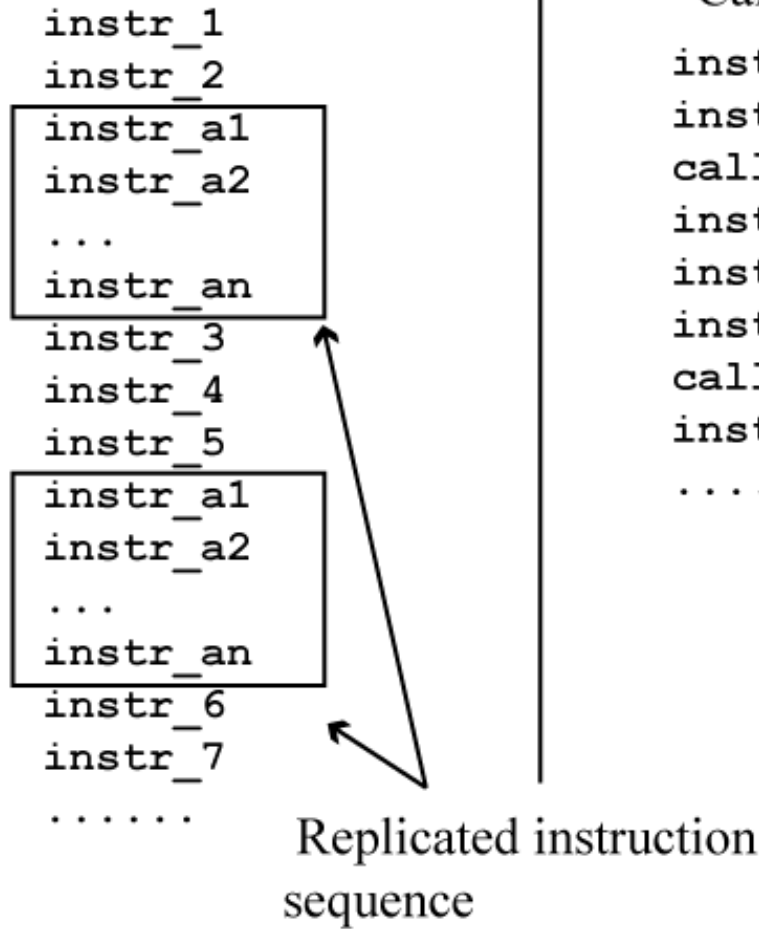


Subroutines

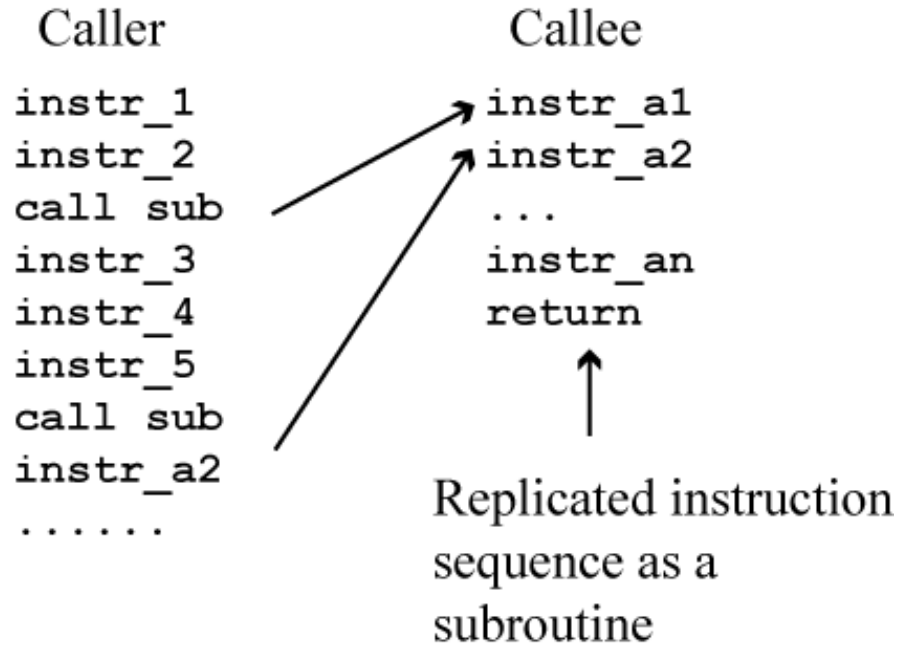
- A **subroutine** is a block of code that is **called** from different places from within a main program or other subroutines.
 - Saves code space in that the subroutine code does not have to be repeated in the program areas that need it; only the code for the *subroutine call* is repeated.
- A subroutine can have zero or more **parameters** that control its operation
- A subroutine may need to use **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.

Why Subroutines?

