Lecture 16: Passing Parameters on the Stack

- Quick Stack Review
- Passing Parameters on the Stack
- Binary/ASCII conversion
  ** needed for HW 5 **
- Homework 5

Push Examples

; assume SP = 0202
mov ax, 124h
push ax
push 0af8h
push 0eeeh

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

• We’ve been using registers to pass parameters (when? for the INT 21h calls!)
• You can also use registers to pass a base address of a group of memory locations.
• Example – our keyboard buffer:

```assembly
maxlen db 20 ; max chars to input
actulen db ? :DOS will put the number read here
inbuf db 20 dup (‘ ’) ; where DOS will put the data read in
```

– If you load the address of maxlen into a register (using MOV BX offset maxlen) before calling the subroutine then the subroutine will be able to access the other data members by using indirect addressing.

Passing Parameters Using Registers, cont.

• Advantages:
  - Easy for passing a small number of parameters
  - Useful for passing the start address of a block of parameters
• Problems:
  - There are only four general purpose registers (AX, BX, CX, DX) and they are needed for many things.

Passing Parameters using the Stack

• For several parameters, it’s better to use the stack.
  (this is what high level languages do)
• Before the CALL, push the parameters on the stack.
• But… you can’t just pop them off again in your program… why?

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.

```
SS →
  IP
  SP
  param2
  param1
  push param1
  push param2
  call mysub
```

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!!!!
  – If you forget to put IP back on top, you will return off into never-never land!
• 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

```assembly
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? so it can be restored later).
  – 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:

```
<table>
<thead>
<tr>
<th>SS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|      | push 5
|      | push 6
|      | call mysub
| SP   |  |
| 6    | [SP + 2]
| 5    | [SP + 4]

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

Note: stack drawn as 16-bit WORDS
### Stack Frame Example, cont.

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

```
SS →  
| old BP   |
| old IP   |
| 6        |
| 5 [BP + 6] |
```

Note: stack drawn as 16-bit WORDS

### Retrieving Parameters

- Use indirect addressing with displacement:

```
SS →  
| old BP   |
| old IP   |
| 6 [BP + 4] |
| 5 [BP + 6] |
```

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 in the following order: the start address of the array, the constant value to be added, and the number of elements in the array.

```assembly
.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 9AHCH, 0DEFH

code
startup

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

```assembly
ArrayInc PROC NEAR
push bp ; set up frame for this call
mov bp, sp
push bx ; save registers destroyed by this procedure
push cx
push dx
mov ax, [bp+4] ; size of array
mov bx, [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?

• There were three parameters pushed on the stack.
• After the return, the parameters were just left there.
• If you did this enough, you’d fill up the stack with old data!
• What you should do is 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

• Once you’ve set up your stack frame, you can use it for local variables as well:

```
  8 ← SP, [BP – 4]   push 7
  7 [BP – 2]         push 8
old BP ← BP         mov bx, [BP – 2]
bx = 7
dx = 8
6 [BP + 4]
5 [BP + 6]
```

Note: stack drawn as 16-bit WORDS

Pass By Reference vs. Pass By Value

• The book talks about pass by value and pass by reference.
  – Pass by value – the value of the parameter is passed into the subroutine
  – Pass by reference – the address of the parameter is passed into the subroutine
    (we just saw this in the array example!)
• Either the registers or the stack can be used to pass by value or by reference.
• Why does it matter? Pass by reference is used if you want the procedure to modify a variable.

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
mov ax, 10
sub si, si
mov al, 10
mov bh, 0
mov cl, 30h
add ax, cx
addax
jmp GetNext

Finis:
cmp al, 0
jz done

Conv:
cmp al, '0'
jb Finis

GetNxt:
mov ah, 1
int 21h
jmp Conv

Finis:
cmp si, 1
jz NotNeg

NotNeg:
add bx, 0
jmp num

num:
op
jmp Conv