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.

C Subroutine (*vlshift*)

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

main(){
 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);
}
```

subroutine

parameters

subroutine body

subroutine return

Main program

subroutine call

vlshift.asm

```
;  Parameter space for vlshift
 CBLOCK 0x040
  v, amt
 ENDC
;; return value in w

vlshift
  movf   amt,f
vlshift_loop
  btfsc  STATUS,Z   ; amt==0?
  goto   vl_return
  bcf    STATUS,C
  rlf    v,f        ; v = v << 1
  decf   amt,f      ; amt--
  goto   vlshift_loop
vl_return
  movf   v,w        ;return(v)
  return
```

vlshift parameters

subroutine body

move result into *w* register before return

return to main program

vlshift.asm (cont.)

```
;Parameter space for main
CBLOCK 0x020
 i,j,k
ENDC

 org    0
 ; initialize main program variables
 movlw  0x24
 movwf  i      ; i = 0x24
 movlw  0x2
 movwf  j      ; j = 2
 ;; setup subroutine parms
 movf   i,w
 movwf  v
 movf   j,w
 movwf  amt
 call   vlshift
 movwf  k      ; k = vlshift(v,amt);
here
 goto here
```

main() variables

initialize *i* , *j* variables

copy *i* , *j* variables to parameters *v*, *amt* for subroutine

subroutine call

return value in *w* register, copy to *k*.

call, *return* Statements

	B B B B B B B B B B B B B B B B 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 3 2 1 0 9 8 7 6 5 4 3 2 1 0
call *k*	1 0 0 k k k k k k k k k k k k
return	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
retlw k	1 1 0 1 x x k k k k k k k k

call (call subroutine) : Push PC of next instruction onto stack, theb PC[10:0] ← k, PC[12:11] ← PCLATH

return (ret from subroutine): PC ← pop top-of-stack

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

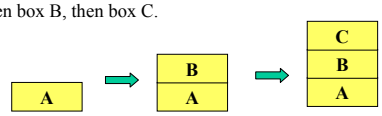
The Stack

- In µPs, 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 PIC, 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 PIC, the return address is PC+1, if PC is the location of the *call* 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.

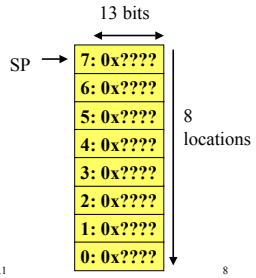
The PIC16 Stack

The PIC16 stack has limited capability compared to other µPs. It is only used during call/return, and is 8 locations deep.

For a call, PC+1 is *pushed* onto the stack.

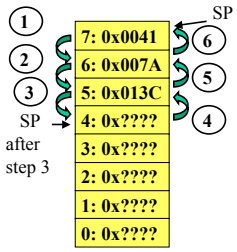
A **push** means to store PC+1 at [SP], decrement SP ([SP] ←PC+1, SP--)

A return instruction pops the PC off the stack. A **pop** means to increment SP, then read [SP] and store to the PC (SP++, PC ←[SP])



Call/Return Example

```
main
  ....
  0x40  call Sub_A
  ....
Sub_A
  ....
  0x69  call Sub_B
  ....
  return
Sub_B
  ....
  0x13B call Sub_C
  return
Sub_C
  ....
  return
```



Stack Overflow

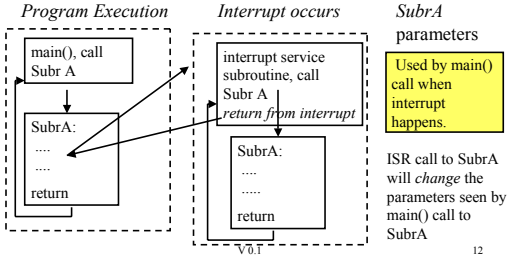
- Stack **overflows** on the 9th *call* instruction without a return
  - Stack pointer wraps back to 7, overwrites the return address of the first *call* instruction
  - Obviously, this results in incorrect program behavior
  - Hard to debug, no status flags that indicate stack overflow
- Caution: on PIC16, do not nest subroutine calls more than 8 deep
  - actually want to nest less than this if interrupts are used, more on this later.

Back to Parameter Passing

- The *vlshift.asm* program used a *static* memory area for parameters
  - *static* means that the location is fixed
- This happens to be the most efficient method for the PIC16, but has limitations
  - 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.

The problem with Static Parameters

An ***interrupt*** is an external event to the processor (e.g. a change in pin voltage value) that causes the program to jump to an interrupt service routine (ISR). The ISR is finished, normal program execution is resumed. If the ISR calls Subr A, then static parameters are overwritten!



Indirect Addressing

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.

Pointers in *C*

```
char s1[] = "Upper/LOWER.";
unsigned char strcnt (ptr)
unsigned char *ptr;
{
   unsigned char i;
   i = 0;
   while (*ptr != 0) {
     ptr++; i++;
   }
   return(i);
}

main(){
unsigned char i;
 i = strcnt(s1);
}
```

s1 is address of first character

Will use *C* to illustrate pointer usage, then show how this is implemented in PIC16 assembly.

strcnt returns number of characters in string reference by ptr. Recall that *C* strings are terminated by 0x00.

'*' operator declares that a variable is a pointer variable

'*ptr' returns data that pointer is accessing

ptr++ increments pointer to next address of data. For *char* data, increment by 1.

Pointers in PIC16

FSR special register (location 0x04) : holds the value of the pointer itself (contains the address of the data)

INDF special register (location 0x00) : used to access the data that FSR points to.