Without Subroutines



With Subroutines



C Subroutines (aka. Function)

General form of a C subroutine is:

```
(return_type) subname (parm list)
{
    local_variable_decl;
    subroutine_body;
    return(return_value);
}
```

The statement `i << j` is valid in C, this is only used for example purposes.

```
vlshift Subroutine
// variable left shift
unsigned char vlshift(unsigned char v,
unsigned char amt)
{
    while (amt) {
        v = v << 1;
        amt--;
    }
    return(v);
}

main(void) {
    unsigned char i,j,k;

    i=0x24; j = 2;
    k = vlshift(i,j);
    printf(
        "i=0x%x, shift amount: %d,result: 0x%x\n",
        i,j,k);
}
```

parameter list: gives types and names

subroutine body

subroutine return

main program

subroutine call

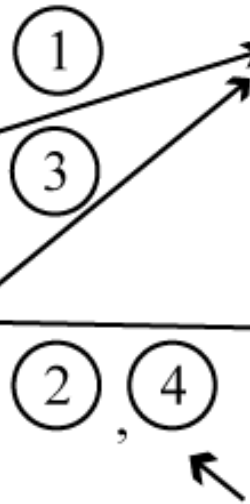
Implementing Subroutine call/return with *goto*

Calling Program

```
instr1
instr2
C1
goto subA
R1
instr3
instr4
C2
goto subA
R2
instr5
instr6
..rest of program..
```

Subroutine A

```
instr1
instr2
.....
instrN-1
instrN
goto R1 ; ;return
```



The second return is to the wrong place! Should return to R2, not R1.

Need specialized instructions to implement subroutine call/return that have more capability than *goto* as we need to remember where to return to after a subroutine call.

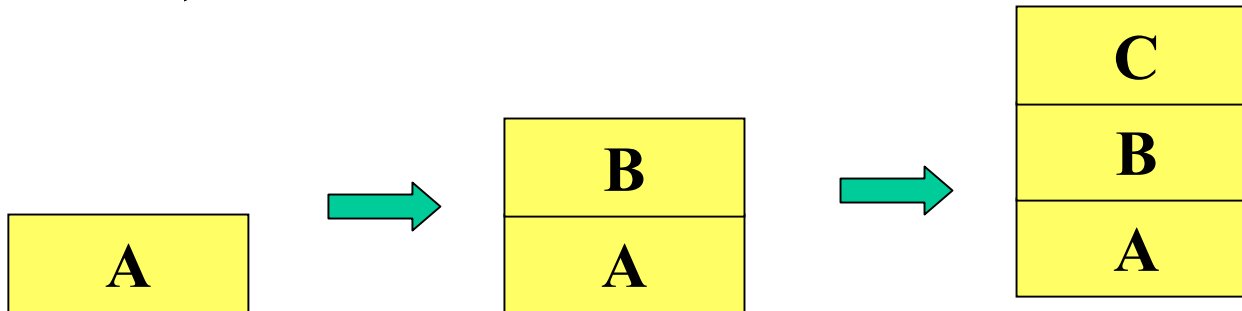
The Stack

- In μ Ps, the **stack** is a memory area intended for storing temporary values.
- Data in the stack is usually accessed by a special register called a **stack pointer**.
- In the PIC18Fxx2, 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 on subroutine exit.
 - On the PIC18Fxx2, the return address is PC+4, if PC is the location of the *call* instruction (PC is the location of the *call* instruction). The return address is PC+2 if it is a *rcall* instruction.

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.

