Stack and Procedures

Computer Organization
&
Assembly Language Programming

Dr Adnan Gutub

aagutub ‘at’ uqu.edu.sa

[Adapted from slides of Dr. Kip Irvine: Assembly Language for Intel-Based Computers]
Most Slides contents have been arranged by Dr Muhamed Mudawar & Dr Aiman El-Maleh from Computer Engineering Dept. at KFUPM

Presentation Outline

- ❖ Runtime Stack
- ❖ Stack Operations
- ❖ Defining and Using Procedures
- ❖ Program Design Using Procedures

What is a Stack?

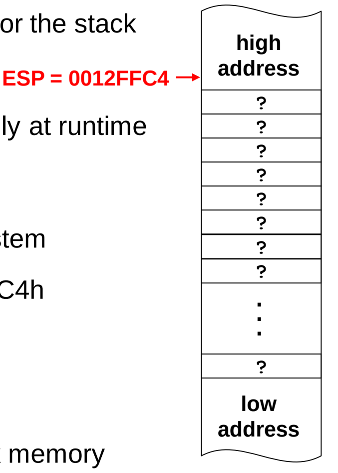
- Stack is a **Last-In-First-Out** (LIFO) data structure
  - Analogous to a stack of plates in a cafeteria
  - Plate on **Top of Stack** is directly accessible
- Two basic stack operations
  - **Push**: inserts a new element on top of the stack
  - **Pop**: deletes top element from the stack
- View the stack as a linear array of elements
  - Insertion and deletion is restricted to one end of array
- Stack has a maximum capacity
  - When stack is **full**, no element can be pushed
  - When stack is **empty**, no element can be popped

Runtime Stack

- **Runtime stack:** array of consecutive memory locations
- Managed by the processor using two registers
  - Stack Segment register **SS**
    - Not modified in protected mode, **SS** points to segment descriptor
  - Stack Pointer register **ESP**
    - For 16-bit real-address mode programs, **SP** register is used
- **ESP** register points to the **top of stack**
  - Always points to last data item placed on the stack
- Only words and doublewords can be pushed and popped
  - But not single bytes
- Stack grows **downward** toward lower memory addresses

Runtime Stack Allocation

- **.STACK** directive specifies a runtime stack
  - Operating system allocates memory for the stack
  - Runtime stack is initially empty
  - The stack size can change dynamically at runtime
- Stack pointer **ESP**
  - **ESP** is initialized by the operating system
  - Typical initial value of **ESP** = 0012FFC4h
- The stack grows **downwards**
  - The memory below **ESP** is free
  - **ESP** is decremented to allocate stack memory



Presentation Outline

- Runtime Stack
- Stack Operations
- Defining and Using Procedures
- Program Design Using Procedures

Stack Instructions

- Two basic stack instructions:
  - `push source`
  - `pop  destination`
- **`Source`** can be a word (16 bits) or doubleword (32 bits)
  - General-purpose register
  - Segment register: CS, DS, SS, ES, FS, GS
  - Memory operand, memory-to-stack transfer is allowed
  - Immediate value
- **`Destination`** can be also a word or doubleword
  - General-purpose register
  - Segment register, except that **`pop CS`** is NOT allowed
  - Memory, stack-to-memory transfer is allowed

Push Instruction

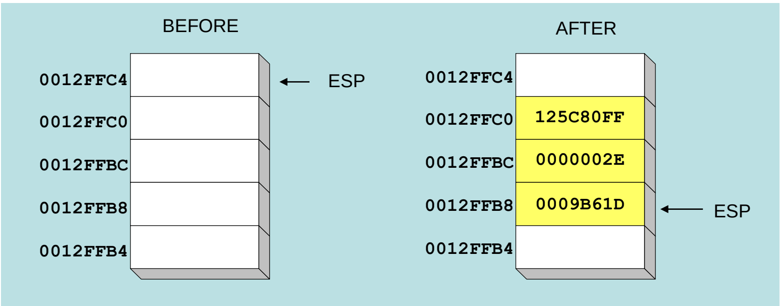
- **`Push source32 (r/m32 or imm32)`**
  - **ESP** is first decremented by **4**
    - **ESP = ESP – 4** (stack grows by 4 bytes)
  - 32-bit source is then copied onto the stack at the new **ESP**
    - **[ESP] = source32**
- **`Push source16 (r/m16)`**
  - **ESP** is first decremented by **2**
    - **ESP = ESP – 2** (stack grows by 2 bytes)
  - 16-bit source is then copied on top of stack at the new **ESP**
    - **[ESP] = source16**
- Operating system puts a limit on the stack capacity
  - **Push** can cause a **Stack Overflow** (stack **cannot grow**)

