Computer Org. & Assembly Language Lab

Lab#06: Procedures

Agenda

- Stack and Operations
- PUSH instruction
- POP instruction
- PUSHFD & POPFD instructions
- PUSHAD, PUSHA & POPAD, POPA instructions
- Procedures
  - Local and Global variables
  - Passing Parameters to Procedures
  - Uses Operator

Stack and Operations

A stack data structure follows LIFO (Last In First Out). Generically it has only two operations

- **Push:** it add an element on the top of stack
- **Pop:** it removes/deletes the top most element of the stack

PUSH and POP Instructions

Eight types of PUSH/POP instructions are used in assembly.

1.1 PUSH	1.2. PUSHAD	1.3. PUSHA	1.4. PUSHFD
2.1 POP	2.2. POPAD	2.3. POPA	1.4.POPFD

PUSH Instruction

This instructions first decrements ESP and then copies a 16/32 – Bit source operand into stack. A 16 – Bit operand causes ESP to be decremented by 2 and likewise 4 for 32 – Bit Operand. Syntax is given below.

```
PUSH r/m16
PUSH r/m32
PUSH imm32
```

POP Instruction

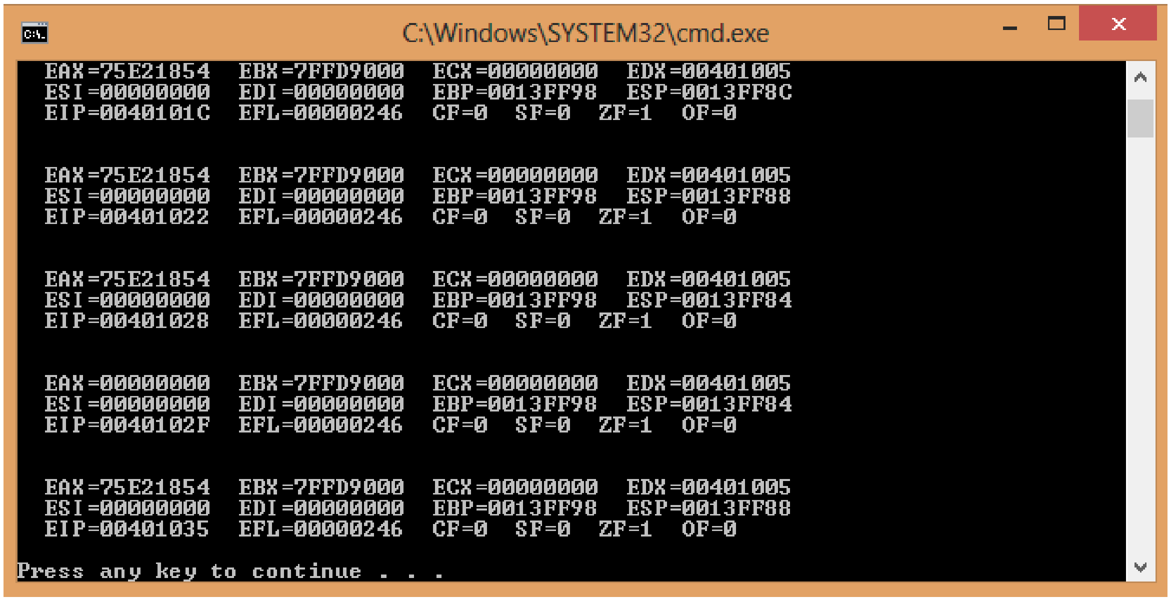
This instructions first copies the contents of the stack element pointed to by ESP into 16/32 – Bit destination operand and then increments ESP. A 16 – Bit operand causes ESP to be incremented by 2 and likewise 4 for 32 – Bit Operand. General syntax of use is given below.

```
POP r/m16
POP r/m32
```

Using PUSH and POP

```
Include Irvine32.inc

.code
main PROC
        call DumpRegs
        push 1
        call DumpRegs

        push eax
        call DumpRegs

        push eax
        call DumpRegs

        xor eax,eax
        call DumpRegs

        pop eax
        call DumpRegs

        exit
main ENDP
END main
```