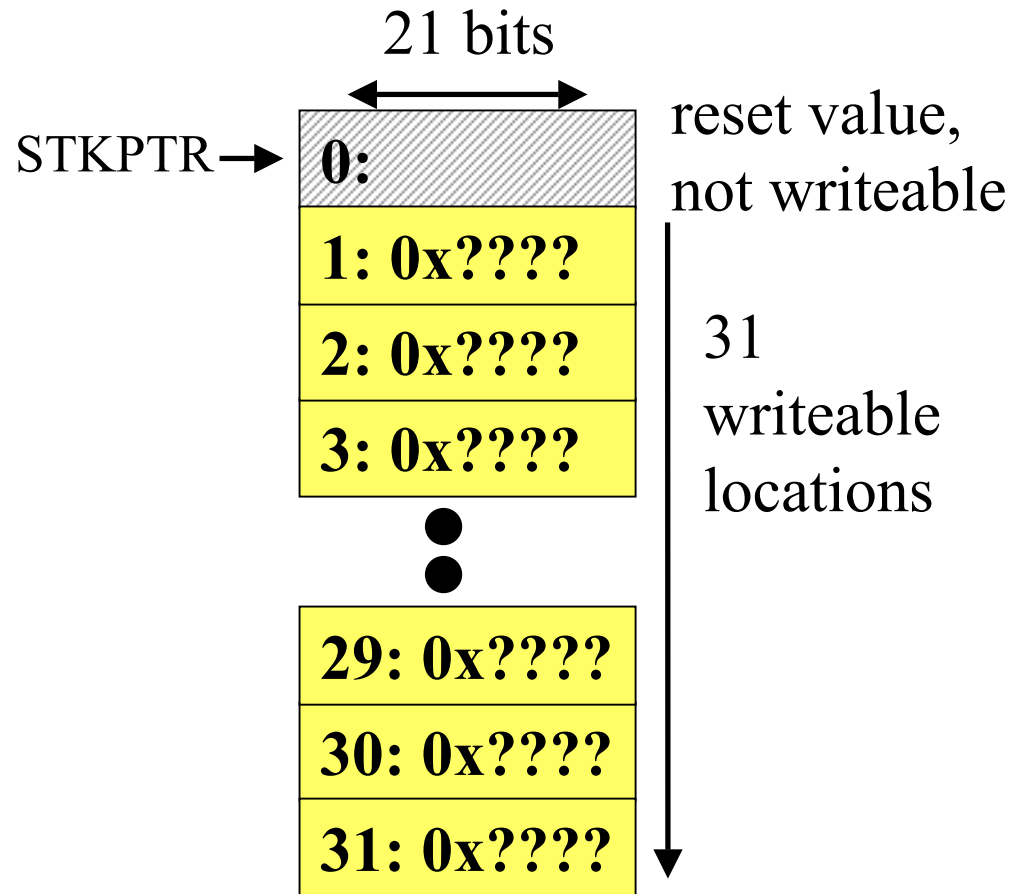
The PIC18 Stack

The PIC18 stack has limited capability compared to other μ Ps. It resides within its on 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 at location [STKPTR].
STKPTR++; [STKPTR] \leftarrow nPC

A return instruction pops the PC off the stack. A **pop** means read [STKPTR] and store to the PC, then decrement STKPTR
(PC \leftarrow [STKPTR], STKPTR--)



call Instruction

B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0

call k

1	1	1	0	1	1	0	s	k	k	k	k	k	k	k	k_0
1	1	1	1	k_{19}	k	k	k	k	k	k	k	k	k	k	k

call (call subroutine at location k) : Push PC of next instruction (nPC = PC+4) onto stack.

If $s = 1$, push W, STATUS, BSR registers into *shadow registers* (aka, the *fast register stack*). By default ($s = 0$).

Then do $PC[20:1] \leftarrow k$

A *call* saves the return address on the stack so that it knows where to return. The shadow registers are really only useful for interrupts; will discuss them when interrupts are covered.

rcall, return, retlw Instructions

	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B
	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
<i>rcall</i> k	1	1	0	1	1	n	n	n	n	n	n	n	n	n	n	n
<i>return</i>	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	<i>s</i>
<i>retlw</i> k	0	0	0	0	1	1	0	0	k	k	k	k	k	k	k	k

rcall (relative call) – pushes PC+2 (nPC) on stack, then does a branch always to subroutine, branch offset is 11 bits (-1024 to +1023). Advantage over *call* is that it only takes one word.

return (ret from subroutine): PC ← pop top-of-stack
 if *s* = 1, restore W, STAUS, BSR from shadow registers

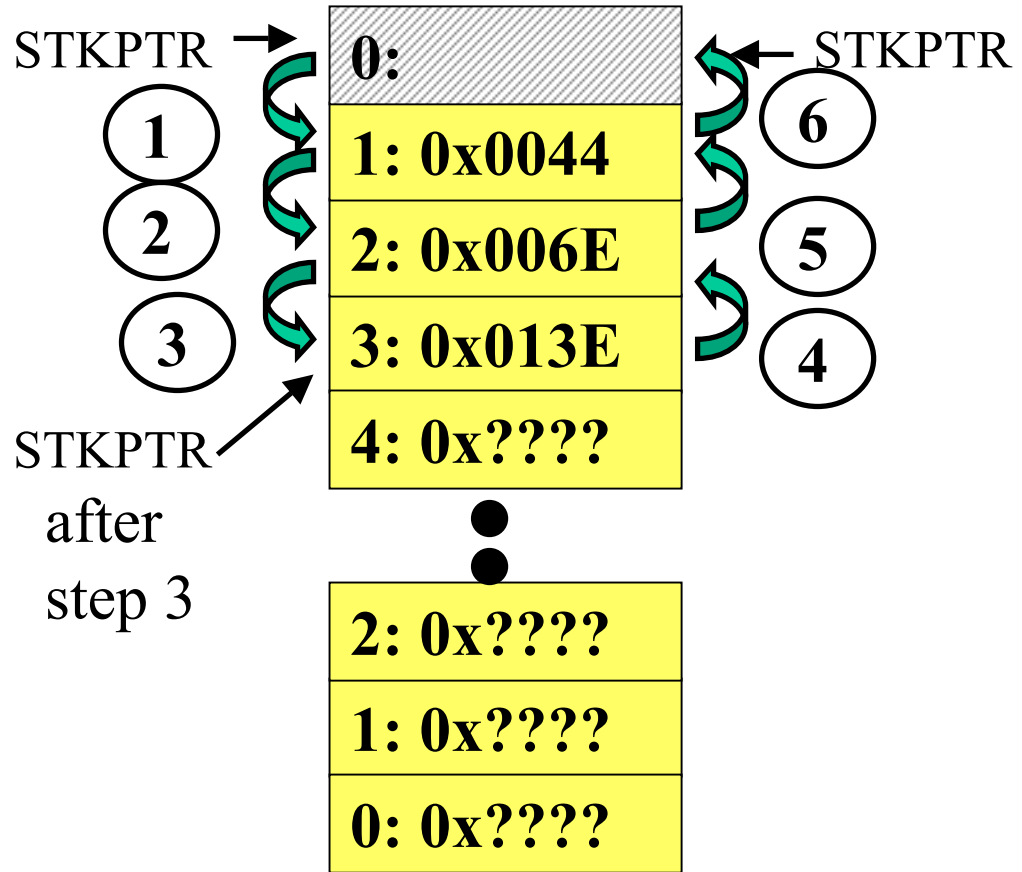
retlw (return with literal in w): w ← k, PC ← pop top-of-stack