Examples on the Push Instruction

❖ Suppose we execute:
- ✧ PUSH EAX ; EAX = 125C80FFh
- ✧ PUSH EBX ; EBX = 2Eh
- ✧ PUSH ECX ; ECX = 9B61Dh

The stack grows downwards

The area below ESP is free



Pop Instruction

❖ **Pop** **`dest32 (r/m32)`**
- ✧ 32-bit doubleword at ESP is first copied into dest32
  - **dest32 = [ESP]**
- ✧ ESP is then incremented by 4
  - **ESP = ESP + 4** (stack shrinks by 4 bytes)

❖ **Pop** **`dest16 (r/m16)`**
- ✧ 16-bit word at ESP is first copied into dest16
  - **dest16 = [ESP]**
- ✧ ESP is then incremented by 2
  - **ESP = ESP + 2** (stack shrinks by 2 bytes)

❖ Popping from an empty stack causes a **stack underflow**

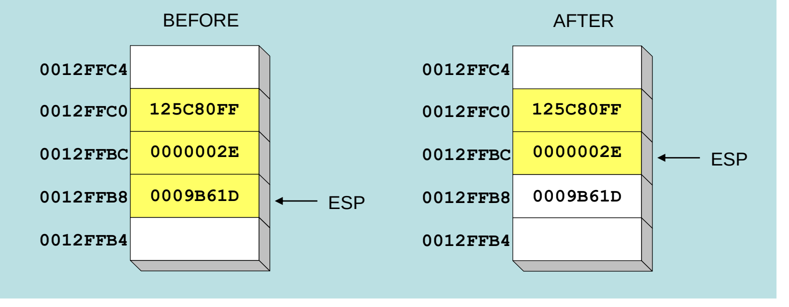
Examples on the Pop Instruction

❖ Suppose we execute:

- ✧ POP SI ; SI = B61Dh
- ✧ POP DI ; DI = 0009h

The stack shrinks upwards

The area at & above ESP is allocated



Uses of the Runtime Stack

❖ Runtime Stack can be utilized for

- ✧ Temporary storage of data and registers
- ✧ Transfer of program control in procedures and interrupts
- ✧ Parameter passing during a procedure call
- ✧ Allocating local variables used inside procedures

❖ Stack can be used as temporary storage of data

- ✧ Example: exchanging two variables in a data segment

```
push var1  ; var1 is pushed
push var2  ; var2 is pushed
pop  var1  ; var1 = var2 on stack
pop  var2  ; var2 = var1 on stack
```

Temporary Storage of Registers

- Stack is often used to free a set of registers

```
push EBX      ; save EBX
push ECX      ; save ECX
. . .
; EBX and ECX can now be modified
. . .
pop  ECX      ; restore ECX first, then
pop  EBX      ; restore EBX
```

- Example on moving DX:AX into EBX

```
push DX       ; push most significant word first
push AX       ; then push least significant word
pop  EBX      ; EBX = DX:AX
```


Example: Nested Loop

When writing a nested loop, push the outer loop counter ECX before entering the inner loop, and restore ECX after exiting the inner loop and before repeating the outer loop

```
    mov   ecx, 100      ; set outer loop count
L1: . . .               ; begin the outer loop
    push ecx            ; save outer loop count

    mov   ecx, 20       ; set inner loop count
L2: . . .               ; begin the inner loop
    . . .               ; inner loop
    loop L2             ; repeat the inner loop

    . . .               ; outer loop
    pop   ecx           ; restore outer loop count
    loop L1             ; repeat the outer loop
```

