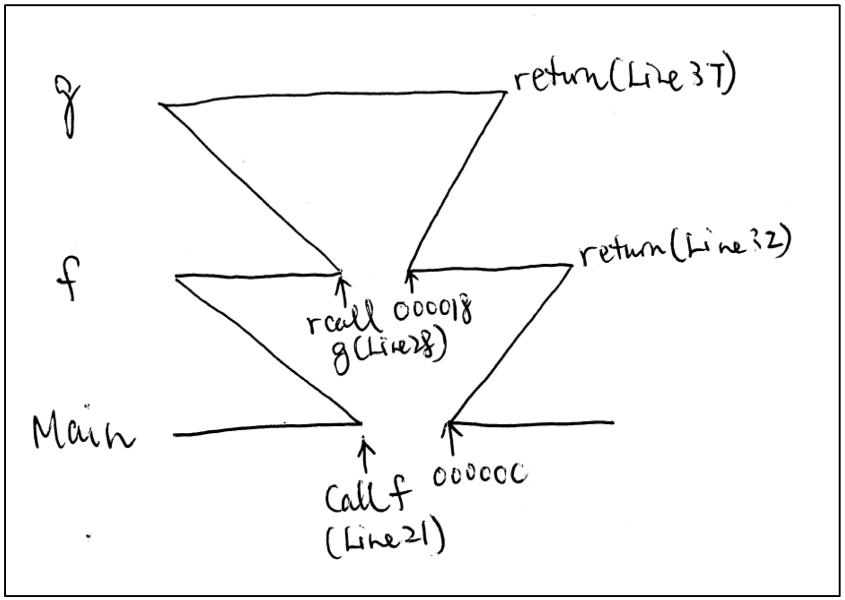
Image courtesy of S. Katzen, The essential PIC18 Microcontroller, Springer

