

ECE390

Computer Engineering II

Lecture 6



Dr. Zbigniew Kalbarczyk
University of Illinois at Urbana- Champaign

Lecture outline



- Program stack
- PUSH & POP instructions
- Procedures

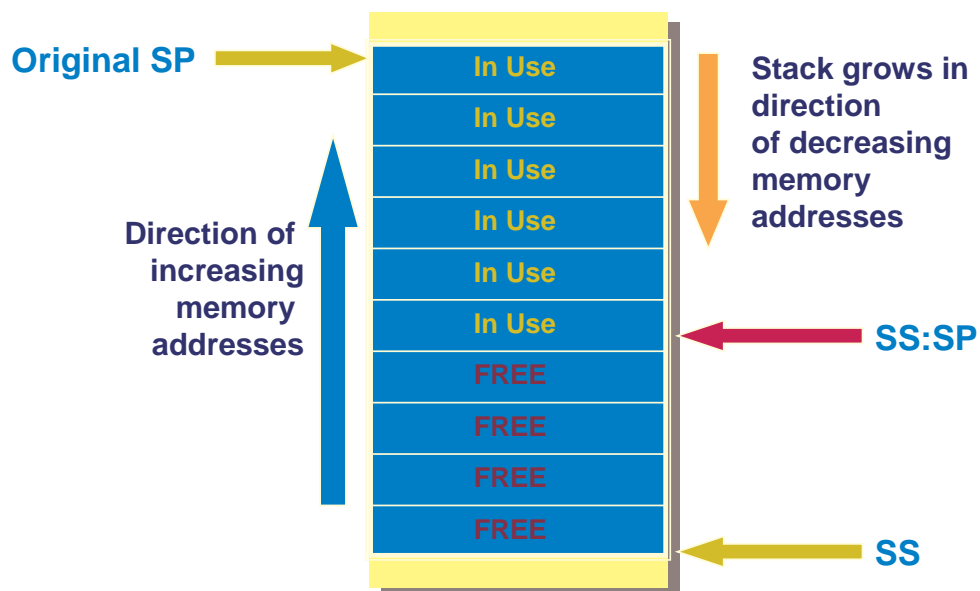
Stack Key Characteristics

- Stores a temporary data during program execution
- One point of access - the top of the stack
- A stack is always operated as Last-In-First-Out (LIFO) store, i.e., data are retrieved in the reverse order to which they were stored
- Instructions that directly manipulate the stack
 - **PUSH** - place element on top of stack
 - **POP** - remove element from top of stack

Z. Kalbarczyk

ECE390

Stack Implementation in Memory



Z. Kalbarczyk

ECE390

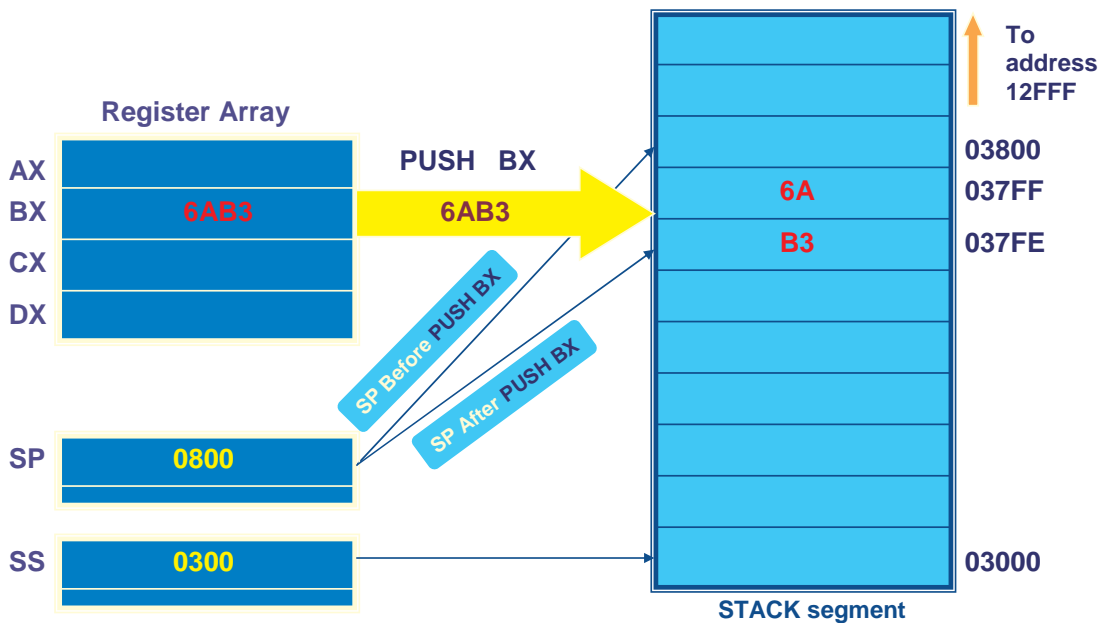
Stack Implementation in Memory (cont.)