Push/Pop All Registers

- ❖ **pushad**
  - ✧ Pushes all the 32-bit general-purpose registers
  - ✧ EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI in this order
  - ✧ Initial ESP value (before pushad) is pushed
  - ✧ ESP = ESP – 32
- ❖ **pusha**
  - ✧ Same as pushad but pushes all 16-bit registers AX through DI
  - ✧ ESP = ESP – 16
- ❖ **popad**
  - ✧ Pops into registers EDI through EAX in reverse order of pushad
  - ✧ ESP is not read from stack. It is computed as: ESP = ESP + 32
- ❖ **popa**
  - ✧ Same as popad but pops into 16-bit registers. ESP = ESP + 16

Stack Instructions on Flags

- ❖ Special Stack instructions for pushing and popping flags
  - ✧ **pushfd**
    - Push the 32-bit EFLAGS
  - ✧ **popfd**
    - Pop the 32-bit EFLAGS
- ❖ No operands are required
- ❖ Useful for saving and restoring the flags
- ❖ For 16-bit programs use **pushf** and **popf**
  - ✧ Push and Pop the 16-bit FLAG register

Next . . .

- ❖ Runtime Stack
- ❖ Stack Operations
- ❖ Defining and Using Procedures
- ❖ Program Design Using Procedures

Procedures

- ❖ A **procedure** is a logically self-contained unit of code
  - ✧ Called sometimes a **function**, **subprogram**, or **subroutine**
  - ✧ Receives a **list of parameters**, also called **arguments**
  - ✧ Performs computation and returns results
  - ✧ Plays an important role in modular program development
- ❖ Example of a procedure (called function) in C language

```
int sumof ( int x,int y,int z ) {
    int temp;
    temp = x + y + z;
    return temp;
}
```

Result type

Formal parameter list

Return function result

- ❖ The above function sumof can be called as follows:

```
sum = sumof( num1,num2,num3 );
```

Actual parameter list

Defining a Procedure in Assembly

- Assembler provides two directives to define procedures
  - **PROC** to define name of procedure and mark its beginning
  - **ENDP** to mark end of procedure
- A typical procedure definition is

```
procedure_name   PROC
        . . .
    ; procedure body
        . . .
procedure_name   ENDP
```

- `procedure_name` should match in PROC and ENDP

Documenting Procedures

- Suggested Documentation for Each Procedure:
  - **Does:** Describe the task accomplished by the procedure
  - **Receives**: Describe the input parameters
  - **Returns**: Describe the values returned by the procedure
  - **Requires**: List of requirements called **preconditions**
- Preconditions
  - Must be satisfied **before** the procedure is called
  - If a procedure is called without its preconditions satisfied, it will probably not produce the expected output

Example of a Procedure Definition

- The `sumof` procedure receives three integer parameters
  - Assumed to be in EAX, EBX, and ECX
  - Computes and returns result in register EAX

```
;------------------------------------------------
; Sumof:    Calculates the sum of three integers
; Receives: EAX, EBX, ECX, the three integers
; Returns:  EAX = sum
; Requires: nothing
;------------------------------------------------
sumof PROC
   add  EAX, EBX          ; EAX = EAX + second number
   add  EAX, ECX          ; EAX = EAX + third  number
   ret                    ; return to caller
sumof ENDP
```

- The `ret` instruction returns control to the caller

The Call Instruction

- To invoke a procedure, the `call` instruction is used
- The `call` instruction has the following format

```
call  procedure_name
```

- Example on calling the procedure `sumof`
  - Caller passes actual parameters in EAX, EBX, and ECX
  - Before calling procedure `sumof`

```
mov   EAX, num1      ; pass first  parameter in EAX
mov   EBX, num2      ; pass second parameter in EBX
mov   ECX, num3      ; pass third  parameter in ECX
call  sumof          ; result is in EAX
mov   sum, EAX       ; save result in variable sum
```