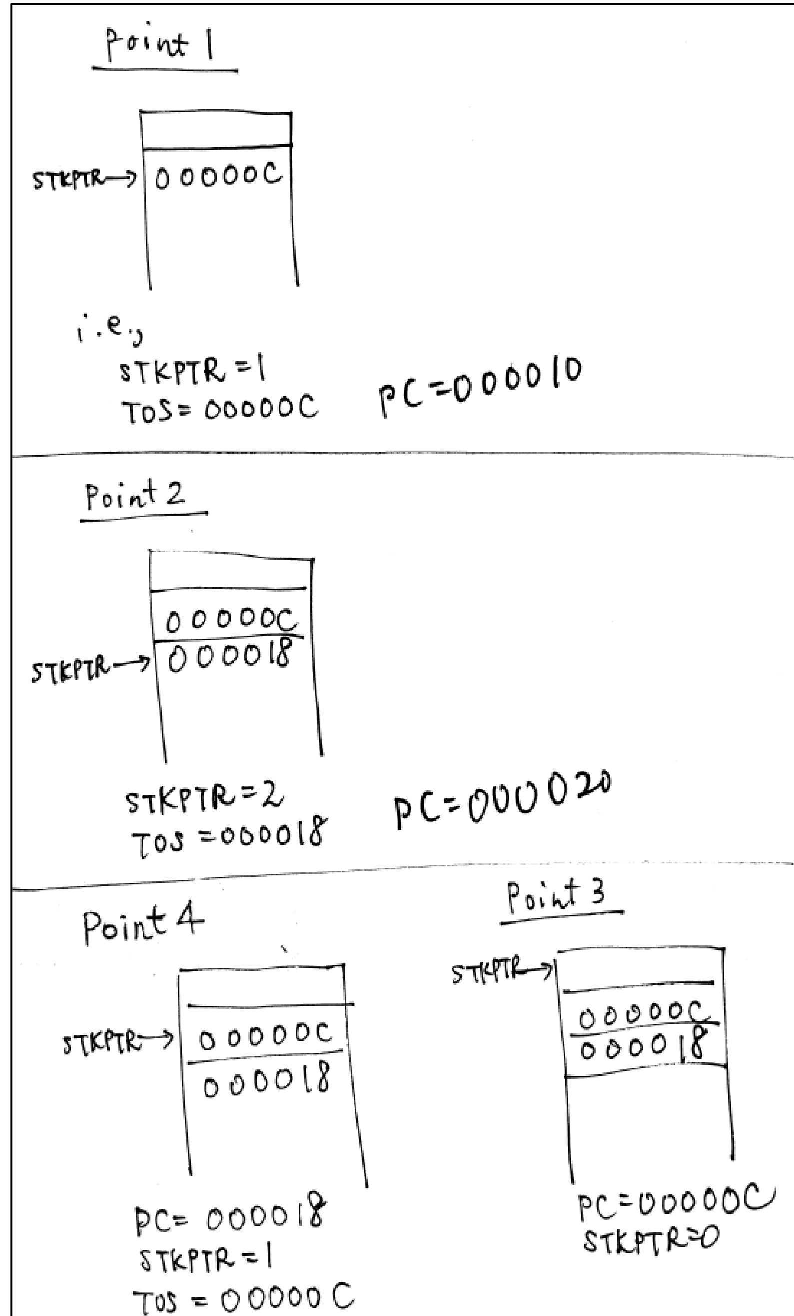
# A demonstration on nested subroutines

- Write a program to calculate the value of f(0x07) where
  - f(x) = g(x+0x18) + 0x05
  - g(x) = x+0x12
- Referring to the `NestedSubroutine.asm` file uploaded to Canvas, determine the status of the stack and the values of TOS, and STKPTR at Points 1-4.

```
Program
Memory    Machine     LINE    SOURCE
Address   Code

                                CBLOCK 0X00
                                finput
                                ginput
                                foutput
                                goutput
                                endc
                          ;-----------------------------------------------------------------------

000000                          ORG     0x0000
000000 EF?? F???                goto    Main  ;go to start of main code
                          ;-----------------------------------------------------------------------
000004 0E07               Main: movlw 0x07
000006 6E00                     movwf finput, A
000008 EC?? F???                call f                          ; Point 1
00000C 5002                     movf foutput, W, A
00000E D7FF                     bra $

000010 5000               f:    movf finput, W, A     ;[WREG] = [finput]
000012 0F18                     addlw 0x18            ;[WREG] = [WREG] + 0x18
000014 6E01                     movwf ginput, A       ;[ginput] = [WREG]
000016 D???                     rcall g                         ; Point 2
000018 5003                     movf goutput, W, A    ;[WREG] = [goutput]
00001A 0F05                     addlw 0x05            ;[WREG] = [WREG] + 0x05
00001C 6E02                     movwf foutput, A      ;[foutput] = [WREG]
00001E 0012                     return                          ; Point 3

000020 5001               g:    movf ginput, W, A; [WREG] = [ginput]
000022 0F12                     addlw 0x12; [WREG] = [WREG] + 0x12
000024 6E03                     movwf goutput, A; [goutput] = [WREG]
000026 0012                     return                          ; Point 4
```

# Flow Diagram of the Program



# You should be able to ....

- Code PIC subroutine.
- Describe the stack and its use in subroutine.
- Discuss how branching and subroutine calling affect the pipelining mechanism.
- Describe the difference of the `call` and `rcall` instructions.

# 4.5 Register Indirect Addressing Mode in PIC18

# Outline

Dealt with in Chapter 2:

- Immediate addressing mode
- Direct addressing mode

New:

- Register Indirect addressing mode

# Immediate Addressing Mode

- The operand is a literal constant
- The instruction has a ‘L’ (literal)
- Can be used in loading information and performing arithmetic and logic operations ***ONLY in the WREG register***
- Examples:
  - **`movlw 0x25; load 0x25 into [WREG]`**
  - **`sublw  D’62’; subtract [WREG] from 62`**
  - **`addlw  0x40; add WREG with 0x40`**

# Register Direct Addressing Mode

- The operand data is in a file register in data memory.
- The address of the file register is provided as a part of the instruction.
- Example:
  - **`movwf 0x40, A; copy [WREG] into file register location 0x040`**

# Destination Option in Direct Addressing Mode

- Provides an option to store the result either in WREG or in file register.
- Example:

```
movlw   0              ;[WREG] = 0
movwf   0x20, A        ;[0x20] = 0, [WREG] = 0
incf    0x20, W, A     ;[0x20] = 0, [WREG] = 1
incf    0x20, W, A     ;[0x20] = 0, [WREG] = 1
incf    0x20, F, A     ;[0x20] = 1, [WREG] = 1
incf    0x20, F, A     ;[0x20] = 2, [WREG] = 1
```

# Register Indirect Addressing Mode

- Suppose you want to copy value 0x55 to location 0x040 to 0x044.
- A fixed address must be specified in direct addressing mode as an operand.
- Thus, using direct addressing mode, one instruction copies to one register:

```
movlw 0x55
movwf 0x40, A
movwf 0x41, A
movwf 0x42, A
movwf 0x43, A
movwf 0x44, A
```

- How about if you want to copy 0x55 to 1000 consecutive memory locations?