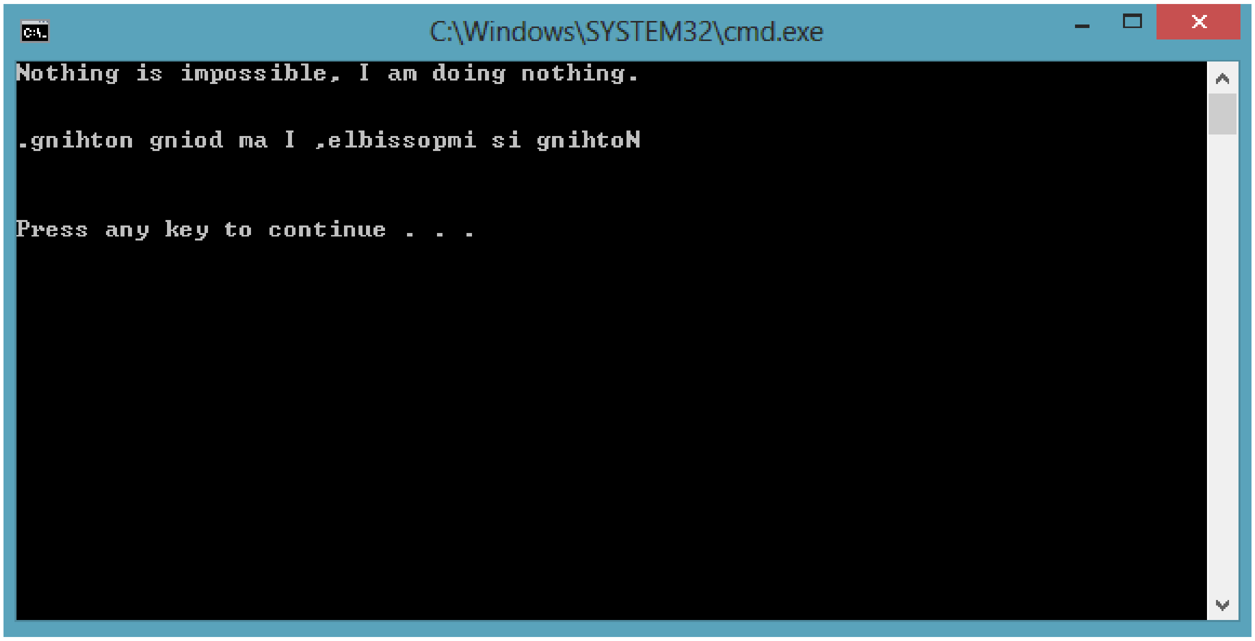
Output



```
  EAX=75E21854  EBX=7FFD9000  ECX=00000000  EDX=00401005
  ESI=00000000  EDI=00000000  EBP=0013FF98  ESP=0013FF8C
  EIP=0040101C  EFL=00000246  CF=0  SF=0  ZF=1  OF=0


  EAX=75E21854  EBX=7FFD9000  ECX=00000000  EDX=00401005
  ESI=00000000  EDI=00000000  EBP=0013FF98  ESP=0013FF88
  EIP=00401022  EFL=00000246  CF=0  SF=0  ZF=1  OF=0


  EAX=75E21854  EBX=7FFD9000  ECX=00000000  EDX=00401005
  ESI=00000000  EDI=00000000  EBP=0013FF98  ESP=0013FF84
  EIP=00401028  EFL=00000246  CF=0  SF=0  ZF=1  OF=0


  EAX=00000000  EBX=7FFD9000  ECX=00000000  EDX=00401005
  ESI=00000000  EDI=00000000  EBP=0013FF98  ESP=0013FF84
  EIP=0040102F  EFL=00000246  CF=0  SF=0  ZF=1  OF=0


  EAX=75E21854  EBX=7FFD9000  ECX=00000000  EDX=00401005
  ESI=00000000  EDI=00000000  EBP=0013FF98  ESP=0013FF88
  EIP=00401035  EFL=00000246  CF=0  SF=0  ZF=1  OF=0

Press any key to continue . . .
```

```
INCLUDE Irvine32.inc;

.data
Msg1 BYTE "Nothing is impossible, I am doing nothing.",0

.code
main PROC

        mov edx,OFFSET Msg1
        call WriteString
        call Crlf
        call Crlf
        call Crlf

        mov ecx,lengthof Msg1
        dec ecx                         ;to remove the null character's length from string length
        mov esi,0

        Labl1:
                movzx eax,Msg1[esi]     ; get char by char
                push eax                ;push on stack
                inc esi
        loop Labl1

        XOR ESI,ESI
        mov ecx,lengthof Msg1
        dec ecx

        Labl2:
                pop eax
                mov Msg1[esi],al
                inc esi
        loop Labl2

        mov edx,OFFSET Msg1
        call WriteString
        call Crlf
        call Crlf
        call Crlf
        call Crlf

exit
main ENDP
END main
```

Output



```
Nothing is impossible, I am doing nothing.

.gnihton gniod ma I ,elbissopmi si gnihtoN


Press any key to continue . . .
```

PUSHFD & POPFD Instruction

PUSHFD: pushes 32 – Bit EFL register on the stack.

