3.5   addressing modes

- the ways of accessing data are called addressing modes

- data could be in register, program memory, data memory, or in instruction (*immediate value*)

- addressing modes are determined by designer

- PIC18 provides 4 addressing modes:
    immediate
    direct
    register indirect
    indexed-ROM

3.5.1 immediate addressing mode

- operand is part of instruction (*immediately after the opcode, you can see it in program memory window*)

- in PIC, immediate value is also called literal

- use this addressing mode to load data into WREG

- also use this addressing mode for arithmetic and logic instructions

Examples

```
movlw        0x25
sublw        d'62'            immediate values
andlw        b'01000000'
```

```
count   EQU   0x30           use EQU to access immediate data
…
movlw   count            ; WREG = 30H
```

Cannot load immediate value to file registers!

```
        movlw
        movwf
```

3.5.2 direct addressing mode

- the entire data memory can be accessed using direct addressing mode

- address (memory *lo*cation) of operand is known

- address is given as part of the instruction (*you can see it in program memory window*)

Address	Machine Code	
0000	0E56	MOVLW 0x56
0002	6E40	MOVWF 0x40, ACCESS
0004	C040	MOVFF 0x40, 0x50
0006	F050	

56 → WREG

WREG → loc 40H

(Loc 40H) → loc 50H

MOVWF

0110	111A	ffff	ffff

$0 \leq$ ffff ffff $\leq$ FF

A = 0   use access bank

A = 1   use bank pointed to by BSR

What is the difference between

```
INCF fileReg, W
INCF fileReg, F
```

movlw	0	
movwf	0x20	
incf	0x20, W	w = 1, 0x20 = 0
incf	0x20, F	w = 1, 0x20 = 1
incf	0x20	0x20 = 2

We have the option to save the result in fileReg or WREG.

What is the difference between

```
DECF fileReg, W
DECF fileReg, F
```

	clrf	TRISB
	movlw	5
	movwf	MyReg
	clrf	PORTB
Loop:	comf	PORTB
	decf	MyReg, F
	bnz	Loop
	setf	PORTB

What happens when decf MyReg, F is changed to decf MyReg, W?

inf loop

SFR Registers and their addresses

- can be accessed by their names
- can be accessed by their addresses

- Which is easier to remember?

```
MOVWF PORTB
    or
MOVWF 0xF81
```

F80h	PORTA
F81h	PORTB
F82h	PORTC
F83h	PORTD
F84h	PORTE
F85h	----
F86h	----
F87h	----
F88h	----
F89h	LATA
F8Ah	LATB
F8Bh	LATC
F8Ch	LATD
F8Dh	LATE
F8Eh	----
F8Fh	----
F90h	----
F91h	----
F92h	TRISA
F93h	TRISB
F94h	TRISC
F95h	TRISD
F96h	TRISE
F97h	----
F98h	----
F99h	----
F9Ah	----
F9Bh	----
F9Ch	----
F9Dh	PIE1
F9Eh	PIR1
F9Fh	IPR1

FA0h	PIE2
FA1h	PIR2
FA2h	IPR2
FA3h	----
FA4h	----
FA5h	----
FA6h	----
FA7h	----
FA8h	----
FA9h	----
FAAh	----
FABh	RCSTA
FACh	TXSTA
FADh	TXREG
FAEh	RCREG
FAFh	SPBRG
FB0h	----
FB1h	T3CON
FB2h	TMR3L
FB3h	TMR3H
FB4h	----
FB5h	----
FB6h	----
FB7h	----
FB8h	----
FB9h	----
FBAh	CCP2CON
FBBh	CCPR2L
FBCh	CCPR2H
FBDh	CCP1CON
FBEh	CCPR1L
FBFh	CCPR1H

FC0h	----	
FC1h	ADCON1	
FC2h	ADCON0	
FC3h	ADRESL	
FC4h	ADRESH	
FC5h	SSPCON2	
FC6h	SSPCON1	
FC7h	SSPSTAT	
FC8h	SSPADD	
FC9h	SSPBUF	
FCAh	T2CON	
FCBh	PR2	
FCCh	TMR2	
FCDh	T1CON	
FCEh	TMR1L	
FCFh	TMR1H	
FD0h	RCON	
FD1h	WDTCON	
FD2h	LVDCON	
FD3h	OSCCON	
FD4h	----	
FD5h	T0CON	
FD6h	TMR0L	
FD7h	TMR0H	
FD8h	STATUS	
FD9h	FSR2L	
FDAh	FSR2H	
FDBh	PLUSW2	*
FDCh	PREINC2	*
FDDh	POSTDEC2	*
FDEh	POSTINC2	*
FDFh	INDF2	*

|		
FE0h	BSR	
FE1h	FSR1L	
FE2h	FSR1H	
FE3h	PLUSW1	*
FE4h	PREINC1	*
FE5h	POSTDEC1	*
FE6h	POSTINC1	*
FE7h	INDF1	*
FE8h	WREG	
FE9h	FSR0L	
FEAh	FSR0H	
FEBh	PLUSW0	*
FECh	PREINC0	*
FEDh	POSTDEC0	*
FEEh	POSTINC0	*
FEFh	INDF0	*
FF0h	INTCON3	
FF1h	INTCON2	
FF2h	INTCON	
FF3h	PRODL	
FF4h	PRODH	
FF5h	TABLAT	
FF6h	TBLPTRL	
FF7h	TBLPTRH	
FF8h	TBLPTRU	
FF9h	PCL	
FFAh	PCLATH	
FFBh	PCLATU	
FFCh	STKPTR	
FFDh	TOSL	
FFEh	TOSH	
FFFh	TOSU	

- SFRs have addresses from F80H to FFFH

- in listing file (*.lst*), you will see that the SFR names are replaced with their addresses

- WREG is one of the SFRs and has address FE8h

```
000000                      00008              ORG    0x0000
000000 EF80 F000            00009              goto   Main
                            00010
000100                      00011              ORG    0x0100
000100 0E0F                 00012 Main:        movlw  0x0F
000102 6EC1                 00013              movwf  ADCON1
000104 6A95                 00014              clrf   TRISD
000106 6A83                 00015              clrf   PORTD
000108                      00016 Loop:
000108 0E5B                 00017              movlw  0x5B
00010A 6E83                 00018              movwf  PORTD
00010C EF84 F000            00019              goto   Loop
                            00020              END
```
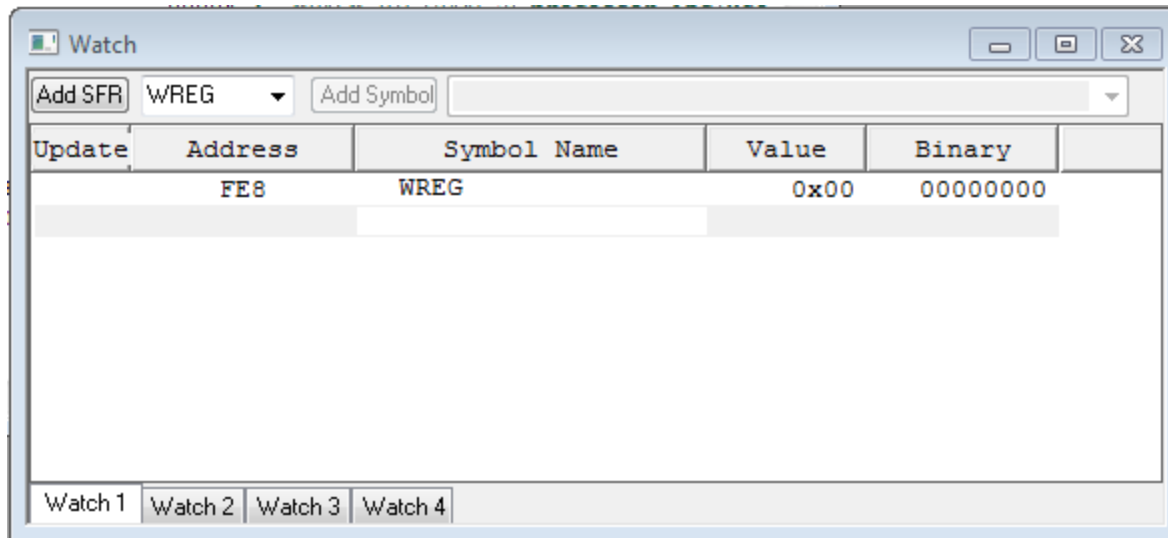
3.5.3 register indirect addressing mode