# Register Indirect Addressing Mode

- Three registers known as *file select registers* (FSRx, where x = 0, 1, 2) store addresses of the data memory location (i.e., pointers).
- A FSR is a 12-bit register which is split into two 8-bit registers, known as FSRxL and FSRxH.
- To load a data memory address into a FSR, use LFSR (Load FSR):
  - **LFSR 0, 0x030; load FSR0 with 0x030**
  - **LFSR 1, 0x040; load FSR1 with 0x040**
  - **LFSR 2, 0x06F; load FSR2 with 0x06F**

# Register Indirect Addressing Mode

- FSRx is associated with a INDFx register (where x = 0, 1, 2).
- When reading from (writing to) the INDFx register, we are reading from (writing to) the file register pointed to by the FSR
- Example:

```
LFSR 0, 0x030    ;FSR0 points to data memory
                 address 0x030
movwf INDF0      ;copy the content of WREG into
                 data memory address 0x030
```

# Examples

**e.g., Write a program to copy the value 0x55 to location 0x40 to 0x44**

**Direct Addressing Mode**

```
movlw 0x55
movwf 0x40, A
movwf 0x41, A
movwf 0x42, A
movwf 0x43, A
movwf 0x44, A
```

**Indirect Addressing Mode**

```
COUNT equ 0x00
movlw 0x05
movwf COUNT, A
movlw 0x55
LFSR 0, 0x040
Loop: movwf INDF0
      incf FSR0L, F
      decfsz COUNT, F, A
      bra Loop
```

# Register Indirect Addressing Mode

- Indirect addressing mode allows looping
- However, we incremented only [FSRxL]
- To deal with FSRxH, we need to use branching instructions conditioned on the Carry Bit.
- Solution: Auto-increment option

# Auto-increment option for FSR

- POSTDECx
  - **`movwf POSTDEC0`** does what `movwf INDF0` did, but in addition, [FSR0] will be *decremented* by 1 *after* the execution
- POSTINCx
  - **`movwf POSTINC0`** *increments* [FSR0] by 1 *after* the move operation.
- PREINCx
  - **`movwf PREINC0`** *increments* [FSR0] by 1 *before* the move operation.
- PLUSWx
  - **`movwf PLUSW0`** *adds an offset to [FSR0] that equals to the content of WREG* *before* the move operation. *However, the content of FSR0 will not be modified after operation (different from previous three SFRs)*

# Example

- Try to compile and run `FSRAutoIncOptions.asm` to verify what I describe here.
- Before operation:

  **`[FSR0] = 0x020, [WREG] = 0x05`**

- After operation:
  - **`movwf POSTDEC0:`**

    **`[FSR0] = 0x01F, [020] = 0x05`**
  - **`movwf POSTINC0`**

    **`[FSR0] = 0x021, [020] = 0x05`**
  - **`movwf PREINC0`**

    **`[FSR0] = 0x021, [021] = 0x05`**
  - **`movwf PLUSW0`**

    **`[FSR0] = 0x020 (unchanged), [025] = 0x05`**

# Previous example revisited

**e.g., Write a program to copy the value 0x55 to location 0x040 to 0x044**

```
COUNT equ 0x00
movlw 0x05
movwf COUNT, A
movlw 0x55
LFSR 0, 0x040
Loop: movwf POSTINC0
      decfsz COUNT, F, A
      bra Loop
```

# Introduce movff before next example...

| **MOVFF** | **Move f to f** | | | |
|---|---|---|---|---|
| Syntax: | MOVFF $f_s$,$f_d$ | | | |
| Operands: | $0 \le f_s \le 4095$<br>$0 \le f_d \le 4095$ | | | |
| Operation: | $(f_s) \rightarrow f_d$ | | | |
| Status Affected: | None | | | |
| Encoding: | | | | |
| 1st word (source) | 1100 | ffff | ffff | $ffff_s$ |
| 2nd word (destin.) | 1111 | ffff | ffff | $ffff_d$ |

# Example

- Copy a block of 5 bytes of data from data memory locations starting from 0x030 to data memory locations starting from 0x060

```
        COUNT equ 0x00
        movlw 0x05
        movwf COUNT, A
        lfsr 0, 0x030
        lfsr 1, 0X060
Loop:   movff POSTINC0, POSTINC1
        decfsz COUNT, F, A
        bra Loop
```

# Example

Add the contents in data memory locations 0x040-043 together and place the result in locations 0x006 and 0x007

```
COUNT equ 0x00
L_BYTE equ 0x06
H_BYTE equ 0x07
           LFSR 0, 0X040
           movlw 0x04
           movwf COUNT, A
           clrf H_BYTE, A
           clrf L_BYTE, A
Loop:      movf POSTINC0, W;
           addwf L_BYTE, F, A;
           bnc Next
           incf H_BYTE, F, A;
Next:      decfsz COUNT, F, A
           bra Loop
```

# You should be able to .....

- List all the addressing modes of the PIC18 microcontroller.
- Contrast and compare addressing modes.
- Code PIC instructions using each addressing mode.
- Access the data memory file register using various addressing modes.

