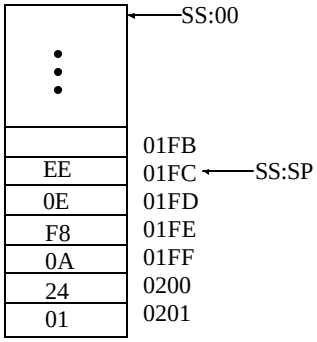
Lecture 16: Passing Parameters on the Stack

- Quick Stack Review
- Passing Parameters on the Stack
- Binary/ASCII conversion

Push Examples

```
;assume SP =
   0202

mov  ax, 124h
push  ax
push   0af8h
push  0eeeh
```



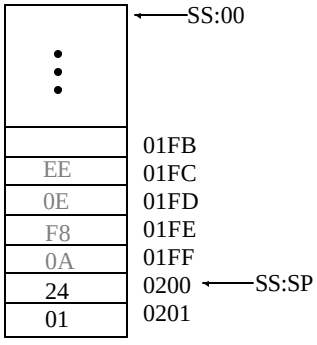
Pop Examples

```
;initial SP = 01FC

pop      BX
pop      DX

BX = 0EEEh
DX = 0AF8
```

If we were to pop again, the next value popped off would be 0124.



CALL and RET

- CALL – pushes IP on the stack (recall, IP holds the address of the next instruction), puts the address of the label (subroutine) into IP.
- RET – pops the stack into IP to return to the point at which the subroutine was called.

Passing Parameters using Registers

- Passing parameter values to a subroutine
- Passing base address of parameter to a subroutine.
- Example – keyboard buffer:

```
maxlen db      20   ; max chars to
  input
actuallen db   ?    ;DOS will put the
  number read here
inbuf  db      20 dup (' ') ; where
  DOS will put the data read in
….
mov   dx, offset maxlen
```

Passing Parameters Using Registers, cont.

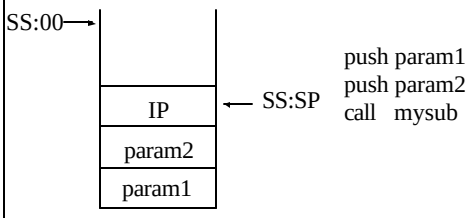
- Advantages:
- Problems:

Passing Parameters using the Stack

- Several parameters? – use the stack
- Push parameters before the CALL
- Why can't you just pop them off again in your subroutine?

Passing Parameters on the Stack, cont.

- When you execute CALL, it pushes IP on the stack.
- If your called routine executes a pop it won't get a parameter, it will get the return IP.



Note: stack drawn as 16-bit WORDS

Getting at Our Parameters

- You can pop the IP off the stack and then get your data.
  - Don't do this!!!!
  - Why?
- A better solution – use the BP register!
  - BP works like BX except it holds an offset from SS (recall BX holds an offset from DS)

Indirect Addressing Using BP

```
mov ax, [bx] ; this moves the
             ; word at DS:BX
             ; into AX

mov  dx, [bp]    ; this moves the
                 ; word at SS:BP
                 ; into DX
```

- you can also use indirect addressing with displacement:

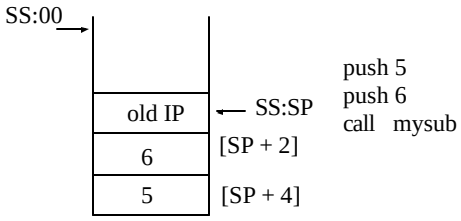
```
mov   ax, [bp + 2] ;take the address
                   ;in BP, add 2
                   ;load AX with the
                   ;contents of the word at
                   ;that address.
```

Setting up a Stack Frame

- Parameters are passed on the stack by setting up a stack frame.
  - The calling routine pushes the parameters on the stack.
  - The CALL instruction is used to call the subroutine.
  - The subroutine pushes BP on the stack (why?).
  - The subroutine copies the value in SP into BP so it can be used to retrieve the variables.

Stack Frame Example

- After calling our subroutine:

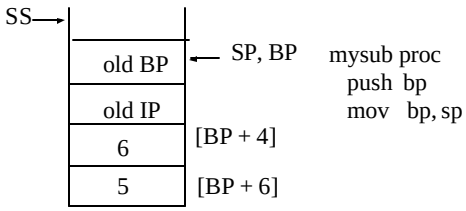


Note: stack drawn as 16-bit WORDS

old IP? The return address – this is the offset of the instruction immediately after the call.

Stack Frame Example, cont.