- register is used as pointer to data memory location

- 3 registers are used: FSR0, FSR1, and FSR2
  (*FSR means file select register*)

- each is 12-bit (*can access the entire 4KB data memory*)

- to use FSRx, load the data memory address

  lfsr   0, 0x30   ; load FSR0 with 0x30

- FSRx has low-byte (FSRxL) and high-byte (FSRxH)

- FSRxH uses only the lower 4 bits

- each register is associated with INDFx (indirect register for using FSRx)

```
lfsr        0, 0x30        ; FSR0 = 30H
movwf       INDF0          ; move contents of WREG to the location pointed by FSR0
```

N.B. "indf0 is wrong! "fsrxl" is wrong!

```
cblock 0x020
    data_table: 5
    R1
    R2
endc
```

```
Main: ORG            0x0000
        lfsr        0, data_table   ; FSR0=020h
        movlw   0x55
        movwf   R1
        movwf   R2
        movwf   INDF0
Here: goto      Here
```

immediate addressing

direct addressing

register indirect addressing

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
020	55	00	00	00	00	55	55	00	00	00	00	00	00	00	00	00
030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Hex   Symbolic

```
cblock 0x000              Main ORG              0x0000
    R2                          lfsr  0, data_table
endc                            movlw  01H
cblock 0x120  ←————————  movlb  high R1
    data_table: 5               movwf  R1
    R1  0x125                   movlw  02H
endc                            movwf  R2
                                movlw  03H
                                movwf  INDF0
                        Here goto here
```

R1 address = 125H
R2 address = 000H
data_table address = 120H

Program Memory

	Line	Address	Opcode	Disassembly
	1	0000	EE01	LFSR 0, 0x120
	2	0002	F020	NOP
	3	0004	0E01	MOVLW 0x1
	4	0006	0101	MOVLB 0x1
	5	0008	6F25	MOVWF 0x25, BANKED
	6	000A	0E02	MOVLW 0x2
	7	000C	6E00	MOVWF 0, ACCESS
	8	000E	0E03	MOVLW 0x3
	9	0010	6EEF	MOVWF 0xfef, ACCESS
⇒	10	0012	EF09	GOTO 0x12
	11	0014	F000	NOP
	12	0016	FFFF	NOP

Program Memory

Line	Address	Opcode	Disassembly
1	0000	EE01	LFSR 0, 0x120
2	0002	F020	NOP
3	0004	0E01	MOVLW 0x1
4	0006	0101	MOVLB 0x1
5	0008	6F25	MOVWF 0x25, BANKED
6	000A	0E02	MOVLW 0x2
7	000C	6E00	MOVWF 0, ACCESS
8	000E	0E03	MOVLW 0x3
9	0010	6EEF	MOVWF 0xfef, ACCESS
10	0012	EF09	GOTO 0x12
11	0014	F000	NOP
12	0016	FFFF	NOP

File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
000	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
120	03	00	00	00	00	01	00	00	00	00	00	00	00	00	00	00

Advantages:

- can access data dynamically

- can copy large amount of data to data memory efficiently, e.g. use a loop (*looping is not possible in direct addressing mode*)

Example:
Write a program to copy the value 55H into data memory locations 40H to 44H using
a) direct addressing mode
b) register indirect addressing mode without a loop
c) register indirect addressing mode with a loop

a) direct addressing mode

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

b)   register indirect addressing mode without a loop

```
movlw           0x55
lfsr            0, 0x40
movwf           INDF0
incf            FSR0L, f
movwf           INDF0
incf            FSR0L, f
movwf           INDF0
incf            FSR0L, f
movwf           INDF0
incf            FSR0L, f
movwf           INDF0
incf            FSR0L, f
movwf           INDF0
```

There is no such instruction as incf FSR0, f

c) register indirect addressing mode with a loop

```
        count           EQU     0x10
        movlw           5
        movwf           count
        lfsr            0, 0x40
        movlw           0x55
loop:   movwf           INDF0
        incf            FSR0L, f   w=55
        decf            count, f
        bnz             loop
```

Summary

- immediate addressing mode

- direct addressing mode

- register indirect addressing mode

3.6 look-up table, stack, subroutine

- look-up table, table processing

- stack

- subroutine call

3.6.1 look-up table and table processing

- we can use program memory to store data

- program ROM has enough space (2 MB!)

- can store and access 8-bit fixed data using the DB (*define byte*) directive

- numbers can be in decimal (d' '), binary (b' '), hex, or ASCII (single character ' ', string " ")

Example:

```
                    ORG     0x500
Label                               Data
        DATA1       DB      d'28'           @500
        DATA2       DB      b'00110101'     @501
        DATA3       DB      0x39            @502


              ORG   0x510

        DATA4       DB      'z'         Single character @510
        DATA5       DB      "Hello All"     String @511
                                        next data start @51A
```

- Program counter is 21-bit, which is used to point to any location in ROM space

- How to fetch data from the code space?

- We need SFR to point to the data to be – register indirect ROM addressing mode

- This addressing mode is often called **table processing**

- To read the fixed data byte, we need an address pointer (*points to the data in ROM*) and a register (*to store the data*)

- TBLPTR is a 21-bit register (*there is no instruction to load the 21-bit address to TBLPTR!*)

- TBLPTR is divided into 3 8-bit registers
    TBLPTRL **(low)**   **TBLPTRH** **(high)**
    load these registers if data are stored in 0000H - FFFFH
    TBLPTRU **(upper)** 5-bit
    load this register if data are stored in 10000H or beyond