Call/Return Example

after
step 6

```

main
  ....
  0x0040  call  Sub_A  (1)
  ....
Sub_A
  ....
  0x006A  call  Sub_B  (2)
  ....
  return  (6)
Sub_B
  ....
  0x13C  rcall Sub_C  (3)
  return  (5)
Sub_C
  ....
  return  (4)
  
```



For call, $nPC = PC + 4$.

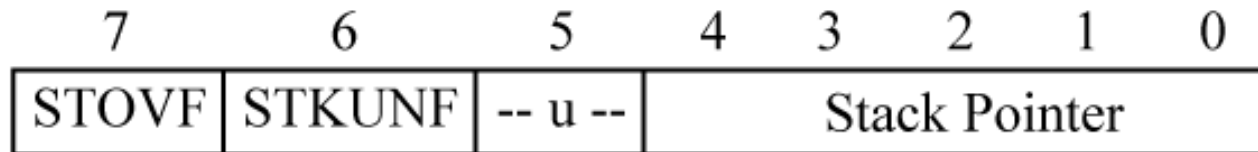
For rcall, $nPC = PC + 2$

Stack Overflow/Underflow

- Stack **overflows** on the 32nd *call* instruction without a return
 - By default, the processor self-resets itself on stack overflow. After reset, can check a status bit (STKFUL) to determine if stack overflow caused reset.
 - Can configure the processor to simply freeze the stack, and set an error bit when the stack overflows.
- Stack **underflow** occurs if attempt to do a *return* while $STKPTR = 0$.
 - In this case, PC gets set to 0x0 (which is the reset vector), and an error bit is set (STKUNF) – so reset code can determine if stack underflow occurred.

STKPTR Register

STKPTR Register



STOVF : stack overflow (set to 1 on overflow)

STKUNF: stack underflow (set to 1 on underflow)

-- u -- : unimplemented

Stack Pointer (SP): 5-bit stack pointer, cleared to 0 on reset

STOVF, STKUNF can only be cleared by user or by a power on reset

Lower 5-
bits used to
access stack
locations

Upper two bits are the
stack overflow and
underflow status bits.

Accessing the Stack via a Program

- The value on the top of the stack can be accessed by three registers: TOSU (upper, bits 20-16), TOSH (high, bits 15-8), TOSL (lower, bits 7-0).
- PUSH and POP instructions can be used to push the current PC onto the stack.
- This method of accessing the stack values is extremely primitive, and very difficult to use effectively.
 - We will ignore this capability.

vlshift Function in Assembly Language

In C

```
// variable left shift
unsigned char vlshift(
unsigned char v,
unsigned char amt)
{
    while (amt) {
        v = v << 1;
        amt--;
    }
    return(v);
}
```

In Assembly

```
;Parameter space for vlshift
CBLOCK 0x040
    v, amt ← Static allocation for
                parameters
ENDC
;; return value in w
vlshift
    movf    amt,f
vlshift_loop ←
    bz      vl_return    ;amt=0?
    bcf     STATUS,C
    rlc     v,f           ; v = v << 1
    decf    amt,f        ; amt--
    bra     vlshift_loop
    vl_return
    return
```

Static allocation is used for subroutine parameters which means that fixed memory locations are used for these locations. This has low instruction overhead, but recursive functions calls cannot be made.

Subroutine call to *vlshift*

In C

```
main(void){  
  unsigned char i, j, k;
```

```
  i=0x24;  
  j = 2;
```

```
  k = vlshift(i, j);
```

```
}
```

In Assembly

```
CBLOCK 0x0
```

```
  i, j, k
```

```
ENDC
```

```
  org 0
```

```
  goto main
```

```
  org 0x200
```

```
main
```

```
; initialize main program variables
```

```
  movlw 0x24
```

```
  movwf i ; i = 0x24
```

```
  movlw 0x2
```

```
  movwf j ; j = 2
```

```
;; setup subroutine parms
```

```
  movff i, v
```

```
  movff j, amt
```

```
  call vlshift
```

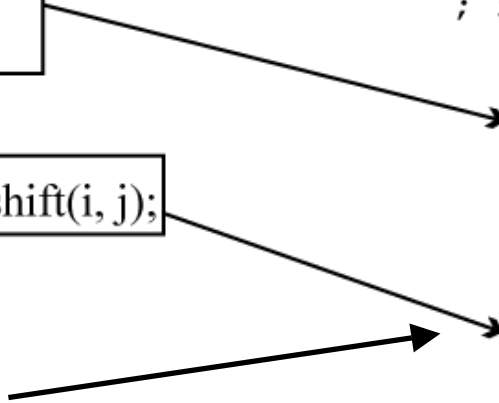
```
  movff v, k
```

```
; k = vlshift(v, amt);
```

```
here
```

```
  goto here
```

← Static allocation for
main variables



Copy *i, j* to *v, amt*
paramters before call.