- `call sumof` will call the procedure `sumof`

How a Procedure Call / Return Works

- How does a procedure know where to return?
  - There can be multiple calls to same procedure in a program
  - Procedure has to return differently for different calls
- It knows by saving the **return address** (RA) on the stack
  - This is the **address of next instruction** after **call**
- The **call** instruction does the following
  - Pushes the **return address** on the stack
  - Jumps into the first instruction inside procedure
  - **ESP = ESP – 4; [ESP] = RA; EIP = procedure address**
- The **ret** (return) instruction does the following
  - Pops return address from stack
  - Jumps to return address: **EIP = [ESP]; ESP = ESP + 4**

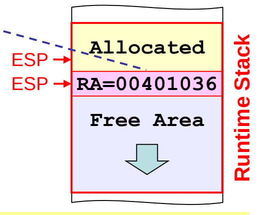
Details of CALL and Return

Address	Machine Code	Assembly Language
		`.CODE`
		`main PROC`
00401020	A1 00405000	`mov EAX, num1`
00401025	8B 1D 00405004	`mov EBX, num2`
0040102B	8B 0D 00405008	`mov ECX, num3`
00401031	**E8** 0000004B	`call sumof`
00401036	A3 0040500C	`mov sum, EAX`
. . .	. . .	`. . .`
		`exit`
		`main ENDP`
		`sumof PROC`
00401081	03 C3	`add EAX, EBX`
00401083	03 C1	`add EAX, ECX`
00401085	C3	`ret`
		`sumof ENDP`
		`END main`

IP-relative call

```
EIP = 00401036
    + 0000004B
EIP = 00401081
```

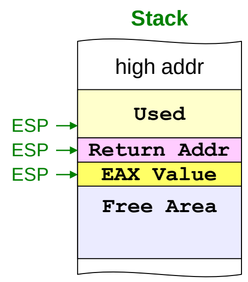
Before Call ESP = 0012FFC4
After Call ESP = 0012FFC0
After Ret (Return) ESP = 0012FFC4



Don't Mess Up the Stack !

- Just before returning from a procedure
  - Make sure the stack pointer ESP is pointing at return address
- Example of a messed-up procedure
  - Pushes EAX on the stack before returning
  - Stack pointer ESP is NOT pointing at return address!

```
main PROC
    call  messedup
    . . .
    exit
main ENDP
messedup PROC
    push  EAX
    ret
messedup ENDP
```



Where to return?

EAX value is NOT the return address!

Nested Procedure Calls

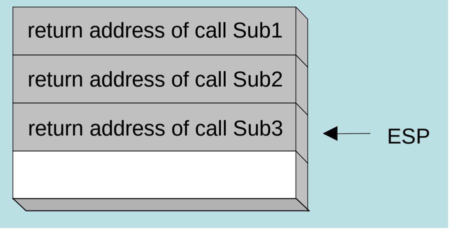
```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```

By the time Sub3 is called, the stack contains all three return addresses



Parameter Passing

- Parameter passing in assembly language is different
  - More complicated than that used in a high-level language
- In assembly language
  - Place all required parameters in an accessible storage area
  - Then call the procedure
- Two types of storage areas used
  - Registers: general-purpose registers are used (register method)
  - Memory: stack is used (stack method)
- Two common mechanisms of parameter passing
  - Pass-by-value: parameter **value** is passed
  - Pass-by-reference: **address** of parameter is passed

Passing Parameters in Registers

```
;-----------------------------------------------------
; ArraySum: Computes the sum of an array of integers
; Receives: ESI = pointer to an array of doublewords
;           ECX = number of array elements
; Returns:  EAX = sum
;-----------------------------------------------------
ArraySum PROC
   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
```

ESI: **Reference** parameter = array address

ECX: **Value** parameter = count of array elements

Preserving Registers

- Need to preserve the registers across a procedure call
  - Stack can be used to preserve register values
- Which registers should be saved?
  - Those registers that are modified by the called procedure
    - But still used by the calling procedure
  - We can save all registers using **`pusha`** if we need most of them
    - However, better to save only needed registers when they are few
- Who should preserve the registers?
  - Calling procedure: saves and frees registers that it uses
    - Registers are saved before procedure call and restored after return
  - Called procedure: **preferred method** for modular code
    - Register preservation is done in one place only (inside procedure)

Example on Preserving Registers