- Storage register **TBLLAT**

F80h	PORTA	F A0h	PIE2	F C0h	----	FE0h	BSR		
F81h	PORTB	F A1h	PIR2	F C1h	ADCON1	FE1h	FSR1L		
F82h	PORTC	F A2h	IPR2	F C2h	ADCON0	FE2h	FSR1H		
F83h	PORTD	F A3h	----	F C3h	ADRESL	FE3h	PLUSW1	*
F84h	PORTE	F A4h	----	F C4h	ADRESH	FE4h	PREINC1	*
F85h	----	F A5h	----	F C5h	SSPCON2	FE5h	POSTDEC1	*
F86h	----	F A6h	----	F C6h	SSPCON1	FE6h	POSTINC1	*
F87h	----	F A7h	----	F C7h	SSPSTAT	FE7h	INDF1	*
F88h	----	F A8h	----	F C8h	SSPADD	FE8h	WREG	
F89h	LATA	F A9h	----	F C9h	SSPBUF	FE9h	FSR0L	
F8Ah	LATB	FAAh	----	F CAh	T2CON	FEAh	FSR0H	
F8Bh	LATC	FABh	RCSTA	F CBh	PR2	FEBh	PLUSW0	*
F8Ch	LATD	FACh	TXSTA	FCCh	TMR2	FECh	PREINC0	*
F8Dh	LATE	FADh	TXREG	FCDh	T1CON	FEDh	POSTDEC0	*
F8Eh	----	FAEh	RCREG	F CEh	TMR1L	FEEh	POSTINC0	*
F8Fh	----	FAFh	SPBRG	FCFh	TMR1H	FEFh	INDF0	*
F90h	----	F B0h	----	F D0h	RCON	F F0h	INTCON3	
F91h	----	F B1h	T3CON	F D1h	WDTCON	F F1h	INTCON2	
F92h	TRISA	F B2h	TMR3L	F D2h	LVDCON	F F2h	INTCON	
F93h	TRISB	F B3h	TMR3H	F D3h	OSCCON	F F3h	PRODL	
F94h	TRISC	F B4h	----	F D4h	----	F F4h	PRODH	
F95h	TRISD	F B5h	----	F D5h	T0CON	F F5h	TABLAT	
F96h	TRISE	F B6h	----	F D6h	TMR0L	F F6h	TBLPTRL	
F97h	----	F B7h	----	F D7h	TMR0H	F F7h	TBLPTRH	
F98h	----	F B8h	----	F D8h	STATUS	F F8h	TBLPTRU	
F99h	----	F B9h	----	F D9h	FSR2L	F F9h	PCL	
F9Ah	----	FBAh	CCP2CON	F DAh	FSR2H	F FAh	PCLATH	
F9Bh	----	FBBh	CCPR2L	F DBh	PLUSW2	*	F FBh	PCLATU	
F9Ch	----	FBCh	CCPR2H	F DCh	PREINC2	*	FFCh	STKPTR	
F9Dh	PIE1	FBDh	CCP1CON	F DDh	POSTDEC2	*	FFDh	TOSL	
F9Eh	PIR1	FBEh	CCPR1L	F DEh	POSTINC2	*	F FEh	TOSH	
F9Fh	IPR1	FBFh	CCPR1H	FDFh	INDF2	*	FFFh	TOSU	

TBLRD*+ = read then pointer point to the next

TBLRD+* = pointer point to the next then read

- A group of instructions for table processing

TBLRD*	Table Read	After Read, TBLPRTR stays the sam
TBLRD*+	Table Read with Post-inc	Reads and inc. TBLPTR
TBLRD*-	Table Read with Post-dec	Reads and dec TBLPTR
TBLRD+*	Table Read with pret-inc	Increments TBLPTR and then reads

- Use instruction TBLRD* to read the data from ROM (put it in TBLLAT)

Procedure:

1. Declare table
2. Load starting address of table into TBLPTR
3. Perform TBLRD*
4. Read the data from TABLAT

Load table pointer

- Method 1:

```
MOVLW        MYDATA_L      ; low-byte address
MOVWF        TBLPTRL       ; low-byte table pointer
MOVLW        MYDATA_H      ; high-byte address
MOVWF        TBLPTRH       ; high-byte table pointer
MOVLW        MYDATA_U      ; upper-byte address
MOVWF        TBLPTRU       ; upper-byte table pointer
```

- Method 2:
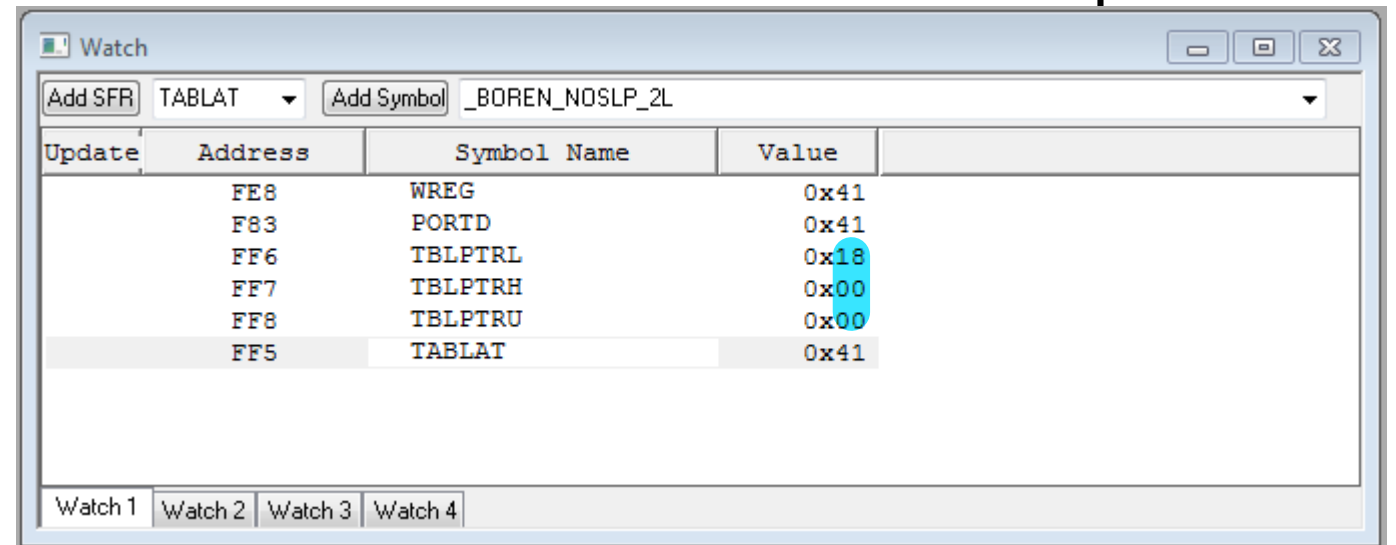
```
MOVLW        low     MYDATA
MOVWF        TBLPTRL
MOVLW        high    MYDATA
MOVWF        TBLPTRH
MOVLW        upper   MYDATA
MOVWF        TBLPTRU
```

Example

Write a program to read a byte in MYDATA (at location 18H), and then put it to PORTD.

```
Main:    ORG      0x0000
         clrf     TRISD
         movlw    0x18
         movwf    TBLPTRL
         movlw    0
         movwf    TBLPTRH
         movlw    0
         movwf    TBLPTRU
         tblrd*
         movf     TABLAT,W
         movwf    PORTD
Here:    goto     Here
         ORG      0x18
MYDATA   DB       'A'
         END
```

Or movff TABLAT PORTD

	Watch				
Add SFR	TABLAT ▾	Add Symbol	_BOREN_NOSLP_2L		▾
Update	**Address**	**Symbol Name**	**Value**		
	FE8	WREG	0x41		
	F83	PORTD	0x41		
	FF6	TBLPTRL	0x18		
	FF7	TBLPTRH	0x00		
	FF8	TBLPTRU	0x00		
	FF5	TABLAT	0x41		