After call, copy new *v*
to *k*.

Back to Parameter Passing

- The *vlshift.asm* program used a *static* memory area for parameters
 - *static* means that the location for parameters is fixed
- This is an easy method to understand and has low instruction overhead to implement, but means that subroutines cannot be called *recursively*.
 - A subroutine cannot be called from within itself (this is because the static memory area for parameters is already in use!!!)
 - If a subroutine is interrupted, then the subroutine cannot be called from the interrupt service routine.
 - We will discuss more sophisticated parameter passing methods later.

Arrays and Pointers in C

- A pointer variable is a variable that contains the address of another variable.
- An array is a collection of like elements, such as an array of integers, array of characters, etc.
- One use of pointer variables in *C* is to pass the address of an array to a function, instead of passing all of the elements of the array to function.
 - Only have to pass one element instead of multiple elements!

Arrays and Pointers in C (example)

```
(1) char sa[10], sb[] = "Hello";
(2) char *ptra, *ptrb; //declare these as pointer char types

(3) ptra = sa; // ptra is assigned starting address of sa
(4) ptra = &sa[0]; // same as previous line

(5) sa[0] = 0x20; // assign the value 0x20 to element 0
(6) *ptra = 0x20; // same as previous line

(7) sa[2] = 0x45; // assign the value 0x20 to element 2
(8) *(ptra+2) = 0x45 // same as previous line
```

*ptra refers to the element that ptra is pointing to!

& is the “address-of” operator, takes the address of a variable.

Arrays and Pointers in C (example cont.)

```
(1)  char sa[10], sb[] = "Hello";
(2)  char *ptr_a, *ptr_b;

(9)  sb[0] = sa[0];    // copy sa[0] to element sb[0]
(10) sb[0] = *ptr_a;   // same as previous line, since ptr_a points
                        // points to sa[0] from a previous statement

(11) ptr_a++; ptr_a++; // ptr_a now points to sa[2]
(12) sb[0] = *ptr_a;   // sa[2] copied into sb[0]

(13) ptr_b = sb;      // ptr_b is assigned starting address of sb
(14) sa[2] = sb[1]    // copy element 1 of sb to element 2 of sa

(15) *(ptr_a) = *(ptr_b+1) // same as previous line
(16) ptr_a = ptr_b+1;    // ptr_a now points to element sb[1]

(17) do_my_sub(sb);     // pass address of sb to subroutine
(18) do_my_sub(ptr_b);  // same as previous line
```

*char** Pointer Example with Mem addresses

C Code	Location	Contents (values in Hex)
<code>char sa[10];</code>	0150	00 00 00 00 00 00 00 00 00 00 00 sa
<code>char sb[]="Hello"</code>	015A	48 65 6C 6C 6F 00 sb
<code>char *ptra,*ptrb;</code>	0160	00 00 ptra
	0162	00 00 ptrb

<code>ptra = sa;</code>		
<code>ptra = &sa[0];</code>	→ 0160	50 01 ptra = 0x150, value of ptra is stored at 0x160 in little endian order
<code>sa[0] = 0x20;</code>		
<code>*ptra = 0x20;</code>	→ 0150	20 ptra points at location 0x150, *ptra modifies contents of 0x150
<code>sa[2] = 0x45;</code>		
<code>*(ptra+2) = 0x45;</code>	→ 0150	20 00 45 ptra+2 is location 0x152, new value is 0x45
<code>sb[0] = sa[0];</code>		
<code>sb[0]= *ptra;</code>	→ 015A	20 contents of location 0x150 (0x20) copied to location 0x15A (sb[0])
<code>ptra++;</code>	→ 0160	51 01 ptra is now 0x151
<code>ptra++;</code>	→ 0160	52 01 ptra is now 0x152

Pointer arithmetic

```
char *ptr_a;
```

```
int *ptr_b;
```

Pointer arithmetic of $ptr + n$ is performed as

$$ptr + \text{sizeof}(\text{element}) * n$$

`ptr_a++;` ← increment by 1 since a *char* is one byte.

`ptr_b++;` ← increment by 2 since a *int* is two bytes.

`ptr_a = ptr_a + 2;` ← add $2 * 1$ since a *char* is one byte.

`ptr_b = ptr_b + 2;` ← add $2 * 2$ since a *int* is two bytes.

Pointer Example with *int* data

C Code	Location	Contents (values in Hex)								
<code>signed int sa[10];</code>	0150	<table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td>00</td><td>00</td><td>00</td><td>00</td><td>00</td><td>00</td><td>00</td><td>00</td></tr></table>	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00			
<code>signed int sb[]= {</code>	0158	<table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td>00</td><td>00</td><td>00</td><td>00</td><td>00</td><td>00</td><td>00</td><td>00</td></tr></table> <code>sa</code>	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00			
<code> -20,1000,-546,</code>	0160	<table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td>00</td><td>00</td><td>00</td><td>00</td></tr></table>	00	00	00	00				
00	00	00	00							
<code> 23444}</code>	0164	<table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td>EC</td><td>FF</td><td>E8</td><td>03</td><td>DE</td><td>FD</td><td>94</td><td>5B</td></tr></table> <code>sb</code>	EC	FF	E8	03	DE	FD	94	5B
EC	FF	E8	03	DE	FD	94	5B			
<code>signed int *ptrA;</code>	016C	<table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td>00</td><td>00</td></tr></table> <code>ptrA</code>	00	00						
00	00									
<code>signed int *ptrB;</code>	016E	<table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td>00</td><td>00</td></tr></table> <code>ptrB</code>	00	00						
00	00									

