CSC 3210
Computer Organization and Programming

CHAPTER 5: PROCEDURES (STACK)

Why Procedures? (Why stack?)

- Procedures, Functions, Methods, or **subroutine**
  - ARE the most fundamental language feature for:
    - Code reuse.
    - Ease debugging
    - Abstraction
- They allow us to **refer to some piece of code by a name**
- All we need to know is
  - how many arguments are needed,
  - what type of arguments,
  - what the subroutine returns,
  - what the subroutine computes
    - it's not necessary to know how the **subroutine** does what it does

```
Main (){
            pickMin( int x, int y, int z )

}
```

```
int pickMin( int x, int y, int z )
{
  int min = x ;
     if ( y < min ) min = y ;
     if ( z < min ) min = z ;
   return min ;
}
```

Why Procedures? (Why stack?)

- **What happens in a subroutine call?**
  1. When a **subroutine call** is executed,
     - The arguments **need to be evaluated** to values
  2. **Then**:
     - **control flow jumps** to the body of the **subroutine (How?)**,
     - and code **begins executing**
  3. Once a **return statement** has been encountered,
     - we're done with the **subroutine** ,
     - and **return** back to the **subroutine** call (**How?**)

pickMin(int x, int y, int z)

```
int pickMin( int x, int y, int z )
{
  int min = x ;
     if ( y < min ) min = y ;
     if ( z < min ) min = z ;
   return min ;
}
```

- In order to understand **subroutine** calls, you need to understand **the stack**

Outline

- **Stack Operations**
- Defining and Using Procedures
- Linking to an External Library
- The Irvine32 Library