Watch 1 | Watch 2 | Watch 3 | Watch 4

Example

Write a program to read a byte in MYDATA, and then put it to PORTD.

```
000000              00004  Main     ORG     0x0000
000000  6A95        00005                    CLRF     TRISD
000002  0E18        00006                    MOVLW    low MYDATA
000004  6EF6        00007                    MOVWF    TBLPTRL
000006  0E00        00008                    MOVLW    high MYDATA
000008  6EF7        00009                    MOVWF    TBLPTRH
00000A  0E00        00010                    MOVLW    upper MYDATA
00000C  6EF8        00011                    MOVWF    TBLPTRU
00000E  0008        00012                    TBLRD*
000010  50F5        00013                    MOVF     TABLAT, W
000012  6E83        00014                    MOVWF    PORTD
                    00015
000014  EF0A F000   00016             here   goto     here
                    00017
000018  0041        00018  MYDATA   DB  "A"
                    00019
                    00020             END
```

Example

Assume that ROM space starting at 250H contains "Embedded System", write a program to send all characters to PORTD one byte at a time.

```
000000            00004  Main       ORG    0x0000
000000  6A95      00005             CLRF          TRISD
000002  0E50      00006             MOVLW   low MYDATA
000004  6EF6      00007             MOVWF   TBLPTRL
000006  0E02  00 02 50  00008       MOVLW   high MYDATA
000008  6EF7      00009             MOVWF   TBLPTRH
00000A  0E00      00010             MOVLW   upper MYDATA
00000C  6EF8      00011             MOVWF   TBLPTRU
00000E  0009      00012  B7         TBLRD*+
000010  50F5      00013             MOVF          TABLAT,W
000012  E002      00014             BZ            EXIT     Break loop when 0
000014  6E83      00015             MOVWF   PORTD
000016  D7FB      00016             BRA     B7
000018  EF0C F000 00017  EXIT       GOTO    EXIT
000250            00018             ORG     0x250
000250  6D45 6562 6464  00019  MYDATA  DB "Embedded System",0
        6465 5320 7379                                    0 at the end
        6574 006D
                  00020                      END
```

What is TBLPTR when the program finished?

00 02 60

Watch (Window 1)

Add SFR | TABLAT | Add Symbol | _BOREN_NOSLP_2L

Update	Address	Symbol Name	Value	Char
	FE8	WREG	0x45	'E'
	F83	PORTD	0x45	'E'
	FF6	TBLPTRL	0x51	'Q'
	FF7	TBLPTRH	0x02	'.'
	FF8	TBLPTRU	0x00	'.'
	FF5	TABLAT	0x45	'E'

Watch 1 | Watch 2 | Watch 3 | Watch 4

Watch (Window 2)

Add SFR | TABLAT | Add Symbol | _BOREN_NOSLP_2L

Update	Address	Symbol Name	Value	Char
	FE8	WREG	0x6D	'm'
	F83	PORTD	0x6D	'm'
	FF6	TBLPTRL	0x5F	'_'
	FF7	TBLPTRH	0x02	'.'
	FF8	TBLPTRU	0x00	'.'
	FF5	TABLAT	0x6D	'm'

Watch 4

Watch (Window 3)

Add SFR | TABLAT | Add Symbol | _BOREN_NOSLP_2L

Update	Address	Symbol Name	Value	Char
	FE8	WREG	0x00	'.'
	F83	PORTD	0x6D	'm'
	FF6	TBLPTRL	0x60	'`'
	FF7	TBLPTRH	0x02	'.'
	FF8	TBLPTRU	0x00	'.'
	FF5	TABLAT	0x00	'.'

Watch 1 | Watch 2 | Watch 3 | Watch 4

Look-Up table

- access elements of a frequently used table with minimum operations

- Example: x^2

- we can use a look-up table instead of calculating the values **WHY?**

- store the function $f(x)$ in a table (RAM or ROM)

- get the base address of the table

- input x provides the displacement

- RETLW (Return from subroutine with Literal to WREG) will provide the desired look-up table element in WREG

- Before execute RETLW, we need to add a fixed value (displacement) to the PCL (low-byte of PC) to index into the look-up table. So PC = PC + displacement

  And the corresponding RETLW is at the PC + displacement

Example

Write a program to get x^2. Use look-up table instead of a multiply instruction.

```
Main:
        ORG     0x0000
        movlw   d'4'
        call    XSQR_TABLE
EXIT:   goto    EXIT
```

```
XSQR_TABLE
        MULLW   0x2
        MOVFF   PRODL, WREG
        ADDWF   PCL
        RETLW D'0'
        RETLW D'1'
        RETLW D'4'
        RETLW D'9'
        RETLW D'16'  @8
        RETLW D'25'
        RETLW D'36'
        RETLW D'49'
        RETLW D'64'
        RETLW D'81'
END
```

Since RETLW occupies two bytes, we need MULLW 02

ADDWF does not affect the whole PC

Any potential problem?

Example

Write a program to get x^2. Use look-up table and TABLAT instead of a multiply instruction.

```
        Input EQU 0
Main:        ORG    0x0000
             MOVLW      d'9'
             MOVWF      input
             call       XSQR
EXIT:    GOTO EXIT
XSQR:    MOVLW     low XSQR_TABLE
         MOVWF    TBLPTRL
         MOVLW     high XSQR_TABLE
         MOVWF    TBLPTRH
         MOVLW     upper XSQR_TABLE
         MOVWF    TBLPTRU
          MOVF       input, W
          ADDWF      TBLPTRL, F   pointer + 9
         MOVLW     0
         ADDWFC    TBLPTRH
         ADDWFC    TBLPTRU
         TBLRD*
         MOVF       TABLAT, W
           RETURN
```

```
ORG     0x0FE
XSQR_TABLE
 db D'0', D'1',D'4',D'9',
 db D'16',D'25',D'36',D'49',
 db D'64',D'81'
END
```

Look-Up table in RAM

- table can also be stored in RAM

- table elements can be changed

- use FSR (12-bit) as pointer (cover the entire 4 KB RAM space)

- use WREG as an index into the look-up table

Use "incf FSR0L, F" to increment the pointer can cause problem!

Auto-increment/decrement of FSRn for clrf instruction

Instruction	Function
clrf INDFn	After clearing fileReg pointed to by FSRn, FSRn stays the same
clrf POSTINCn	After clearing fileReg pointed to by FSRn, FSRn is incremented
clrf PREINCn	FSRn is incremented, then fileReg pointed to by FSRn is cleared (new loacation)
clrf POSTDECn	After clearing fileReg pointed to by FSRn, FSRn is decremented
clrf PLUSWn	Clears fileReg pointed to by FSRn + WREG, FSRn and WREG unchanged

The auto-increment or auto-decrement affects the entire 12 bits of FSRn
(*no effect on status register*)

Example:

MOVFF PLUSW2 , PortD

will copy data from location pointed by FSR2+WREG into Port D

Example

Write a program to get x^2. Use look-up table and FSR instead of a multiply instruction. FSR provide the base address, WREG provide displacement

```
table     equ 0
Main:     ORG    0x0000
          LFSR   0, table
          MOVLW 0
          MOVWF POSTINC0
          MOVLW d'1'
          MOVWF POSTINC0
          MOVLW d'4'
          MOVWF POSTINC0
          MOVLW d'9'
          MOVWF POSTINC0
          MOVLW d'16'
          MOVWF POSTINC0
          MOVLW d'25'
          MOVWF POSTINC0
          MOVLW d'36'
          MOVWF POSTINC0
          MOVLW d'49'
          MOVWF POSTINC0
          MOVLW d'64'
          MOVWF POSTINC0
          MOVLW d'81'
          MOVWF POSTINC0
     ;for initialization of the table in RAM
```

```
          MOVLW d'7'
          call   XSQR
EXIT: GOTO EXIT

XSQR      LFSR       0, table          setup base address
          MOVFF      PLUSW0, WREG FSR0 + W -> W
          RETURN
          END
```

Example

Write a program to copy the value 55H into RAM locations 40h to 45h using

1. register indirect addressing mode and auto-increment of FSR without a loop

2. register indirect addressing mode and auto-increment of FSR with a loop

Solution 1

```
MOVLW  55H
LFSR   0,0x40
MOVWF  POSTINC0
MOVWF  POSTINC0
MOVWF  POSTINC0
MOVWF  POSTINC0
MOVWF  POSTINC0
MOVWF  POSTINC0
```

Solution 2

```
            LFSR      0, 0x040
            MOVLW     0x6
            MOVWF     COUNT
            MOVLW     0x55
B1          MOVWF     POSTINC0
            DECF      COUNT,F
            BNZ       B1
```

Which method is better?