<code>ptrA = sa;</code>	→ 016C	<table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td>50</td><td>01</td></tr></table> <code>ptrA</code> is now 0x150	50	01						
50	01									
<code>ptrB = sb;</code>	→ 016E	<table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td>64</td><td>01</td></tr></table> <code>ptrB</code> is now 0x164	64	01						
64	01									
<code>ptrB++;</code>	→ 016E	<table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td>66</td><td>01</td></tr></table> <code>ptrB</code> is now 0x166, note that it increased by 2 because each element size is 2 bytes; <code>ptrB</code> points at <code>sb[1]</code>	66	01						
66	01									
<code>*ptrA = *ptrB;</code>	→ 0150	<table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td>E8</td><td>03</td></tr></table> copy integer at location 0x166 (<code>ptrB</code>) to location 0x150 (<code>*ptrA</code>). This is the same as <code>sa[0] = sb[1]</code>	E8	03						
E8	03									
<code>* (ptrA+3) = * (ptrB+2);</code>	→ 0150	<table border="0" style="display: inline-table; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;"><code>sa[0]</code></td> <td style="padding-right: 10px;"><code>sa[1]</code></td> <td style="padding-right: 10px;"><code>sa[2]</code></td> <td style="padding-right: 10px;"><code>sa[3]</code></td> </tr> <tr> <td style="border: 1px solid black; padding: 2px;"><code>E8 03</code></td> <td style="border: 1px solid black; padding: 2px;"><code>00 00</code></td> <td style="border: 1px solid black; padding: 2px;"><code>00 00</code></td> <td style="border: 1px solid black; padding: 2px;"><code>94 5B</code></td> </tr> </table> <code>ptrA+3</code> refers to location <code>sa[3]</code> (0x156) as <code>ptrA</code> is <code>sa</code> (0x150) <code>ptrB+2</code> refers to location <code>sb[3]</code> (0x16A) as <code>ptrB</code> is <code>&sb[1]</code> (0x166)	<code>sa[0]</code>	<code>sa[1]</code>	<code>sa[2]</code>	<code>sa[3]</code>	<code>E8 03</code>	<code>00 00</code>	<code>00 00</code>	<code>94 5B</code>
<code>sa[0]</code>	<code>sa[1]</code>	<code>sa[2]</code>	<code>sa[3]</code>							
<code>E8 03</code>	<code>00 00</code>	<code>00 00</code>	<code>94 5B</code>							

Indirect Addressing and Pointer registers

The stack pointer is an example of a **pointer register**. A pointer register contains the address of data that is to be accessed.

The data is retrieved or stored using the pointer register (data is accessed **indirectly** via the pointer register). The address of the data must first be loaded into the pointer register before using it to access the data.

The previous addressing method we used is called **direct addressing** because the address is specified directly in the instruction:

```
movf 0x20, w ; w ← (0x20)
```

The address 0x20 is encoded directly in the instruction. This instruction will **always** access location 0x20.

PIC18 Indirect Addressing

The PIC18 has three 12-bit registers named FSRx used for holding pointers that contain data memory addresses: FSR0, FSR1, FSR2

A FSRx register is modified by accessing the high/low register bytes as FSRxH, FSRxL

	High byte	Low Byte
FSR0	FSR0H	FSR0L
FSR1	FSR1H	FSR1L
FSR2	FSR2H	FSR2L

To access the *memory contents* referenced by a FSRx register, the corresponding INDFx register is used (i.e, INDF0 is paired with the FSR0 register). The memory location pointed to by FSRx is accessed **INDirectly** through register INDFx.

Using FSR_n/INDF_n to implement C pointers

In C	In Assembly
<pre>char sa[5]; char sb[10]; char *ptr_a, *ptr_b;</pre>	<pre>CBLOCK 0x100 sa:5, sb:10 ;sa is 0x100, sb is 0x105 ENDC ; FSR0 used for ptr_a, FSR1 for ptr_b</pre>
<pre>ptr_a = sa;</pre>	<pre>movlw low sa movwf FSR0L ;init FSR0L movlw high sa movwf FSR0H ;FSR0 = 0x0100</pre>
<pre>*ptr_a = 0x30;</pre>	<pre>movlw 0x30 movwf INDF0 ;(FSR0) ← 0x30</pre>
<pre>ptr_b = sb;</pre>	<pre>lfsr FSR1, sb ;FSR1 = 0x105</pre>
<pre>*ptr_b = *ptr_a;</pre>	<pre>movff INDF0, INDF1 ;(FSR1) ← ((FSR0))</pre>

lfsr can be used to load a 12-bit literal directly into an FSR_n register.