POPFD: pops – 32 Bit EFL register from the stack into EFLAG.

```
pushfd
popfdf
```

A sample is given below to show how the contents of flags are saved and restored.

```
; copy this code into main procedure
mov al,0
pushfd          ;save the current flags
call dumpregs

inc al          ;change in flag(s)
call DumpRegs

popfd           ;it restores the flags as it was before computation
call DumpRegs
```

PUSHAD, PUSHA & POPAD, POPA Instruction

PUSHAD: pushes all 32 – Bit general purpose registers on the stack in the following order

EAX, ECX, EDX, EBX, ESP, EBP, ESI and EDI
POPAD: pops – 32 Bit registers in the reverse order as by PUSHAD

PUSHA: pushes all 16 – Bit registers in the order. AX, CX, DX, BX, SP, BP, SI and DI
POPA: pops – 16 Bit registers in the reverse order as by PUSHA

```
PUSHAD ; 32 – Bit Registers
POPAD

PUSHA ; 16 – Bit Registers
POPHA
```

```
TITLE Instructions PUSHAD,POPAD

; copy this code into .data section
Msg1 BYTE "for 32 - bit registers",0

; copy this code into main procedure
; 32 - bit registers
        mov edx,OFFSET Msg1
        call WriteString
        call Crlf

        XOR EAX,EAX
        XOR EBX,EBX
        XOR ECX,ECX
        XOR EDX,EDX
        call DumpRegs

        pushad
        mov eax,12345678h
        mov ebx,5678h
        mov ecx,1234h
        mov edx,1359h
        call DumpRegs
        popad

        call DumpRegs
```

```
TITLE Instructions PUSHA, POPA

; copy this code into .data section as variable declaration section
Msg2 BYTE "for 16 - bit registers",0

; copy this code into main procedure
;16 - bit registers
            mov edx,OFFSET Msg2
            call WriteString
            call Crlf

            XOR EAX,EAX
            XOR EBX,EBX
            XOR ECX,ECX
            XOR EDX,EDX
            call DumpRegs

            pusha
            mov ax,1234h
            mov bx,5678h
            mov cx,1357h
            mov dx,2468h
            call DumpRegs
            popa

            call DumpRegs
```

Procedures

A Procedure is a named block of statements that ends in a return statement. It is good programming practice to divide your program into procedures. In assembly PROC and ENDP Directives are used for procedures.

Following is an assembly language procedure named sample:

```
sample PROC
.
.
.
ret
sample ENDP
```

Adding 3 Numbers

```
INCLUDE Irvine32.inc;
.data
.code
main PROC
        mov eax,12d
        mov ebx,228d
        mov ecx,10d

        call sum

        call WriteInt

        call Crlf
exit
main ENDP
        sum proc
            add eax,ebx
            add eax,ecx
            ret
        sum endp
END main
```

Note

The CALL instruction calls a procedure

- pushes offset of next instruction on the stack
- copies the address of the called procedure into EIP

The RET instruction returns from a procedure

- pops top of stack into EIP

What about nested procedure calls?

Local and Global Labels

A local label is visible only to statements inside the same procedure. A global label is visible everywhere.

```
main PROC
        jmp L2          ; error
        L1::            ; global label
        exit
main ENDP


sub2 PROC
```

```
        L2:             ; local label
        jmp L1          ; ok
        ret

sub2 ENDP
```

Passing Parameters to Procedures

An example of summation.

The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names:

```
ArraySum PROC
mov esi,0                       ; array index
mov eax,0                       ; set the sum to zero
mov ecx,LENGTHOF myarray        ; set number of elements


L1:     add eax,myArray[esi]    ; add each integer to sum
        add esi,4               ; point to next integer
loop L1                         ; repeat for array size
mov theSum,eax                  ; store the sum
ret

ArraySum ENDP
```

Alternatively

This version of ArraySum returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

```
ArraySum PROC
; Receives: ESI points to an array of doublewords,
;   ECX = number of array elements.
; Returns: EAX = sum
;-----------------------------------------------------
mov eax,0               ; set the sum to zero

L1:     add eax,[esi]   ; add each integer to sum
        add esi,4       ; point to next integer
loop L1                 ; repeat for array size
ret

ArraySum ENDP
```