# 4.6 Lookup Table

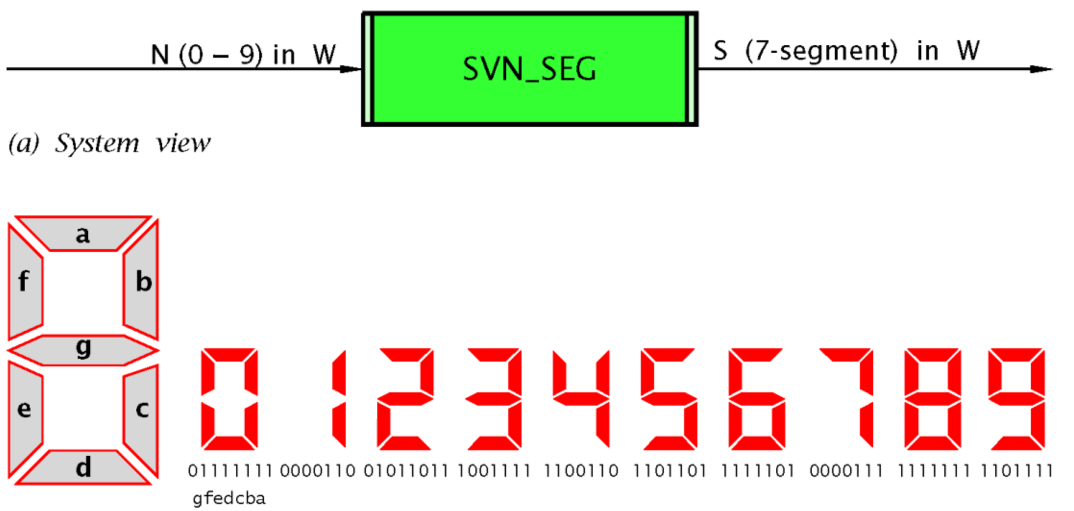
# Why Lookup Table?

- Suppose you want to compute the value of the function *f* at *x*.
  - Computation may be complicated, thereby taking long time to get the result.
  - The function cannot be represented analytically.
- You may represent the results in an array or <u>lookup table</u> and retrieve the suitable answer as needed without going through the computation once again. This is much more efficient.
- There are two methods of implementing lookup table in PIC18:
  - Computed goto
  - Using table read operations

# Applications: Mapping number to digit pattern to be displayed in LED

Use lookup table to implement a subroutine with the following input/output:

- Input: A number N ranging from 0 to 9 stored in WREG.
- Output: The 7-segment LED digit pattern corresponding to the input number.




*(a) System view*

*(b) The 7-segment font*

Image courtesy of S. Katzen, The essential PIC18 Microcontroller, Springer

# Computed Goto

- Add an offset to PCL to access the appropriate item in the table
- `retlw` will load the desired item to WREG
- e.g., Implement a look-up table in program memory, and write a program to find y where $y(x) = x^2+5$, and x is between 0 and 4.

# Computed Goto

e.g., **f:**

```
movf x, W, A
addwf WREG, W
addwf PCL, F
PC+0 ⟶ retlw d‘5’;   f(0)
PC+2 ⟶ retlw d‘6’;   f(1)
PC+4 ⟶ retlw d‘9’;   f(2)
PC+6 ⟶ retlw d‘14’;  f(3)
PC+8 ⟶ retlw d‘21’;  f(4)
```

# Disadvantage of computed goto

- 16-bit instruction is used to store each 8-bit value. In each instruction,
  - `retlw` takes 8-bit
  - the value stored takes 8-bit
- For a more efficient use of program memory space, use *table read operation*.

