Lecture 14: The Stack

- What is it?
- What is it used for?

The Stack

- A special memory buffer (outside the CPU) used as a temporary holding area for addresses and data
- The stack is in the stack segment.
- The stack is a buffer of words.
- SP holds the address (offset from SS) of the last data element to be added to the stack. -> the TOP of the stack.
- The stack is a LIFO structure (last-in, first-out)

Adding Elements

- Elements are added by pushing them on to the stack.
  - `mov ax, 0AFh`
  - `push ax`

```
(SS) -> 5000

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>501FB</td>
<td>01FB</td>
<td></td>
</tr>
<tr>
<td>502FC</td>
<td>01FC</td>
<td></td>
</tr>
<tr>
<td>501FD</td>
<td>01FD</td>
<td></td>
</tr>
<tr>
<td>501FE</td>
<td>01FE</td>
<td></td>
</tr>
<tr>
<td>501FF</td>
<td>01FF</td>
<td></td>
</tr>
<tr>
<td>50200</td>
<td>9A</td>
<td></td>
</tr>
<tr>
<td>50201</td>
<td>1A</td>
<td></td>
</tr>
</tbody>
</table>

SS 5000
SP 0200

(SS) -> 5000

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>501FB</td>
<td>01FB</td>
<td></td>
</tr>
<tr>
<td>502FC</td>
<td>01FC</td>
<td></td>
</tr>
<tr>
<td>501FD</td>
<td>01FD</td>
<td></td>
</tr>
<tr>
<td>501FE</td>
<td>AF</td>
<td></td>
</tr>
<tr>
<td>501FF</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>50200</td>
<td>9A</td>
<td></td>
</tr>
<tr>
<td>50201</td>
<td>1A</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Adding elements, cont.

• When data is pushed,
  – SP = SP – 1
  – high order data byte is stored in SS:SP
  – SP = SP – 1
  – low order data byte is stored in SS:SP
• SP is left pointing to the item just pushed (the “top” of the stack)

Removing (Retrieving) Information

• Elements are removed by *popping* them from the stack.
  – pop bx

Retrieving, cont.

• When data is popped:
  – copy SS:SP into low-order byte of the register/memory location
  – increment SP
  – copy SS:SP into high-order byte of the register/memory location
  – increment SP
• SP is pointing to the new top of stack.
• Notice: the data does not get deleted from the memory location it was in, but it will be overwritten the next time data is pushed on to the stack.

A Word of Warning!

• The book draws the stack going from high memory to low memory:

  – picture from Irvine
Warning, cont.

• Why draw it that way?
• Well, the stack grows down, from high memory to low memory until SP = SS.
• If it’s drawn like the book has it, then “down” is actually down.
• If it’s drawn like I’ve done it, it looks more like the other drawings of memory we’ve seen in this class. Plus the item at the top of the stack looks like it’s on top.

So...

• You need to be aware of what is happening to SP when items are being pushed and popped.
• Remember that memory can be drawn differently but the assembly code is still doing the same thing!
• SP always points to the “top” (the item that would be popped first).
• It’s important when drawing the stack to be sure write the offset or address next to it so you can tell what is going on.

Push

• PUSH decrements SP and copies a 16 bit or 32 bit register or memory operand onto the stack at the location pointed to by SP.
• Allowed forms:
  – push reg
  – push memval
  – push imm
• Examples:
  – push AX
  – push count ; where count dw ?
  – push 0a23h
• You can’t push a byte on to the stack!

Push Examples

; assume SP = 0202

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

```
01FB
EE
0E
F8
0A
24
01
01FF
01FE
0200
0201
```
Pop

- POP copies the contents of the stack pointed to by SP into a register or variable and increments SP. Two registers (CS and IP) cannot be used as operands.
- Allowed forms:
  - pop reg
  - pop memval
- Examples:
  - pop cx
  - pop count ;where count dw ?

Pop Examples

:initial SP = 01FD

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>FF</td>
<td>FB</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

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

More Instructions

- PUSHF – pushes the flags register on to the stack
- POPF – restores the flags register from the stack

Common Stack Uses

- A good temporary save area for registers.
- Subroutine return addresses are saved on the stack.
- Procedure arguments can be passed on the stack (high level languages typically do this).
- High level languages use the stack as a place to store local variables.
Stack Overflow

- The 8086 hardware does not check for stack overflow!
- The CPU will let you keep pushing beyond bounds of the stack (possibly destroying important memory).
- Your program must check by:
  \[\text{CMP SP, 0}\]
  If there is any chance that stack overflow might occur.

Saving and Restoring Registers

- example p 135, Irvine

Procedures

- As with high-level languages, it is useful to be able to call procedures from your assembly program.
- You did this in homework 3 with writn.
- Some terms:
  - function: a procedure that returns a value
  - subroutine: a procedure (the terms are interchangeable)
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.

TITLE Procedure Calls
; two calls to a procedure that increments every element
; of an array
.model small
.stack 100h
.data
List DW 5FFFh, 0Ah, 12h, 17h
Array count1 DW 4
count2 DW 6
code
.startup
mov bx, offset List ;first call
call IncProc
mov bx, offset Array ;second call
call IncProc
nop ;can examine arrays with
.exit ;the debugger here
;return to DOS

 procedural to increment all the elements in an array of words
;BX contains the base address of the array
;CX contains the number of elements in the array
IncProc proc
sub si, si ;start index at 0
Lup: inc word ptr [bx] [si] ;use indexed addressing
add si, 2 ;dealing with words, not bytes
loop Lup ;go to next element
ret ;return - must be used with Call
IncProc endp

 Nested Procedure Calls

- example 4 from text
Near vs Far Calls

• When the caller and subroutine are in the same segment, the CALL instruction generates code for a NEAR CALL.
• When the caller and subroutine are in different segments, the assembler generates a FAR CALL.

FAR CALL

• FAR CALL
  – saves both CS and IP on the stack (pushes CS first).
  – loads the subroutine’s CS and IP
  – generates a different form of return (RETF) that restores CS and IP from the stack

Using FAR CALL

• Use FAR:
  – when linking asm routines to HLL programs (some require FAR calls)
  – when calling certain library routines that are set up for far calls
  – when your program size exceeds 64K (medium or large memory models). In this case, there will be multiple code segments, requiring far calls.
• Mostly, you will use NEAR. The assembler assumes you want NEAR unless you tell it otherwise.
• NEAR calls will execute faster (less pushing and popping)

Far Call, cont.

• example, p. 140 in Irvine