- SS - Stack Segment
- SP (stack pointer) always points to the top of the stack
 - SP initially points to top of the stack (high memory address).
 - SP decreases as data is PUSHed
 - PUSH AX ==> SUB SP, 2 ; MOV [SS:SP], AX
 - SP increases as data is POPed
 - POP AX ==> MOV AX, [SS:SP] ; ADD SP, 2
- BP (base pointer) can point to any element on the stack

Z. Kalbarczyk

ECE390

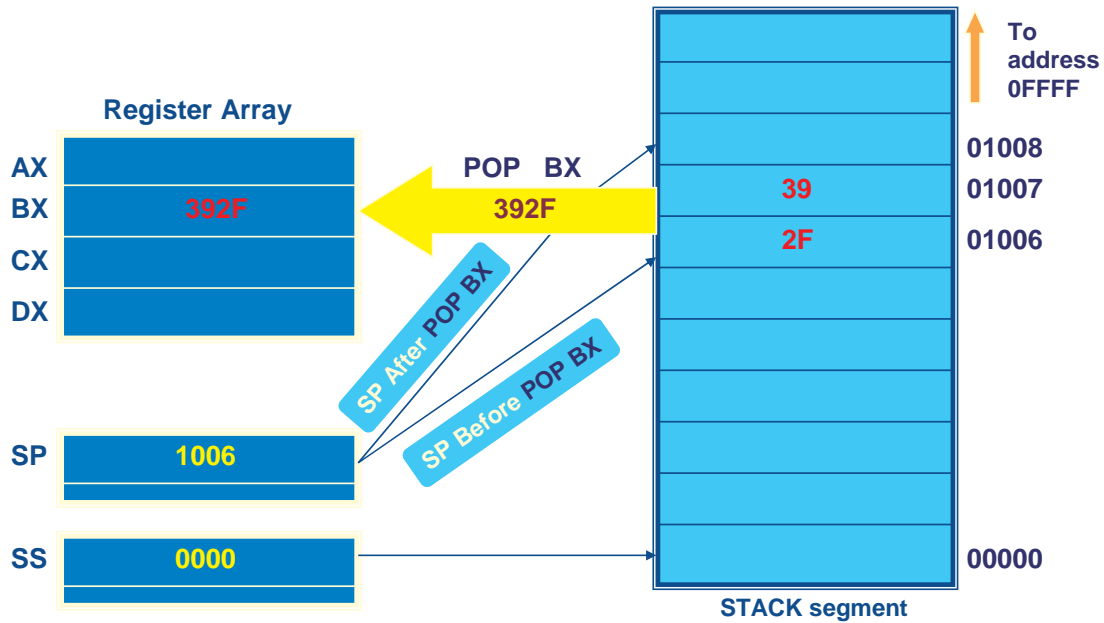
PUSH Instruction Example



Z. Kalbarczyk

ECE390

POP Instruction Example



Z. Kalbarczyk

ECE390

PUSH & POP (More I)

- PUSH and POP always store or retrieve words of data (never bytes)
- The 80386/80486 allow words or double words to be transferred to and from the stack
- The source of data for PUSH
 - any internal 16-bit/32-bit register, immediate data, any segment register, or any two bytes of memory data
- The POP places data into
 - internal register, segment register (except CS), or a memory location

Z. Kalbarczyk

ECE390

PUSH & POP

(More II)

- 80286 and later include PUSHA and POPA to store and retrieve the contents internal register set (AX, BX, CX, DX, SP, BP, SI, DI)
- Stack initialization, example:
 - Assume that the stack segment resides in memory locations 10000h – 1FFFFh
 - The stack segment is loaded with 1000h
 - The SP is loaded with 0000h
 - This makes the stack 64KB
 - First PUSH does 0000h – 0002h = FFFEh, storing data in 1FFFEh and 1FFFFh