Example

Write a program to clear 16 RAM locations starting at location 60H using auto-increment.

```
        COUNTREG    EQU    0x10
        CNTVAL      EQU    D'16'
        MOVLW       CNTVAL
        MOVWF       COUNTREG
        LFSR        1,0x60
B3      CLRF        POSTINC1
        DECF        COUNTREG,F
        BNZ         B3
```

Example

Write a program to copy a block of 5 bytes of data from locations starting at 30H to RAM locations starting at 60H.

```
        COUNTREG    EQU     0x10
        CNTVAL      EQU     D'5'
        MOVLW       CNTVAL
        MOVWF       COUNTREG
        LFSR        0, 0x30
        LFSR        1, 0x60
B3      MOVF        POSTINC0,W
        MOVWF       POSTINC1
        DECF        COUNTREG,F
        BNZ         B3
```

Example

Assume that RAM locations 40-43H have the following hex data. Write a program to add them together and place the result in locations 06 and 07.

```
COUNTREG    EQU  0x20
L_BYTE      EQU  0x06
H_BYTE      EQU  0x07
CNTVAL      EQU  4
MOVLW       CNTVAL
MOVWF       COUNTREG
LFSR        0,0x40
CLRF        WREG
CLRF        H_BYTE
B5   ADDWF  POSTINC0, W
     BNC    OVER
     INCF   H_BYTE,F
OVER DECF   COUNTREG,F
     BNZ    B5
     MOVWF  L_BYTE
```

Address	Data
040H	7D
041H	EB
042H	C5
043H	5B

Review: bit-addressability (*refer to section 3.3.2*)

- need to access individual bits of RAM instead of the entire 8 bits

- PIC18 provides instructions that alter individual bits without altering the rest of the bits

- common bit-oriented instructions:

Instructions	Function
bsf fileReg, bit	Bit Set fileReg
bcf fileReg, bit	Bit Clear fileReg
btg fileReg, bit	Bit Toggle fileReg
btfsc fileReg, bit	Bit test fileReg, skip if clear
btfss fileReg, bit	Bit test fileReg, skip if set

Example

Write a program to add the following multi-byte BCD numbers and save the result at location 60H. 12896577 + 23647839

```
COUNTREG    EQU 0x20
CNTVAL      EQU D'4'
MOVLW       CNTVAL
MOVWF       COUNTREG
LFSR        0,0x30
LFSR        1,0x50
LFSR        2,0x60
BCF         STATUS,C
B3    MOVF      POSTINC0,W
      ADDWFC    POSTINC1,W
      DAW
      MOVWF     POSTINC2
      DECF      COUNTREG,F
      BNZ       B3
```

BCD number

Address	Data
030H	77
031H	65
032H	89
033H	12
050H	39
051H	78
052H	64
053H	23

Summary
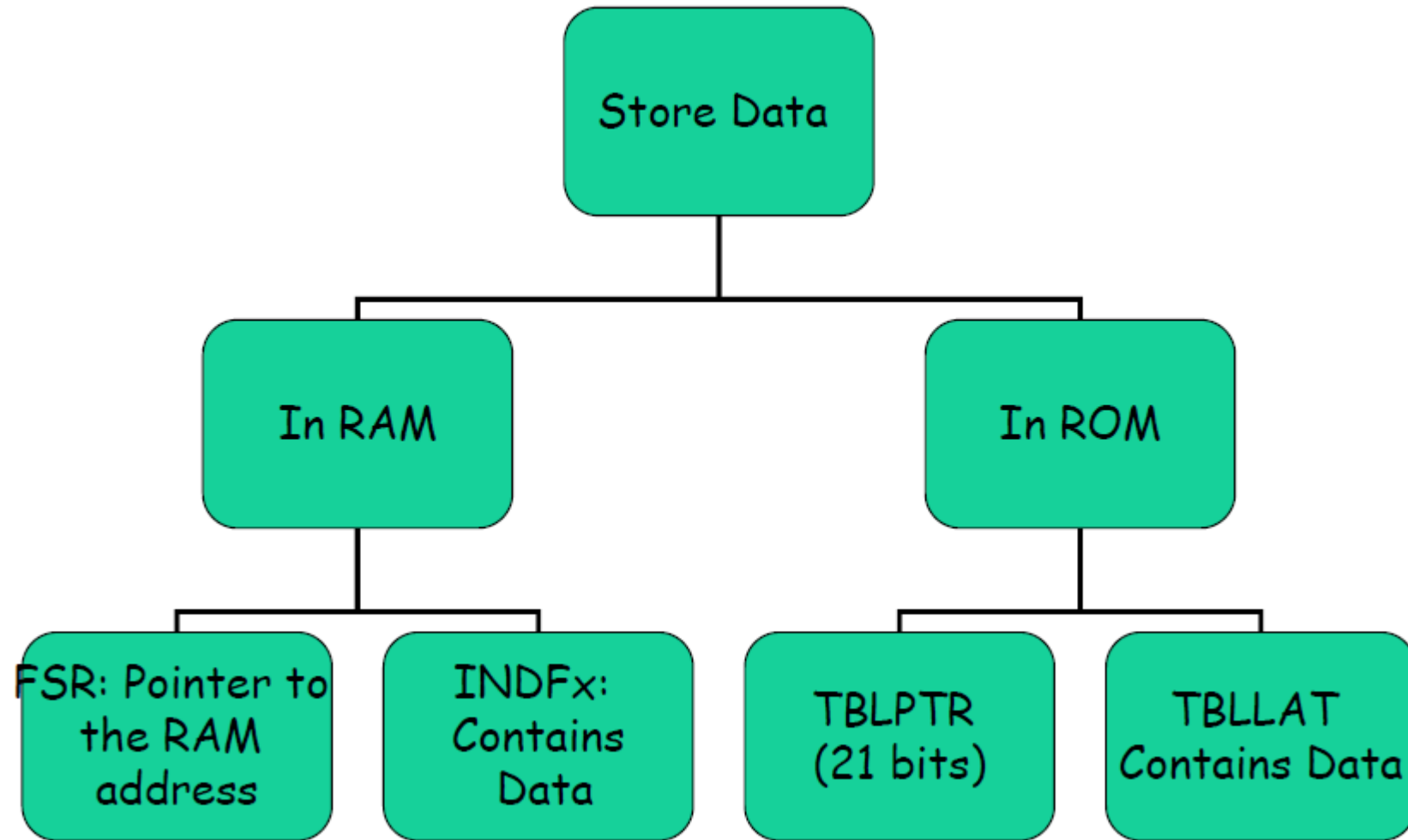
3.6.2 stack

- temporary memory storage space used during the execution of a program

- can be part of R/W memory or specially designed group of registers

- Stack Pointer (SP)
  - a register similar to the program counter, to keep track of available stack locations