PostInc/PostDec/PreInc/Plusw

Incrementing and decrementing the FSRx registers are a common operation.

The FSRx registers can be *automatically* incremented or decremented by using one of the registers below instead of the INDFx register.

```
movf    POSTDEC0, w    ; w ← [FSR0], FSR0--  
movf    POSTINC0, w    ; w ← [FSR0], FSR0++  
movf    PREINC0, w     ; FSR0++, w ← [FSR0]
```

The above registers are useful for stepping linearly through data or implementing stacks. The register below is useful for implementing C array operations:

```
movf    PLUSW0, w      ; w ← [FSR0+w]
```

POSTINCn usage

In C

```
char sa[5];
char sb[10];
char *ptra,*ptrb;
unsigned char i;

ptra = sa;
ptrb = sb;
for (i=0; i < 5; i++) {
    *ptra = *ptrb;
    ptra++;
    ptrb++;
}
```

In Assembly

```
CBLOCK 0x100
    sa:5,sb:0xa,i ;sa is 0x100, sb is 0x105
ENDC

    movlb 1 ;select bank 1
    lfsr FSR0, sa ;init FSR0
    lfsr FSR1, sb; ;init FSR1
    clrf i,f ;i = 0

loop_top
    movlw 5
    cpfslt i ;i < 5?
    bra loop_end ;exit loop
    movff POSTINC1, POSTINC0
    incf i,f ;i++
    bra loop_top
loop_end
...rest of code...
```

Observe that one PIC18 statement replaced three C statements in this case because of the use of POSTINC1, POSTINC0.

PLUSWn Usage with *char* Arrays

In C

```
char sa[5];  
char sb[10];  
unsigned char i;
```

```
for (i=0; i < 5; i++) {
```

```
    sa[i] = sb[i];
```

```
}
```

In Assembly

```
CBLOCK 0x100  
    sa:5,sb:0x10,i      ;sa is 0x100,sb is 0x105  
ENDC  
  
    movlb 1              ;select bank 1  
    lfsr  FSR0, sa       ;init FSR0  
    lfsr  FSR1, sb       ;init FSR1  
    clrf  i,f            ;i = 0  
loop_top  
    movlw 5  
    cpfslt i              ;i < 5?  
    bra   loop_end       ;exit loop  
    movf  i,w             ;get i  
    movff PLUSW1, PLUSW0  
    incf  i,f             ;i++  
    bra   loop_top  
loop_end  
    ...rest of code...
```

PLUSWn Usage with *int* Arrays

In C

```
int sa[5];
int sb[10];
unsigned char i;
```

```
for (i=0; i < 5; i++) {
```

```
    sa[i] = sb[i];
```

```
}
```

In Assembly

```
CBLOCK 0x100
    ;sa is 0x100, sb is 0x10A
    sa:5*2, sb:0x0a*2,i
ENDC

    movlb 1                ;select bank 1
    lfsr  FSR0, sa         ;init FSR0
    lfsr  FSR1, sb;       ;init FSR1
    clrf  i,f              ;i = 0

loop_top
    movlw 5
    cpfslt i                ;i < 5?
    bra  loop_end          ;exit loop

    bcf  STATUS,C
    rlc  i,w                 ;w = i * 2
    movff PLUSW1, PLUSW0    ;LSByte copy
    addlw 1                  ;w = w+1
    movff PLUSW1, PLUSW0    ;MSBbyte copy
    incf  i,f                ;i++
    bra  loop_top

loop_end
    ...rest of code...
```

Must multiply W value by 2 before PLUSn usage because *sa*, *sb* point to *int* type.

A C Function with Pointers

In C

```
unsigned char lcase (  
unsigned char *ptr){
```

```
    unsigned char c;
```

```
    while (*ptr != 0) {
```

```
        c = *ptr;
```

```
        if (c > 0x40 &&  
            c < 0x5B) {
```

```
            // lowercase the char
```

```
            *ptr = *ptr | 0x20;
```

```
        }
```

```
        ptr++;
```

```
    }
```

In Assembly

```
CBLOCK 0x100
```

```
    ptr:2, c ;ptr contains address of string  
endc
```

```
lcase
```

```
    movlb 1 ;select bank 1  
    movff ptr,FSR0L ;set FSR low byte  
    movff ptr+1,FSR0H ;set FSR high byte
```

```
lcase_loop
```

```
    movf INDF0,w ;get current character  
    bz lcase_exit ;exit if zero  
    movwf c ;save character
```

```
    movlw 0x40  
    cpfsgt c ;c > 0x40?  
    bra lcase1 ;no, goto ptr++
```

```
    movlw 0x5B  
    cpfslt c ;c < 0x5B?  
    bra lcase1 ;no, goto ptr++
```

```
    movlw 0x20  
    iorwf INDF0,f ;do lower case
```

```
lcase1
```

```
    movf POSTINC0,f ;ptr++  
    bra lcase_loop ;loop back to top
```

```
lcase_exit
```

```
return
```

lcase changes uppercase characters to lower case.