```
char s1[] = "Upper/LOWER.";

char *s, c;
/* point s at first
   char of s1  */
s =  s1;
/* get first char */
c =  *s;
```

```
CBLOCK 0x20
 s1:16, c
ENDC

movwl 0x20    ; w← addr. of s1
movwf FSR     ;FSR points at
              ;first char

movf  INDF,w  ; w← [0x20]
              ; w←  *s
movwf c       ; c = *s
```


strcnt in PIC16

```
int strcnt (ptr)
unsigned char *ptr;
{
   unsigned char i;
   i = 0;
   while (*ptr != 0) {
     ptr++; i++;
   }
   return(i);
}
```

variable *i* in *strcnt* parameter block will contain string length.

```
;parms for strcnt
CBLOCK 0x50
 i,ptr
ENDC

strcnt
  clrf i         ;; i = 0
  movf   ptr,w   ;; w = ptr
  movwf  FSR     ;; FSR = ptr
strcnt_loop
  movf   INDF,w     ;;  w = *ptr
  btfsc  STATUS,Z   ;; *ptr == 0?
  return            ;; yes, exit
  incf   i,f        ;; i++
  incf   FSR,f      ;; ptr++
  goto   strcnt_loop
```


strcnt in PIC16 (cont.)

```
char s1[] = "Upper/LOWER.";

main(){

 unsigned char i;

 i = strcnt(s1);
}
```

Use label 's1' instead of 0x020, increases code clarity.

After call, variable *i* in *strncnt* parameter block has the string length.

```
;storage for s1 string
CBLOCK 0x20
 s1[16]
ENDC

org 0
; copy string in prog. mem
; to data mem
; the init_s1 code is not shown
call init_s1

;; set up call for strcnt
movlw  s1
movwf  ptr       ;; set ptr = s1
call   strcnt    ;; do strcnt
here
 goto here     ;; loop forever
```


Program Memory vs. Data Memory

The PIC16 has strict separation of program memory vs. data memory (this is known as a **Harvard** architecture).

PIC 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 PIC 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 be accessed if it is in program memory?

Tables in Program Memory

```
char s1[] = "Upper/LOWER.";
```

The above string can be stored in program memory as a series of *retlw* instructions.

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

```
s1const
  retlw 0x55  ;'U'
  retlw 0x70  ;'p'
  retlw 0x70  ;'p'
  retlw 0x65  ;'e'
  retlw 0x72  ;'r'
  retlw 0x2f  ;'/'
  retlw 0x4c  ;'L'
  retlw 0x4f  ;'O'
  retlw 0x57  ;'W'
  retlw 0x45  ;'E'
  retlw 0x52  ;'R'
  retlw 0x2E  ;'.'
  retlw 0x00
```

The *dt* (*define table*) assembler directive causes byte data to be encoded as a series of *retlw* instructions in program memory.

Accessing Table Data

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

Assume we wanted to retrieve *s1const[5]* ('/')

Must be a *call*, so can do a *retlw*

Assembler directive **high** returns higher 5-bits of address *s1const+5*, save in PCLATH.

Put lower 8-bits of *s1const+5* into PCL; write to PCL causes PCLATCH → PCH.

Next instruction executed is at PCH:PCL = *s1const+5*. Instruction is 'retlw 0x2f'! So 0x2f ('/') returned in *w* reg.

```
CBLOCK 0x020      ;
 tdata  ; data value from table
ENDC

org 0
call get_data
movwf tdata  ; save tab data
here
  goto here

get_data
  movlw  high (s1const+5)
  movwf  PCLATH
  movlw  low (s1const+5)
  movwf  PCL  ; does retlw!

s1const
  dt "Upper/LOWER.",0
  end
```


PCH:PCL Loading

FIGURE 2-5: LOADING OF PC IN DIFFERENT SITUATIONS

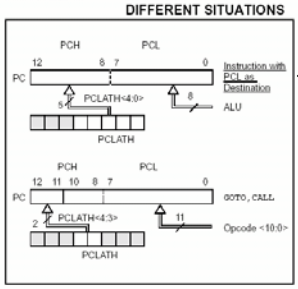


Table read makes use of this capability.

Any instruction with PCL as destination causes transfer of PCLATH<4:0> to PCH.

Back to *strcnt.asm*

The *strcnt* 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.

Complete code for *strcnt.asm* attached to lecture, see if interested in details.

```
;data storage for s1 string
 CBLOCK 0x20
  s1[16]
 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
   dt "Upper/LOWER",0
```


PIC18 Indirect Addressing

The PIC18 has three sets of INDF/FSR registers
⇒ INDF0/FSR0, INDF1/FSR1,INDF2/FSR2

The INDFx registers work as on the PIC16; INDFx is used to access the value that FSRx is pointing to.