- two operations on the stack：
  - PUSH：put an item onto the *top* of the stack
  - POP：remove an item from the *top* of the stack
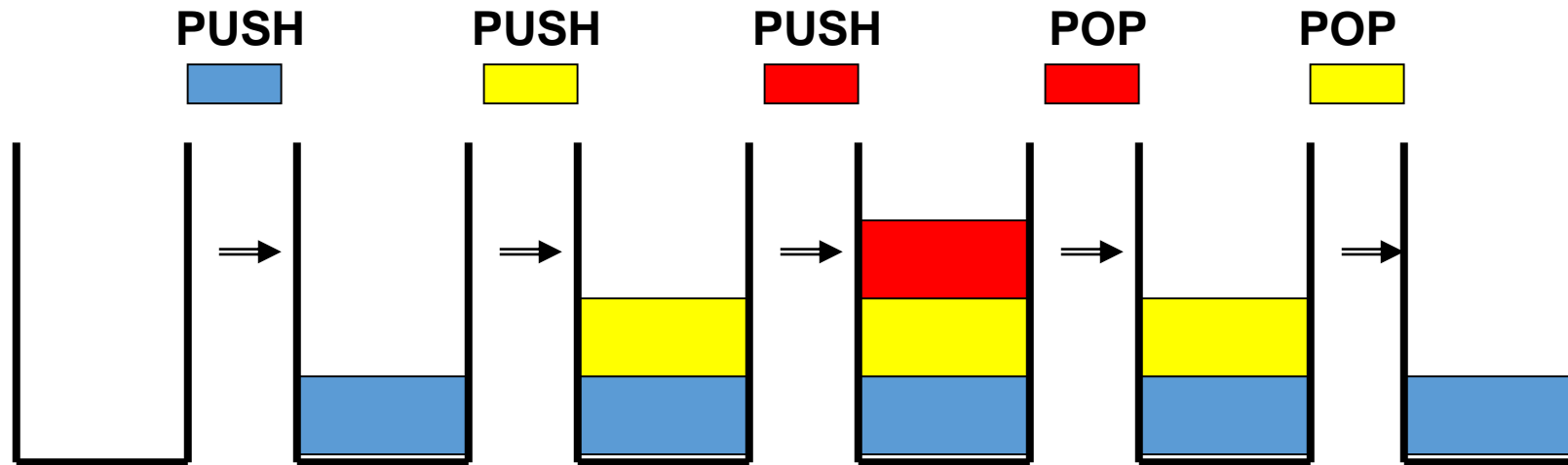
first-in-last-out (or last-in-first-out)

- Hardware Stack
  - 31 registers
  - 21-bits wide
  - Not part of program memory or data registers
- Stack Pointer (STKPTR)
  - 5-bit address
- Top of the Stack (TOS)
  - Pointed to by the stack pointer
  - Copied into three special function registers
  - TOSU (Upper), TOSH (High), and TOSL (Low)

STKPTR Register

B7	B6	B5	B4	B3	B2	B1	B0
STKOF	STKUNF	_	SP4	SP3	SP2	SP1	SP0

1=Stack Overflow
1= Stack Underflow
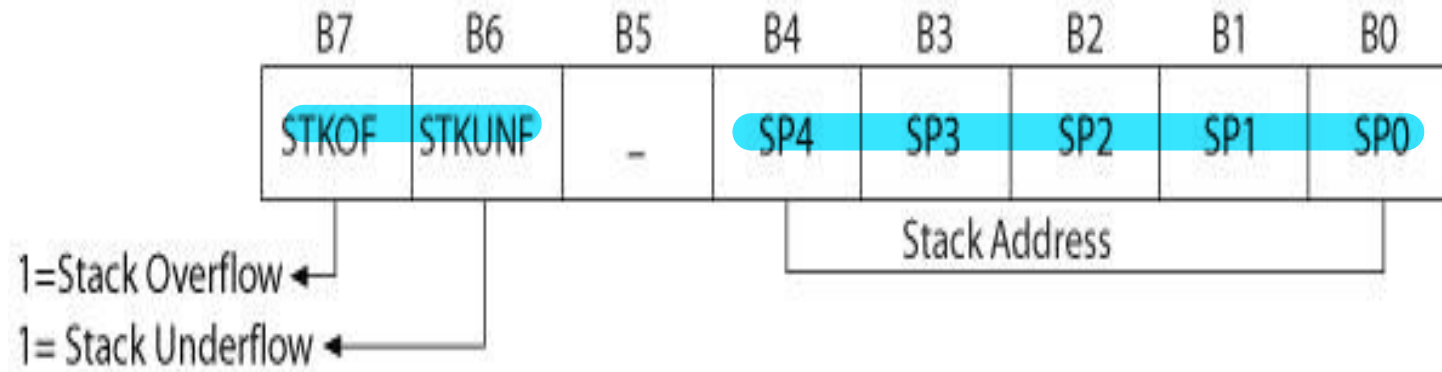
Stack Address

- 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

- STKUNF: Stack underflow
  - When the user attempts to retrieve more information than what is stored previously on the stack

Stack Instructions

- PUSH
  - Increment the memory address in the stack pointer and store the information (e.g. program counter) on the top of the stack
- POP
  - Discard the information at the top of the stack and decrement the stack pointer by one

3.6.3 subroutine call

- A group of instructions that performs a specified task
- Written independent of a main program
- Can be called multiple times to perform task by main program or by another subroutine
- Call and Return instructions used to call a subroutine and return from the subroutine

```
    LIST        P=18F4520                    ;directive to define processor
    #include <P18F4520.INC>                  ;CPU specific variable definitions
                        ORG 0
                        GOTO Main
Main:                   ORG 20H          ; start at address 0
                        MOVLW 25H                ; WREG = 25
                        CALL sub1
                        MOVLW 20H
                        CALL sub1
HERE:                   GOTO HERE                ; stay here forever
sub1:                   nop                      ; a subroutine
                        nop
                        MOVLW 0H
                        Return
                        END                      ; end of asm source file
```

CALL and RCALL

- CALL Label, s   ;Call subroutine at Label
- CALL Label, FAST ;FAST equivalent to s = 1
  - If s = 0:   Increment the stack pointer and store the return address (PC+4) on the top of the stack (TOS) and branch to the subroutine address located at Label
  - If s = 1: Also copy the contents of W, STATUS, and BSR registers in their respective shadow registers
- RCALL, n   ;Relative call to subroutine
  - Increment the stack pointer and store the return address (PC+2) on the top of the stack (TOS) and branch to the location Label within = -2048 to + 2046

- CALL: 4 bytes (long call)

  call a subroutine in anywhere

  format is similar to GOTO

- RCALL: 2 bytes (relative call)

  11-bit relative address

RETURN

- RETURN s   ;Return from subroutine
- RETURN FAST   :FAST equivalent to s = 1
  - If s = 0:   Get the return address from the stack (TOS) and place it in PC and decrement the stack pointer
  - If s = 1:   Also retrieve the contents of W, STATUS, and BSR registers from their shadow registers
- RETLW 8-bit   ;Return literal to WREG
  - Get the return address from the stack (TOS) and place it in PC and decrement the stack pointer
  - Return 8-bit literal to WREG