Calling *lcase()* from *main()*

In C

```
char s1[]=
"Upper/LOWER0123";
```

```
main(void) {
lcase (s1);
}
```

In Assembly

```
CBLOCK 0x280
s1:0x10 ;reserve 16 bytes of space
endc

org 0
goto main

org 0x0200
main
;copy string in program mem
;to s1 data memory
call init_s1

;set up call for strent
movlb 1 ;select bank1
movlw low s1
movwf ptr
movlw high s1
movwf ptr+1 ;set ptr = s1 value
call lcase ;do lower case

here ;end program with infinite loop
goto here
```

Must copy the address of s1 into 'ptr' parameter for use by lcase subroutine.

ignore *init_s1* subroutine for now.

Program Memory vs. Data Memory

The PIC18 has strict separation of program memory vs. data memory (this is known as a **Harvard** architecture).

PIC18 instructions such as *movf*, *incf*, *addwf*, etc. cannot access locations in program memory.

Other processors treat program memory and data memory the same (**unified** memory structure). This allows instructions to access program memory the same as data memory.

Would like to use PIC18 program memory to **store tables of data** or **constant data** that does not change. This saves space in data memory, and provides non-volatile storage for the data.

But how can the table data or constant data be accessed if it is in program memory?

Tables in Program Memory


```
char s1[] = "Upper/LOWER0123";
```

The above string can be stored in program memory, with two bytes packed into one instruction word.

```
s1const  
da "Upper/LOWER", 0
```



```
s1const  
0x55,0x70 ;'U','p'  
0x70,0x65 ;'p','e'  
0x72,0x2f ;'r','/'  
0x4c,0x4f ;'L','O'  
0x57,0x45 ;'W','E'  
0x52,0x30 ;'R','0'  
0x31,0x32 ;'1','2',  
0x33,0x00 ;'3'
```



The *da* (*define ASCII*) assembler directive causes ASCII data to be packed two bytes to each instruction word.

Occupies 8 words of program memory.

Accessing Program Memory

The TBLPTR register is used to hold the program memory address that you want access.

Because a program memory address is 21 bits, three 8-bit registers are used to specify this address:

TBLPTR:

TBLPTRU (bits 20-16),

TBLPTRH (bits 15-8),

TBLPTL (bits 7-0).

Instructions that use TBLPTR to access program memory are *tblrd* (table read) and *tblwr* (table write) instructions. The TABLAT register holds the data that is read or written.

A PIC18 can program itself using the *tblwr* instruction; this is called *bootloading* and we will use a *bootloader* to program the PIC18.

tblrd, tblwr instructions

The forms for the *tblrd* (table read) instructions are:

<i>tblrd</i> *	TABLAT \leftarrow Prog. Mem (TBLPTR)
<i>tblrd</i> *+	TABLAT \leftarrow Prog. Mem (TBLPTR), TBLPTR++
<i>tblrd</i> *-	TABLAT \leftarrow Prog. Mem (TBLPTR), TBLPTR—
<i>tblrd</i> +*	TBLPTR++, TABLAT \leftarrow Prog. Mem (TBLPTR)

The forms of the *tblwr* (table write) instructions are:

<i>tblwt</i> *	HoldReg \leftarrow TABLAT
<i>tblwt</i> *+	HoldReg \leftarrow TABLAT, TBLPTR++
<i>tblwt</i> *-	HoldReg \leftarrow TABLAT, TBLPTR—
<i>tblwt</i> +*	TBLPTR++, HoldReg \leftarrow TABLAT

For writes, there are 8 holding registers as writes are done to program memory in blocks of 4 instruction words. Other registers are involved in the write operation as well, will discuss later.

Back to *lcase.asm*

The *lcase* subroutine expects *s1* to be in **data memory**.

The *init_s1* subroutine uses table reads to copy *s1const* in **program memory** to *s1* in **data memory**.

```
CBLOCK 0x280 ;s1 string
  s1:0x10←
ENDC org 0
```

```
copy string in prog. mem
; to data mem
; the init_s1 code is not shown
call init_s1
```

→

```
init_s1
;;subroutine copies s1const
;; to data memory s1
. . . <code not shown > . . .
s1const
  da "Upper/LOWER",0
```

s1_init: Table Read Example

```
CBLOCK 0x280 ;s1 string
s1:0x10
ENDC org 0
```

s1const

```
da "Upper/LOWER",0
```

s1_init

```
movlw upper s1const
movwf TBLPTRU
movlw high s1const
movwf TBLPTRH
movlw low s1const
movwf TBLPTRL
```

```
lfsr FSR0,s1 ;; point FSR0 at s1
call init_str
return
```

init_str

```
tblrd*+ ;use table read to get byte
movf TABLAT,w ; move to w
movwf POSTINC0 ;store byte to S1, FSR0++
bnz init_str ;loop if not end of string
return ; finished
```

in program memory.

TBLPTR = address of *s1const*

lfsr is a handy instruction for loading a 12-bit literal into a FSRx register.

init_str copies string in program memory pointed to by TBLPTR to data memory location pointed to by FSR0.

What do you have to know?

- How subroutine call/return works
- How the stack on the PIC18 works
- How to pass parameters to a subroutine using a static allocation method
- How pointers work in PIC18 (INDF_x, FSR_x registers)
- How to access table data that is stored in program memory