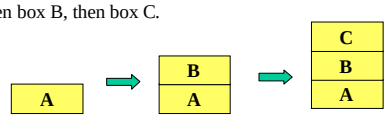
The Stack

- The stack is a memory area intended for storing temporary values.
- The stack is accessed by the SS:SP segment/offset combination (StackSegment: StackPointer)
- Some instructions make use of the stack area during execution (*push*, *pop*, *call*, *ret*, many others)
- If you need to store temporary values in memory, the stack is the best place to do so.

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).

Storing data on X86 stack via PUSH

The SP (Stack Pointer) register is used to access items on the stack. The SP register points to the LAST value put on the stack.

The PUSH operation stores a value to the stack:

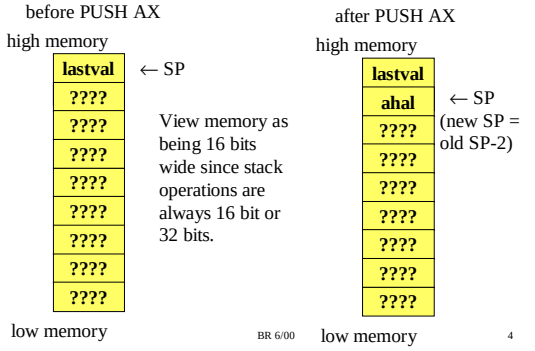
```
PUSH  AX      ;  SP= SP-2,  M[SP] ← AX
```

The "push AX" instruction is equivalent to:

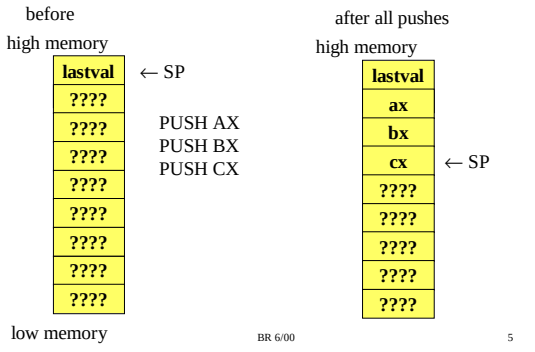
```
sub  SP, 2        ; decrement SP by 2 for word operation
mov [SP], AX   ; write value to stack.
```

Stack access only supports 16-bit or 32-bit operations

Visualizing the PUSH operation



Multiple Pushes



Reading Data from X86 stack via POP

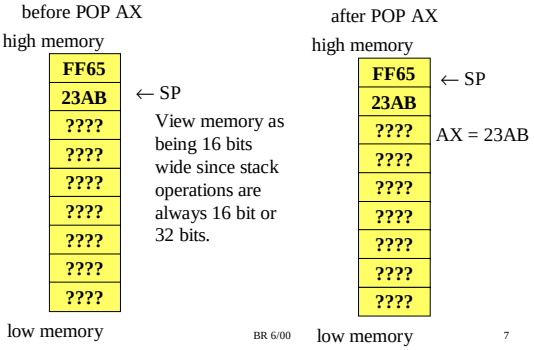
The POP operation retrieves a value from the stack:

```
POP  AX      ; AX ← M[SP] , SP= SP+2
```

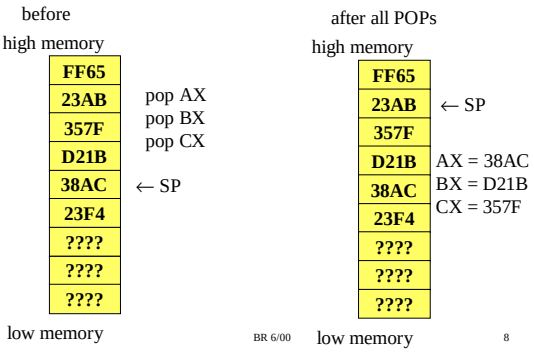
The “pop AX” instruction is equivalent to:

```
mov AX,[SP]      ; read value from top of stack
add  sp,2        ; increment SP by 2 for word operation
```

Visualizing the POP operation



Visualizing multiple POP operations



Saving/Restoring Registers

```
        .data
msg     db 'This is a message. $'
        .code
        push  ax          ;save ax
        push  dx          ;save dx
        mov   ah,9        ;display strng func
        mov   dx,offset msg
        int   21h         ; DOS call
        pop   dx          ; restore dx
        pop   ax          ;restore ax
```

Often need to save registers for some reason – stack is best place to do this. Note that POP operations should occur in reverse order of PUSH operations for correct values to be loaded into registers!

