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

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:
  
  ```
  maxlen  db  20 ; max chars to input
  actualen 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.

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

- `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:
  - `SS:SP` push 5
  - `push 6` call mysub
  - \([SP + 2]\) \([SP + 4]\) old IP
  - 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):

```
SS →       SP, BP  mysub proc
         old BP
         old IP
         [BP + 4]
         6
         [BP + 6]
```

Note: stack drawn as 16-bit WORDS

Retrieving Parameters

- Use indirect addressing with displacement:

```
SS →       SP, BP  mov ax, [BP + 4]
          old BP  ; ax = 6
          old IP  mov cx, [BP + 6]
          [BP + 4]
          5
          [BP + 6]
```

Note: stack drawn as 16-bit WORDS

TITLE Demonstrates parameter passing on the stack
set up frame for this call
save registers destroyed by this procedure
size of array
the constant to be added
start address of array
get array element
add constant to it
store it back in array
point to next array element
restore registers
returns

ArrayInc PROC NEAR
push bp
mov bp, sp
push ax
push bx
push cx
push dx
mov cx, [bp+4]
mov ax, [bp+6]
mov bx, [bp+8]
next:
    mov dx, [bx]
    add dx, ax
    mov [bx], dx
    loop
    next
    pop dx
    pop cx
    pop bx
    pop ax
    pop bp
    ret
ArrayInc ENDP
end
What’s Wrong with this Example?

Cleaning up the Stack

• 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:

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

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.

Note: stack drawn as 16-bit WORDS
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:

Binary-ASCII Conversions

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

```
mov bx, 0 ; bx contains running total
mov ax, 10 ; ax is the multiplier
sub al, si ; subtract bias (positive, and 3 is negative)

mov ah, 01h ; read one character from keyboard
mov al, 21h
cmp al, '0' ; is this a digit?
je Convert
ja Finish

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

mov ch, cl ; cl will be used by MUL
mov bx, bx
mul ax ; multiply ax by 10
sub bx, 30h ; convert single digit to binary
mov ch, cx ; add new digit to running total
mov bx, ax
add bx, bx

GetNext:
  mov bx, 0
  jmp Convert

Finish:
  cmp ax, 0
  je Exit
  jne GetNext

Exit:
  mov bx, 0
  jmp Exit

Exit:
  mov bx, 0
  jmp Exit
```

```
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*