```
;-----------------------------------------------------
; ArraySum: Computes the sum of an array of integers
; Receives: ESI = pointer to an array of doublewords
;           ECX = number of array elements
; Returns:  EAX = sum
;-----------------------------------------------------
ArraySum PROC
    push esi                ; save esi, it is modified
    push ecx                ; save ecx, it is modified
    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
    pop  ecx                ; restore registers
    pop  esi                ; in reverse order
    ret
ArraySum ENDP
```

No need to save EAX. Why?

USES Operator

- The **USES** operator simplifies the writing of a procedure
  - Registers are frequently modified by procedures
  - Just list the registers that should be preserved after **USES**
  - Assembler will **generate** the **push** and **pop** instructions

```
ArraySum PROC USES esi ecx
    mov   eax,0
L1: add   eax, [esi]
    add   esi, 4
    loop L1
    ret
ArraySum ENDP
```

```
ArraySum PROC
    push esi
    push ecx
    mov   eax,0
L1: add   eax, [esi]
    add   esi, 4
    loop L1
    pop   ecx
    pop   esi
    ret
ArraySum ENDP
```


Next . . .

- Runtime Stack
- Stack Operations
- Defining and Using Procedures
- Program Design Using Procedures

Program Design using Procedures

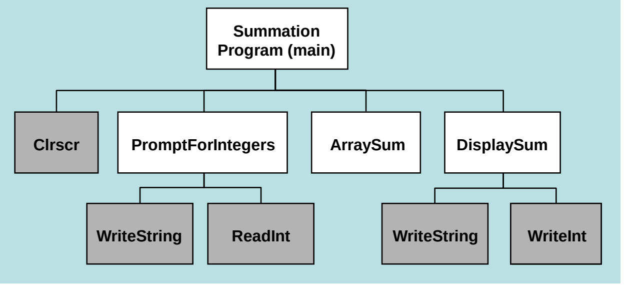
❖ Program Design involves the Following:
- ✧ Break large tasks into smaller ones
- ✧ Use a hierarchical structure based on procedure calls
- ✧ Test individual procedures separately

Integer Summation Program:

Write a program that prompts the user for multiple 32-bit integers, stores them in an array, calculates the array sum, and displays the sum on the screen.

Main steps:
1. Prompt user for multiple integers
2. Calculate the sum of the array
3. Display the sum

Structure Chart



Structure Chart

Above diagram is called a **structure chart**

Describes program structure, division into procedure, and call sequence

Link library procedures are shown in grey

Integer Summation Program – 1 of 4

```
INCLUDE Irvine32.inc

ArraySize EQU 5

.DATA
   prompt1 BYTE  "Enter a signed integer: ",0
   prompt2 BYTE  "The sum of the integers is: ",0
   array   DWORD  ArraySize  DUP(?)

.CODE
main PROC
   call Clrscr                ; clear the screen
   mov  esi, OFFSET array
   mov  ecx, ArraySize
   call PromptForIntegers     ; store input integers in array
   call ArraySum              ; calculate the sum of array
   call DisplaySum            ; display the sum
   exit
main ENDP
```


Integer Summation Program – 2 of 4

```
;-----------------------------------------------------
; PromptForIntegers: Read input integers from the user
; Receives: ESI = pointer to the array
;           ECX = array size
; Returns:  Fills the array with the user input
;-----------------------------------------------------
PromptForIntegers PROC USES ecx edx esi
   mov  edx, OFFSET prompt1
L1:
   call WriteString           ; display prompt1
   call ReadInt               ; read integer into EAX
   call Crlf                  ; go to next output line
   mov  [esi], eax            ; store integer in array
   add  esi, 4                ; advance array pointer
   loop L1

   ret
PromptForIntegers ENDP
```

Integer Summation Program – 3 of 4

```
;-----------------------------------------------------
; ArraySum: Calculates the sum of an array of integers
; Receives: ESI = pointer to the array,
;           ECX = array size
; Returns:  EAX = sum of the array elements
;-----------------------------------------------------
ArraySum PROC USES esi ecx
  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                    ; sum is in EAX
ArraySum ENDP
```


Integer Summation Program – 4 of 4