The Flow of Control Involving a Procedure
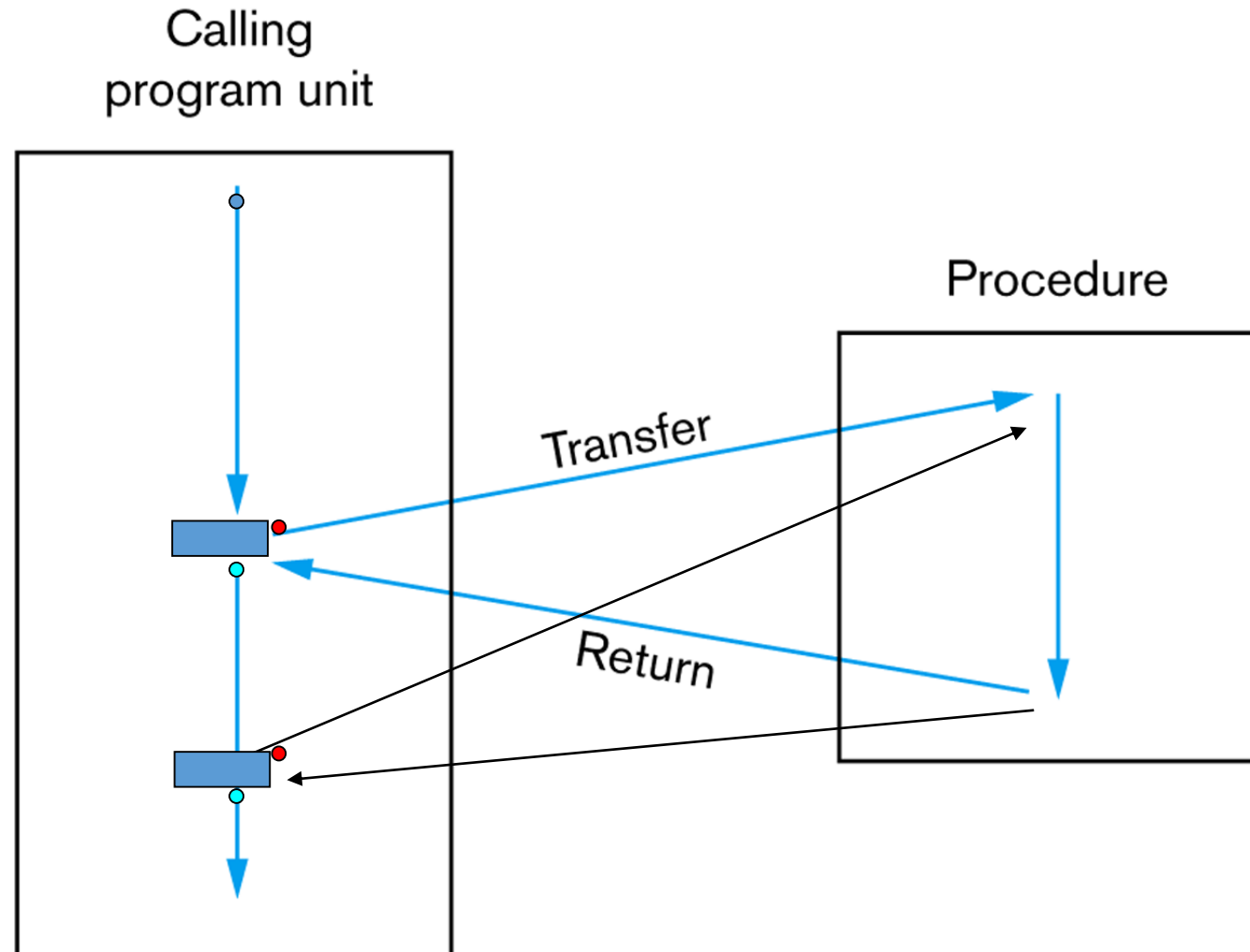
The Process of Calling a Subroutine

- After execution of the called subroutine, the CPU must know where to come back to
- The process of calling a subroutine：
  - A subroutine is called by CALL instruction
  - The CPU pushes the PC onto the stack (in PIC18 it is a hardware stack)
  - The CPU copies the target address to the PC
  - The CPU fetches instructions from the new location
  - When the instruction RETURN is fetched, the subroutine ends
  - The CPU pops the return address from the stack
  - The CPU copies the return address to the PC
  - The CPU fetches instructions from the new location

Sample

```
;MAIN program calling subroutines
        ORG   0
MAIN:
          .
          .
          .
          CALL      SUBR_1
          .
          CALL      SUBR_2
          .
          .
HERE:     GOTO      HERE
;————end of MAIN

SUBR_1:   ....
          ....
          RETURN
;————end of subroutine 1

SUBR_2:   ....
```

Sample

```
;MAIN program calling subroutines

MYREG EQU    0x8                          RETURN
PORTB EQU    0xF8                         END
      ORG 0

BACK: MOVLW 0x55
      MOVWF PORTB
      CALL  DELAY
      MOVLW 0xAA
      MOVWF PORTB
      CALL  DELAY
      GOTO  BACK
DELAY:
      MOVLW 0xFF
      MOVWF MYREG
AGAIN:
      NOP
      NOP
      DECF  MYREG, F
      BNZ   AGAIN
```

```
Main Program                                          Subroutine

0020 0EFE        START: MOVLW   B'11111110'   DELAY50MC:
0022 6E94               MOVWF   TRISC          0040   0EA6   MOVLW   D'166'
0024 6E01               MOVWF   REG1           0042   6E10   MOVWF   REG10
0026 C001 FF82   ONOFF: MOVFF   REG1,PORTC     0044   0610   DECF    REG10,1
002A EC20 F000          CALL    DELAY50MC      0046   E1FE   BNZ     LOOP1
002E 1E01               COMF    REG1,1         0048   0012   RETURN
0030 D7FA               BRA     ONOFF
```

the return address is 002E

ROM address	Code	Line No.		
000000		00003		ORG 0
000000	EF10 F000	00004		GOTO Main
000020		00005	Main:	ORG 20H; start at address 0
000020	0E25	00006		MOVLW 25H ; WREG = 25
000022	EC18 F000	00007		CALL sub1
000026	0E20	00008		MOVLW 20H
000028	EC18 F000	00009		CALL sub1
00002C	EF16 F000	00010	HERE:	GOTO HERE ;stay here forever
000030	0000	00011	sub1:	nop ; a subroutine
000032	0000	00012		nop
000034	0E00	00013		MOVLW 0H
000036	0012	00014		Return
		00015		END ; end of asm source file

- In modern computers, we need a stack to store the return address during subroutine call

- Besides, we need to save the register values that will be modified by the subroutine

- In the PIC18, we only have a hardware stack for storing the return address (*that is why stack is 21-bit wide*!)