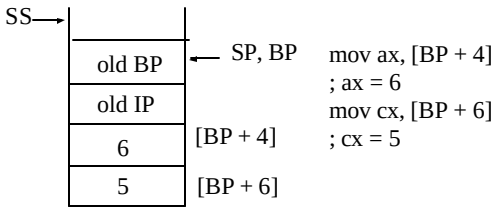
- Save and set up BP (in procedure mysub):



Note: stack drawn as 16-bit WORDS

Retrieving Parameters

- Use indirect addressing with displacement:



Note: stack drawn as 16-bit WORDS

```
 TITLE Demonstrates parameter passing on the stack
;
; This program adds a constant value to each element of an array.
; It uses a procedure, and parameters are passed on the stack i n
; the following order:  the start address of the array, the constant
; value to be added, and the number of elements in the array.

            .model small
            .stack 100H

AddNumber  EQU  12H       ;number to be added
ArraySize  EQU   9H

            .data
TestArray  DW  3AH, 4AH, 5AH, 6AH, 7AH, 1234H, 5678H
             DW  9ABCH, 0DEFH

            .code
            .startup

            mov     ax, offset TestArray
            push    ax
            mov     ax, AddNumber
            push    ax
            mov     ax, ArraySize
            push    ax
            call    ArrayInc
            nop
            .exit
```

```
ArrayInc PROC    NEAR

            push    bp
            mov     bp, sp      ;set up frame for this call

            push    ax          ;save registers destroyed by this
            push    bx          ;procedure
            push    cx
            push    dx

            mov     cx, [bp+4]          ;size of array
            mov     ax, [bp+6]          ;the constant to be added
            mov     bx, [bp+8]          ;start address of array

next:       mov     dx, [bx]    ;get array element
            add     dx, ax      ;add constant to it
            mov     [bx], dx    ;store it back in array
            add     bx,2        ;point to next array element
            loop    next

            pop     dx          ;restore registers
            pop     cx
            pop     bx
            pop     ax

            pop     bp
            ret                 ;returns

ArrayInc ENDP
            end
```

What's Wrong with this Example?

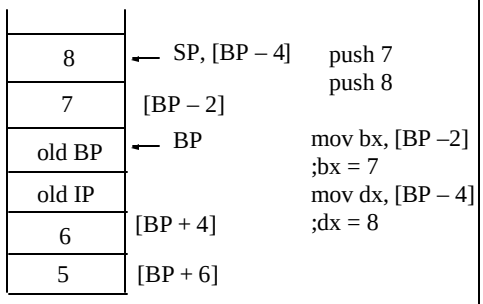
Cleaning up the Stack

- Clean up the stack using RET x, where x is the number of bytes to add to SP after the return:

```
RET 6   ;clean up the 6 bytes of
        ;parameters by adding
        ;6 to SP after popping IP
```

Using the Stack for Local Variables

- The stack is useful for local variables



Note: stack drawn as 16-bit WORDS

Pass By Reference vs. Pass By Value

- Defn:

- Registers or the stack can be used to pass by value or by reference.
- Why does it matter?

ASCII-Binary Conversions

- In a high-level language, you just read the number:
  - read (num) or
  - scanf("%d", &num),
  - cin >> num, or…
- What's going on behind the scenes?
- Say the user enters 361
  - They enter 3 separate keys: "3", "6", "1"
  - These come in as ASCII values
  - They must be converted into the integer 361 and stored.

Algorithm (in Pseudo-Code)

Result <- 0
Multiplier <- 10

Convert:
  Get a character
  If not a digit char, go to Finish
  Else
    Strip the ASCII bias off of the digit character (subtract 30h)
    Result <- Result * multiplier + digit
    go to Convert
Finish:

```
        mov     bx, 0           ;bx contains running total
        mov     di, 10          ;di is the multiplier
        sub     si, si          ;si =0 means positive,
                                ;si =1 means negative

        mov     ah, 01h   ;read one character from keyboard
        int     21h
        cmp     al, '-'   ;was character a minus sign?
        jne     Convert         ;no, it was a digit
        inc     si              ;yes, remember to negate at end
        jmp     GetNext         ;go get first digit

Convert:
        cmp     al, '0'   ;when you read a non-digit, you're
        jb      Finish          ;done
        cmp     al, '9'
        ja      Finish

        mov     cl, al          ;al will be used by MUL
        mov     ax, bx          ;put running total in ax
        mul     di              ;multiply AX by 10

        sub     cl, 30h   ;convert single digit to binary
        mov     ch, 0
        add     ax, cx          ;add new digit to running total

        mov     bx, ax          ;put running total back in BX,
                                ;because AX will be needed by next
                                ;input function

GetNext:
        mov     ah, 1           ;prepare to read in next character
        int     21h
        jmp     Convert

Finish:
        cmp     si, 1           ;see if you need to negate number
        jne     NotNeg
        neg     bx      ;yes you do

NotNeg:
        nop     ;check number with debugger
```

Binary-ASCII Conversions

- To print a number, it needs to be converted from binary to ASCII.
- For example:
  - 361 decimal -> "3", "6", "1"

Algorithm (in Pseudo-Code)

put number in AX
repeat
    divide AX by 10
    convert remainder (DL) to ASCII
    save remainder in a buffer
until AX = 0

-> numbers are generated in *reverse order*