```
;-----------------------------------------------------
; DisplaySum: Displays the sum on the screen
; Receives:   EAX = the sum
; Returns:    nothing
;-----------------------------------------------------
DisplaySum PROC
   mov  edx, OFFSET prompt2
   call WriteString             ; display prompt2
   call WriteInt                ; display sum in EAX
   call Crlf
   ret
DisplaySum ENDP
END main
```

Sample Output

```
Enter a signed integer:  550
Enter a signed integer:  -23
Enter a signed integer:  -96
Enter a signed integer:   20
Enter a signed integer:    7
The sum of the integers is:  +458
```


Parameter Passing Through Stack

❖ Parameters can be saved on the stack before a procedure is called.

❖ The called procedure can easily access the parameters using either the ESP or EBP registers without altering ESP register.

❖ Example

Suppose you want to implement the following pseudo-code:

```
i = 25;
 j = 4;
Test(i, j, 1);
```

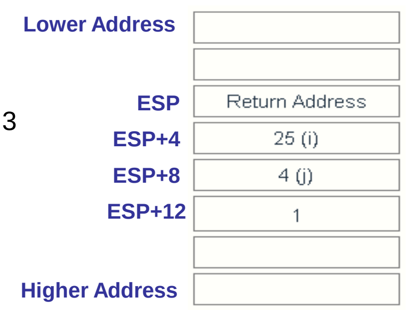
Then, the assembly language code fragment looks like:

```
mov i, 25
mov j, 4
push 1
push j
push i
call Test
```

Parameter Passing Through Stack

Example: Accessing parameters on the stack

```
Test  PROC
  mov AX, [ESP + 4] ;get i
  add AX, [ESP + 8] ;add j
  sub AX, [ESP + 12] ;subtract parm 3
                     (1) from sum
  ret
Test ENDP
```



Call & Return Instructions

Instruction	Operand	Note
CALL	label name	Push IP IP= IP + displacement relative to next instruction
CALL	r/m	Push IP IP = [r/m]
CALL	label name (FAR)	Push CS Push IP CS:IP=address of label name
CALL	m (FAR)	Push CS Push IP CS:IP= [m]
RET		Pop IP
RET	imm	Pop IP SP = SP + imm
RET	(FAR)	Pop IP Pop CS
RET	imm (FAR)	Pop IP Pop CS SP = SP + imm

Freeing Passed Parameters From Stack

❖ Use RET N instruction to free parameters from stack

Example: Accessing parameters on the stack

```
Test  PROC
  mov AX, [ESP + 4] ;get i
  add AX, [ESP + 8] ;add j
  sub AX, [ESP + 12] ;subtract parm. 3
                      (1) from sum
  ret 12
Test ENDP
```


Local Variables

❖ Local variables are dynamic data whose values must be preserved over the lifetime of the procedure, but not beyond its termination.

❖ At the termination of the procedure, the current environment disappears and the previous environment must be restored.

❖ Space for local variables can be reserved by subtracting the required number of bytes from ESP.

❖ Offsets from ESP are used to address local variables.

Local Variables

Pseudo-code (Java-like)	Assembly Language

void Test(int i){
int k;
 	
k = i+9;
……
}
	Test PROC
	push EBP
	mov EBP, ESP
	sub ESP, 4
	push EAX
	mov DWORD PTR [EBP-4], 9
	mov EAX, [EBP + 8]
	add [EBP-4], EAX
	……
	pop EAX
	mov ESP, EBP
	pop EBP
	ret 4
	Test ENDP

Summary

- Procedure – Named block of executable code
  - CALL: call a procedure, push return address on top of stack
  - RET: pop the return address and return from procedure
  - Preserve registers across procedure calls
- Runtime stack – LIFO structure – Grows downwards
  - Holds return addresses, saved registers, etc.
  - PUSH – insert value on top of stack, decrement ESP
  - POP – remove top value of stack, increment ESP