- How to solve this problem?

```
ascii_l   EQU 0x0 ;input parameter
ascii_h   EQU 0x1 ;input parameter
out_bcd   EQU 0x2 ;output value
local_var EQU 0x3
```

Main program

… …
… …
CALL SUB1
… …
… …
CALL SUB1
… …

SUB1

  … …
    this routine may modify
    WREG, STATUS, BSR
  … …
RETURN

Use some locations to store the affected registers

```
ascii_l    EQU 0x0  ;input parameter
ascii_h    EQU 0x1  ;input parameter
out_bcd    EQU 0x2  ;output value
local_var EQU 0x3
W_TEMP_SUB1       EQU 0x4
Flag_TEMP_SUB1 EQU 0x5
BSR_TEMP_SUB1   EQU 0x6
```

Main program

... ...
... ...
CALL SUB1
... ...
... ...
CALL SUB1
... ...

```
SUB1
  MOVWF W_TEMP_SUB1
  MOVFF STATUS, STATUS_TEMP_SUB1
  MOVFF BSR, BSR_TEMP_SUB1
... ...
```
this routine may modify
WREG, STATUS, BSR
 ... ...
```
  MOVFF BSR_TEMP_SUB1, BSR
  MOVFF STATUS_TEMP_SUB1, STATUS
  MOVF W_TEMP_SUB1,W
RETURN
```

Waste resource if we have many routines.
Also, the routines cannot be re-entered. Why?

We can use FSR to implement a user stack.
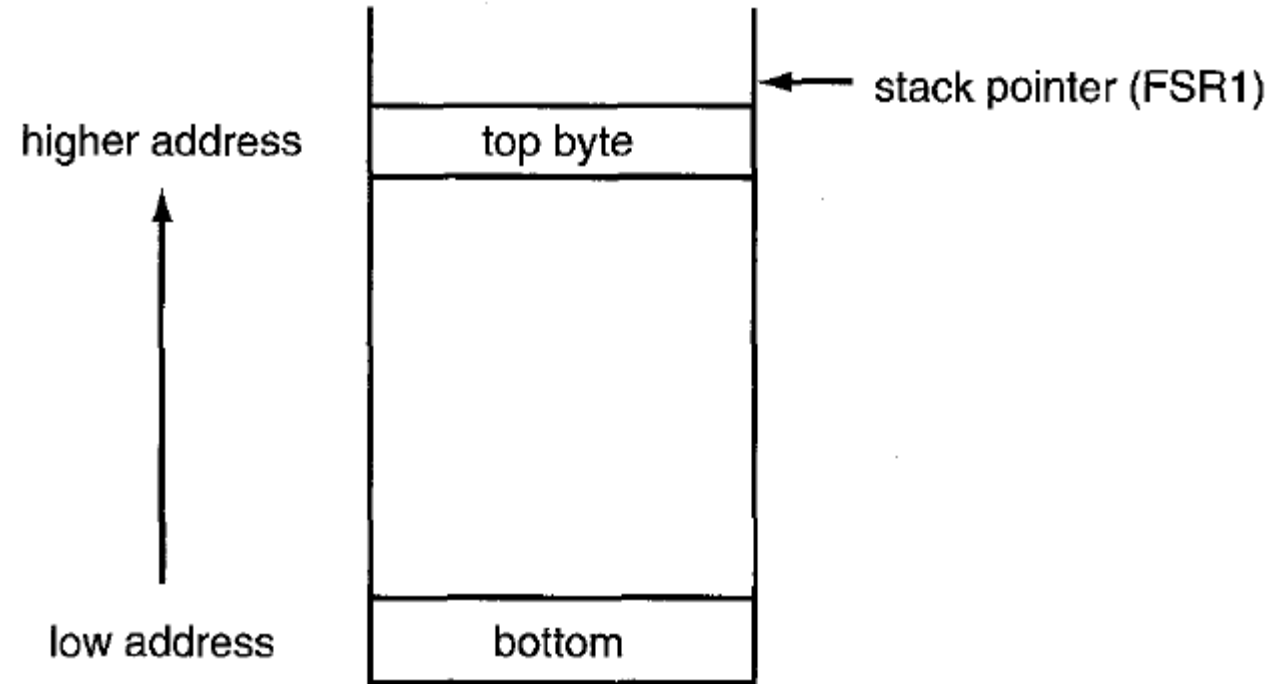
Use LFSR to define a stack

```
LFSR FSR1, 0x500
```

The following instruction will push the STATUS register onto the stack:

```
MOVFF STATUS,PREINC1
```

The following instruction sequence will pull the top byte of the stack onto the STATUS register:

```
MOVFF POSTDEC1,STATUS
```

```
stack    equ 0x200
R0       equ 0
R1       equ 1
R2       equ 2
Main    ORG      0x0000
        LFSR  0, stack              ; define a user stack
        MOVLW 0x00
        MOVWF R0
        MOVLW 0x10
        MOVWF R1
        MOVLW 0x20
        MOVWF R2
        MOVLW 0x99
        MOVFF R0,   PREINC0    ; push R0 to stack    FSR0=0201
        MOVFF R1,   PREINC0    ; push R1 to stack    FSR0=0202
        MOVFF R2,   PREINC0    ; push R2 to stack    FSR0=0203
        MOVFF WREG, PREINC0    ; push WREG to stack FSR0=0204

        MOVLW 0x71             ; Now program change W, R0, R1, R2
        MOVWF R0               ;
        MOVWF R1               ;
        MOVWF R2               ;
                              ; restore the value from stack
        MOVFF POSTDEC0, WREG  ; FSR0=0203
        MOVFF POSTDEC0, R2    ; FSR0=0202
        MOVFF POSTDEC0, R1    ; FSR0=0201
        MOVFF POSTDEC0, R0    ; FSR0=0200

        EXIT GOTO EXIT

    END
```

Use stack to store the affected registers

```
ascii_l   EQU 0x0 ;input parameter
ascii_h   EQU 0x1 ;input parameter
out_bcd   EQU 0x2 ;output value
local_var EQU 0x3
stack     EQU 0x200; define a stack
Main program
    LFSR  0, stack
… …
… …
CALL SUB1
… …
… …
CALL SUB1
… …
```

```
SUB1
  MOVFF  WREG,  PREINC0
  MOVFF  STATUS,  PREINC0
  MOVFF  BSR,  PREINC0
  … …
  this routine may modify
  WREG, STATUS, BSR
  … …
  MOVFF  POSTDEC0,  BSR
  MOVFF  POSTDEC0,  STATUS
  MOVFF  POSTDEC0,WREG
RETURN
```

In general, modern computer systems use stack to pass parameters and to handle local variables.

Parameter passing

- pass parameters to and from a subroutine

- subroutine can process a different set of data each time they are called, e.g. communication application

Example:
a subroutine that processes two BCD digits (packed BCD)

In high level language, we have
pass by value - variables are referred to directly
pass by address (**pointer**) – a pointer to a value or a pointer to a data structure is passed

The procedure is complicated in assembly language

Parameter passing by value (use fixed file register locations)

Registers
advantage: fast,
disadvantages: limited number of parameters

```
ascii_l        EQU 0x0
ascii_h        EQU 0x1
out_bcd        EQU 0x2
local_var      EQU 0x3


               ORG 0
               MOVLW    A'4'
               MOVWF    ascii_h
               MOVLW    A'7'
               MOVWF    ascii_l
               CALL  ASCTOBCD
               MOVLW    A'3'
               MOVWF    ascii_h
               MOVLW    A'9'
               MOVWF    ascii_l
               CALL  ASCTOBCD
here               goto here
```

```
ASCTOBCD:
       MOVF     ascii_h,W
       ANDLW    0x0F
       MOVWF    local_var
       SWAPF    local_var, F
       MOVF     ascii_l,W
       ANDLW    0x0F
       IORWF    local_var, F
       MOVFF    local_var, out_bcd
       return
```

Summary

♦ look-up table and table processing

♦ stack

♦ subroutine