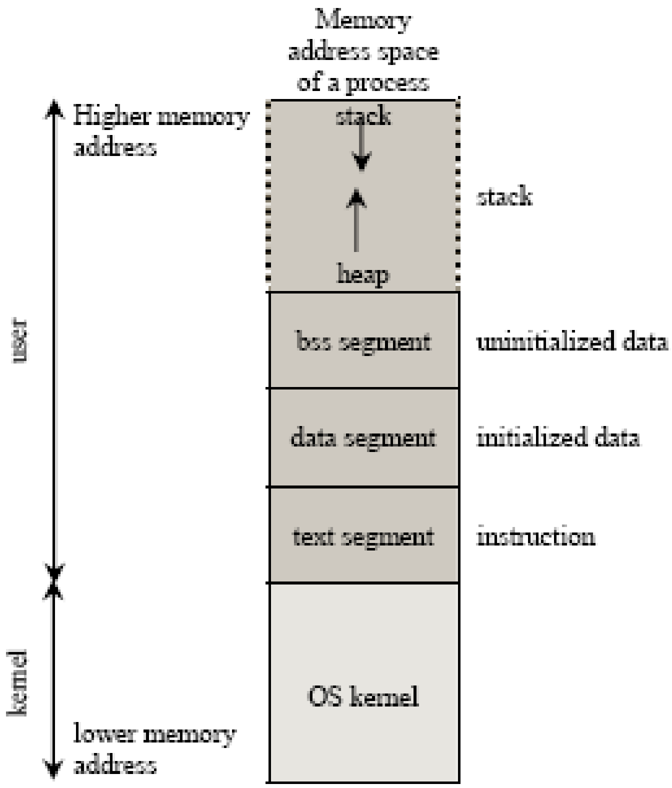
USES Operator

```
INCLUDE Irvine32.inc;
.data

.code
main PROC
        call dumpregs
        call sample
        call dumpregs
exit
main ENDP
sample PROC USES esi ecx
        mov esi, 12345678h
        mov ecx, 87654321h
        call dumpregs
        ret
sample ENDP

END main
```

The code shown in red is automatically generated

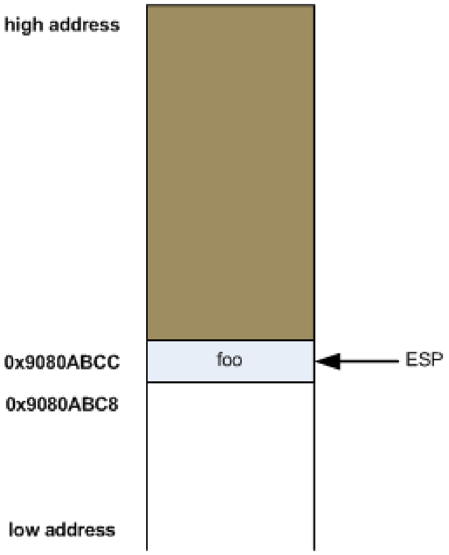
```
sample PROC
        push esi
        push ecx
        .
        .
        pop ecx
        pop esi
        ret
sample ENDP
```

Memory allocation for a process

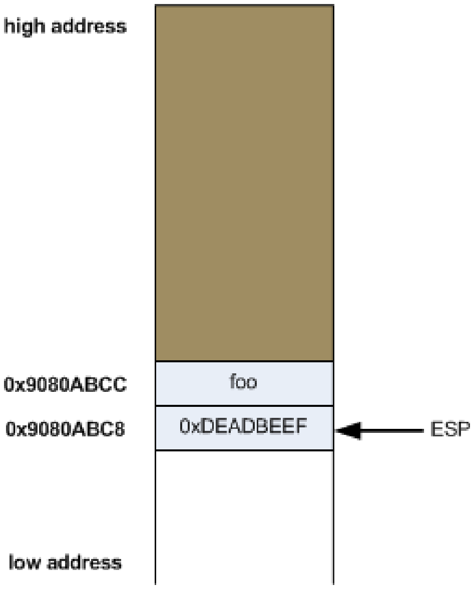


Stack Grows Downward in Memory

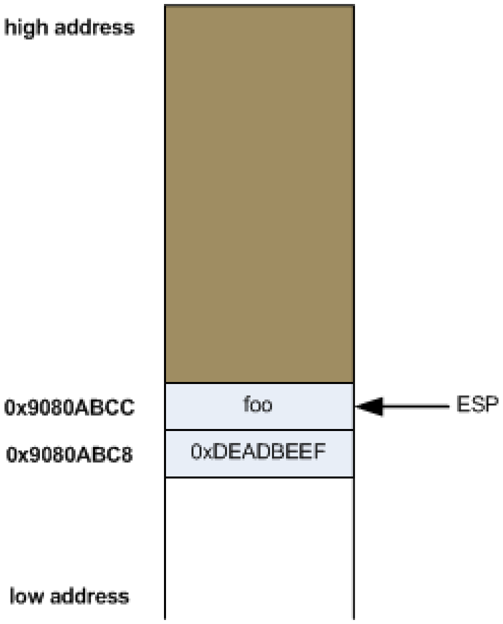
(a)



(b)



(b)


high address
0x9080ABCC
foo
ESP
0x9080ABC8
0xDEADBEEF
low address