```
movf   INDF0, w     ; w ← [FSR0]
```

However, there are four more INDF-like registers associated with each FSRx. These register names, and operations are shown below:

```
movf   POSTDEC0, w   ; w ← [FSR0], FSR0--
movf   POSTINC0, w   ; w ← [FSR0], FSR0++
movf   PREINC0, w    ; FSR0++, w ← [FSR0]
movf   PLUSW0, w     ; w ← [FSR0+w]
```


Indirect Addressing Modes

- The PIC18 advanced indirect addressing modes are common features on other processors. The features below are useful for stack data structures
  - *postdec* : post-decrement indirect addressing
  - *postinc* : post-increment indirect addressing
  - *preinc* : pre-increment indirect addressing
- The *plusw* addressing mode is called **indexed indirect**.
  - This is a base addressing mode of every modern processor.
  - Allows an address to be computed by adding an offset to a base address. Very useful for array addressing (i.e., *myarray[i]*)

vlshift Revisited

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

Data stack on entry to vlshift

TOS+2: amt
TOS+1: v
TOS:

FSR0 ⇨

To pass these parameters, will use a **data stack**.

This is a different stack from what is used for return addresses. This is just a data structure created by the programmer.

v will be at FSR0+1

amt will be at FSR0+2

A *data* stack

A data stack will be used to pass parameters.

Will place the top-of-stack at location 0x7F, will want to the stack to grow down as items are placed on it.

Will use FSR0 as the stack pointer

To store items on stack (*push*):

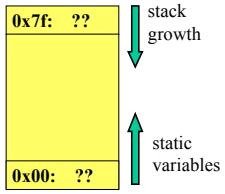
```
movf   POSTDEC0, w
```

To access items on stack:

```
movf   PLUSW, w
```

To remove items from stack (*pop*):

```
movf   PREINC, w
```



Use of data stack with *vlshift()*

Before calling *vlshift*, main routine will push *amt*, *v* on stack. On entry to *vlshift*, stack will look like right.

Data stack on entry to *vlshift*

TOS+2: amt
TOS+1: v
TOS:

FSR0 ⇨

To perform "***amt- -***" from within *vlshift* subroutine, will do:

```
movlw  2      ; w = 2
decf   ADDW0  ; [FSR0+2]--
```

To perform "**v = v << 1** " from within *vlshift* subroutine, will do:

```
bcf    STATUS,C  ; clear carry
movlw  1         ; w = 1
rlcf   ADDW0     ; [FSR0+1]--
```


vlshiftp18.asm (*vlshift* subr.)

```
vshift
 ;will use w to index to v movlw  2
  movf   PLUSW0,f   ;test 'amt'
vshift_loop
  btfsc  STATUS,Z   ; amt==0?
  goto   vl_return
  bcf    STATUS,C
  movlw  1
  rlcf   PLUSW0     ; v = v << 1
  movlw  2
  decf   PLUSW0,f     ; amt--
  goto   vshift_loop
vl_return
  ;; 'v' has result, pop off into w reg
  ;; before return
  movf  PREINC0,w   ; w <- v
  return
```

Want new *v* value to return in *w* reg, so pop this off of stack and place in *w* before returning.

vlshiftp18.asm (*main*)

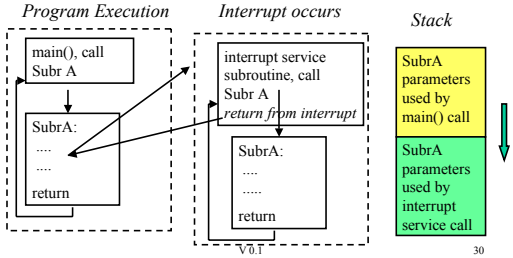
```
org    0
;; initialize software stack
;; use FSR0 as software stack
lfsr FSR0, 0x7f
 ; initialize main program variables
 movlw  0x24
 movwf  i      ; i = 0x24
 movlw  0x2
 movwf  j      ; j = 2
 ;; setup subroutine parms
 movf   j,w
 movwf  POSTDEC0  ; push amt value onto stack
 movf   i,w
 movwf  POSTDEC0  ;push v value onto stack
 call   vshift
 movwf  k    ; k = vshift(v,amt);
 movf   PREINC0,f  ; pop amt off stack to clean up
here
  goto here
```

Place *amt*, then *v* onto data stack before call.

The *v* parameter was removed from stack by *vlshift*; need to remove the *amt* parameter.

Why use stacks for a parameter passing?

Subroutine parameter variables are dynamically allocated.
If subroutine is interrupted, then called again, this works because new space is allocated for parameters



What do you have to know?

- How subroutine call/return works
- How the stack on the PIC16 works
- How to pass parameters to a subroutine using a static allocation method
- How pointers work in PIC16 (INDF, FSR registers)
- How to access byte table data that is stored in program memory
- PIC18 indirect addressing
- How parameter passing via stack works for PIC18