Other Push/Pop operations

a. Can push/pop any register except CS, IP

b. On 286+, can push an immediate value or memvalue:

```
push  AF23h          ; push 16-bit immediate on stack
push [bx+2]          ; push value from Mem on stack
```

c. PUSHF/POPF will push/pop flag register

d. PUSHA/POPA (286+) -- pushes/pops registers AX,CX,DX,BX,SP,BP,SI,DI on stack in this order

e. PUSHAD/POPAD (386+) – pushes same register, but 32-bit value (EAX, ECX, etc)

Procedures

- Group of Instructions that Perform Single Task
  - *(can be used as)* a *SUBROUTINE*

`call` - invokes subroutine - pushes **`ip`**
`ret` - returns from subroutine - pops **`ip`**

- Uses MASM Directives at start/end of subroutine:
  `PROC` and **`ENDP`**
- Must Specify
  `NEAR` - intrasegment
  `FAR` - intersegment
- Difference is op-code that is used for of **`ret`**

`NEAR - c3h - pops IP`
`FAR - cbh - pops CS, pops IP`

`call` Instruction

- Differs from **`jmp`** Since Return Address is pushed on Stack

`NEAR` call: machine code: 3 bytes - 1 opcode,2 for **`IP`**
`IP is pushed on stack`
`FAR` call: 5 bytes - 1 opcode, 2 for **`IP`** and 2 for **`CS`**
`IP, CS pushed on stack`

- Typical use is to simply use 'call subroutine_name'
  call mysub
- **`call`** with operand - can use 16-bit offset in any register except segment registers

```
call     bx    ;pushes ip then jumps to cs:[bx]
```

Call Example

see example ‘subs.asm’ linked to WWW page.

Stack Overflow, Underflow

- If you keep pushing data on the stack without taking data off the stack, then the stack can eventually grow larger than your allocated space
  - Can begin writing to memory area that your code is in or other non-stack data
  - This is called stack OVERFLOW
- If you take off more data than you placed on the stack, then stack pointer can increment past the ‘start’ of the stack. This is stack UNDERFLOW.
- Bottom line: You should allocate sufficient memory for your stack needs, and pop off the same amout of data as pushed in.

Arrangement of Segments in .EXE file

3 segments: Code,stack,data

```
        .model small
        .586
        .stack 100h         ; 256 bytes of stack

        .data
oper_a    DW  12F7h
oper_b    DW  24FFh
sum       DW  0000h

        .code
START:  mov ax, @data  ;ax <-- data segment start address
        mov ds, ax    ;ds <-- initialize data segment register
        lea bx, oper_a  ;; get offset of oper_a
        mov ax,[bx]
        add ax,[bx+2]
        mov [bx+4],ax             ;store sum

        mov ax, 4c00h   ;ax <-- 4c DOS 21h program halt function
        int 21h      ;DOS service interrupt

END      START
```

Linker Map File

Linker *.map* file tells size of segment in terms of bytes

```
 Start  Stop   Length Name                   Class
 00000H 00015H 00016H _TEXT                 CODE
 00016H 0001BH 00006H _DATA                 DATA
 00020H 0011FH 00100H STACK                 STACK

 Origin   Group
 0001:0   DGROUP

Program entry point at 0000:0000
```

Code is 22 bytes long
Data is 6 bytes
Stack is 256 bytes

Segments in Memory from .exe file

The ordering of the segments in memory will be code, data, stack.

For this program, the .exe file is loaded with ordering:

code(CS=09B0, IP=0000
22 bytes long)

data(DS=09B1,
oper_a offset = 0006)

stack(SS=09B2, SP = 0100)

Note that SP is set to 'top' of stack data area. If stack grows larger than 256 bytes, will overwrite data first, then code.

Note that physical address of end of CODE = physical address of start of DATA
physical (CS:IP+016h) = 09B00 + 00000+ 016h = 09B16h
physical (DS:offset) = 09B10 +0006 = 09B16h