Stack Use

- To store
 - Registers
 - Return address information while procedures are executing
 - Local variables that procedures may require
 - Dynamically allocated memory
- To pass
 - Parameters to procedures (function arguments)

Temporary Register Storage

- Push and Pop registers to preserve their value

Example:

```
push  ax           ; Place AX on the stack
push  bx           ; Place BX on the stack
...
< modify contents of registers AX & BX >
...
pop   bx           ; Restore original value of BX
pop   ax           ; Restore original value of AX
```

Temporary register storage

- Why would you want to backup and restore registers?
- Because registers are themselves temporary storage for instruction operands or memory addresses
- You might need to do a task that modifies registers, but you might then need the original contents of those registers later
- Any data that is an end result more than likely should go into a memory location (a variable)

Store Return Address of a Procedure

- `call proc_name`
 - Pushes the instruction pointer and sometimes the code segment register onto the stack
 - Performs an unconditional jump to the label `proc_name`
- `ret`
 - Pops saved IP and if necessary saved CS from the stack and back into the IP and CS registers
 - This causes the instruction following the call statement to be executed next

Procedures

- Procedures are chunks of code that usually perform specific frequently used tasks
- They can be repeatedly called from someplace else in your programs
- These are essentially the same thing as functions or subroutines in high-level languages
- Procedures may have inputs/outputs, both, either, neither
- Essentially just labeled blocks of assembly language instructions with a special return instruction at the end

Procedure example

```
..start
;here is my main program
mov ax, 10h
mov bx, 20h
mov cx, 30h
mov dx, 40h
call AddRegs      ;this proc performs AX + BX + CX + DX à AX

call DosExit

;-----
AddRegs
    add ax, bx
    add ax, cx
    add ax, dx
ret
```

Z. Kalbarczyk

ECE390

Proc prime directive

- *Procs* should never alter the contents of any register that the procedure is not explicitly supposed to modify
- If for example a *proc* is supposed to return a value in AX, it is not only okay but required for it to modify AX
- No other registers should be left changed
- *Push* registers onto the stack at the beginning and *pop* the registers at the end

Z. Kalbarczyk

ECE390

Call

- *Call* does two things
 - It pushes the address of the instruction immediately following the call statement onto the stack
 - It loads the instruction pointer with the value of the label marking the beginning of the procedure you're calling (this is essentially an unconditional jump to the beginning of the procedure)

Two kinds of calls

- Near calls
 - Allow jumps to procedures within the same code segment
 - In this case, only the instruction pointer gets pushed onto the stack
- Far calls
 - Allow jumps to anywhere in the memory space
 - In this case, both the contents of the CS and IP registers get pushed onto the stack

Passing parameters

- Using registers
 - Registers are the ultimate global variables
 - Your *proc* can simply access information the calling program placed in specific registers
 - Also a simple way for your *proc* to send data back to the caller

Passing parameters

- Using global memory locations
 - Memory locations declared at the beginning of your program are global and can be accessed from any procedure
- Using a parameter block
 - You may pass a pointer to an array or other type of memory block instead of an individual data value

Passing parameters

- Using the stack
 - Caller pushes all arguments expected by the proc onto the stack
 - *Proc* can access these arguments directly from the stack by setting up the *stack frame*.

```
    push bp      ; save value of bp
    mov  bp, sp  ; mark start of stack frame
arg1 is at [ss:bp+4] assuming call
arg1 is at [ss:bp+6] assuming call far
```

Example

```
;call proc to calculate area of right triangle.
;proc expects two word sized args to be on the stack
push 3
push 4
call TriangleArea

;now we must remove the variables from the stack
;every push must be popped.
add sp, 4

; ax now contains 6
```

Example continued

```
TriangleArea
  push bp
  mov  bp, sp
  push dx
  mov  ax, [bp + 4]
  mul  word [bp + 6]
  shr  ax, 1      ;divide by two
  pop  dx
  pop  bp
  ret
```

Z. Kalbarczyk

ECE390

What just happened...

```
Push 3
Push 4
Call TriangleArea
```

```
TriangleArea
  push bp
  mov  bp, sp
  push dx
  mov  ax, [bp+4]
  mul  word [bp+6]
  shr  ax, 1
  pop  dx
  pop  bp
  ret
```

```
Add sp, 4
```

	SP	E	
Stack frame	SP	C	0003h
	SP	A	0004h
	SP	8	Return IP
	SP	6	Saved BP
	SP	4	Saved DX
			2
SS:0000		0	

BP

Z. Kalbarczyk

ECE390