# Using Table Read Operations

- We can use program memory to store fixed data.
- Use the `db` (define byte) directive to define fixed data in program memory.
- Example:

```
        org 0x000500
array db d'5', d'6', d'9', d'14', d'21'
```

| Program Memory Address | Value in decimal |
|---|---|
| 000500 | 5 |
| 000501 | 6 |
| 000502 | 9 |
| 000503 | 14 |
| 000504 | 21 |

# Reading Data Using TBLPTR and TABLAT

- To access data in a location in program memory, we need a register specifying which location we want to access → TBLPTR
- TBLPTR must have 21 bits to address the whole range of program memory.
- TBLPTR is divided into three 8-bit parts: TBLPTRL (low), TBLPTRH (high), TBLPTRU (upper).
- We need a register to store the data fetched by a table read operation → TABLAT

# Table Read Operations

- `tblrd*`
  - After read, TBLPTR stays the same
- `tblrd*+`
  - Reads then *increments* TBLPTR
- `tblrd*-`
  - Reads then *decrements* TBLPTR
- `tblrd+*`
  - Increments TBLPTR then reads

# Examples

```
org 0x000500
array db d'5', d'6', d'9',...
```

Before operation:

**[TBLPTR] = 000500**

After operation:

- **tblrd***

  **[TABLAT] = 05; [TBLPTR] = 000500**

- **tblrd*+**

  **[TABLAT] = 05; [TBLPTR] = 000501**

- **tblrd*-**

  **[TABLAT] = 05; [TBLPTR] = 0004FF**

- **tblrd+***

  **[TABLAT] = 06; [TBLPTR] = 000501**

# Steps for Table Read from Program Memory

1. Write an array to program memory by using the `db` directive.
2. Specify the table pointer (TBLPTR) from which data is read.
3. Perform a `tblrd` instruction.
4. The data read is stored in the table latch (TABLAT).

# Example

Implement a look-up table in program memory, and write a program to find y where $y(x) = x^2+5$, and x is between 0 and 4. (implemented using computed goto previously)

Steps to implement this lookup table using `tblrd` operation:

1. In the .asm file, declare `array`:

```
            org 0x000500
      array db d'5', d'6', d'9', d'14', d'21'
```

2. Put <u>starting address</u> of array into `TBLPTR`:

```
movlw   upper array
movwf   TBLPTRU
movlw   high array
movwf   TBLPTRH
movlw   low array
movwf   TBLPTRL
```

array

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 5 | 0 | 0 |

TBLPTR

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 5 | 0 | 0 |
| TBLPTRU | | TBLPTRH | | TBLPTRL | |

3. Perform `tblrd*+` x+1 times (note: x is the argument in y(x))
4. Read the result from `TABLAT`.

# Demonstration

- We will look at the program `LookupTable.asm` together.

# Applications: Mapping number to digit pattern to be displayed in LED

Use lookup table to implement a subroutine with the following input/output:

- Input: A number N ranging from 0 to 9 stored in WREG.
- Output: The 7-segment LED digit pattern corresponding to the input number.

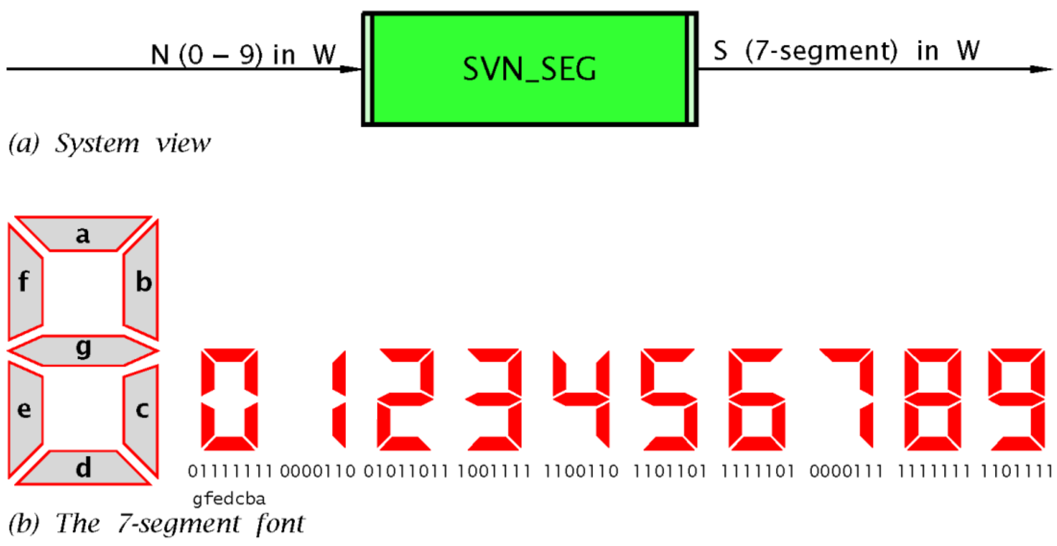



Image courtesy of S. Katzen, The essential PIC18 Microcontroller, Springer

# You should be able to .....

- Use `retlw` and `tblrd` to build a lookup table.
- Read contents from a lookup table.
- Describe the disadvantage of the computed goto (`retlw`) method for implementing a lookup table.
- Build lookup tables for LED display.