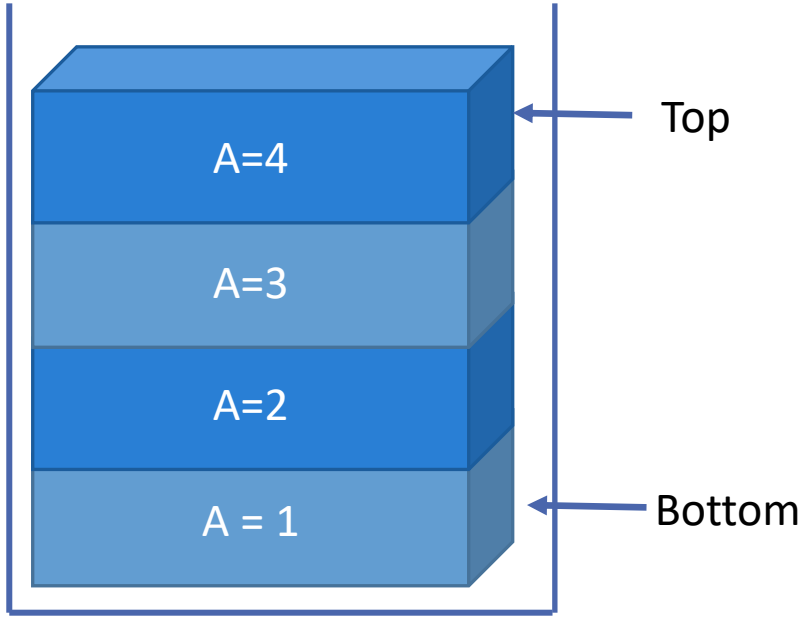
Stack Operations

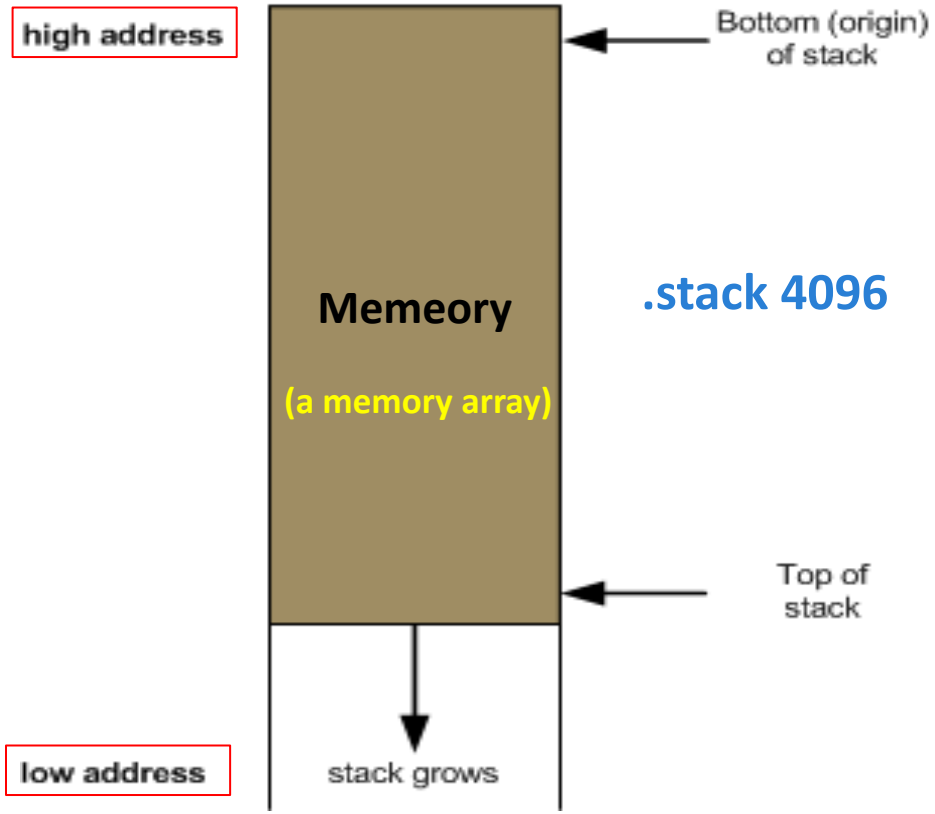
- **Runtime Stack**
- PUSH Operation
- POP Operation
- PUSH and POP Instructions
- Using PUSH and POP
  - Example: Reversing a String
- Related Instructions

Runtime Stack

- Imagine a stack of plates . . .
  - plates are only **added to the top**
  - plates are only **removed from the top**
  - **LIFO** structure

PUSH operation
Stores item in stack
POP operation
Retrieve item from stack



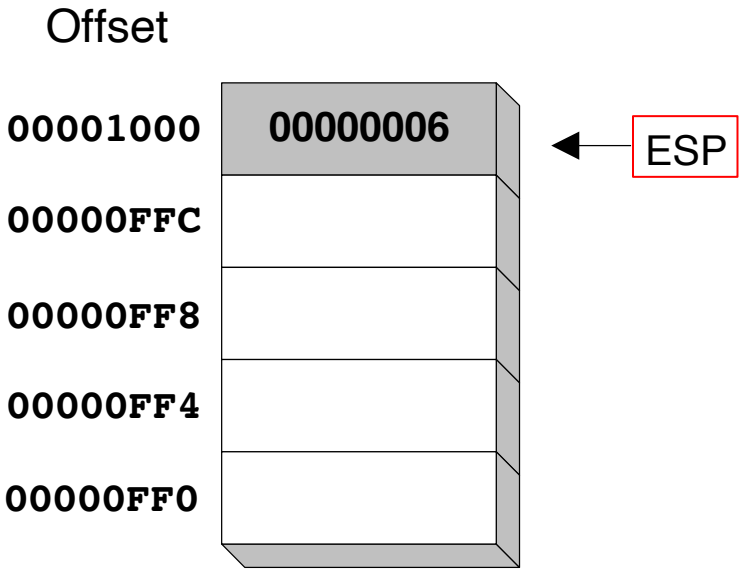


Runtime Stack

- The runtime stack is **a memory array**
- Managed by the CPU, **using two registers**
  - **SS** (stack **segment**)
  - **ESP** (stack pointer) *

- Stack **starts at some address**
- Then **grows *down* to a lower address**.

.stack 4096



* SP in Real-address mode

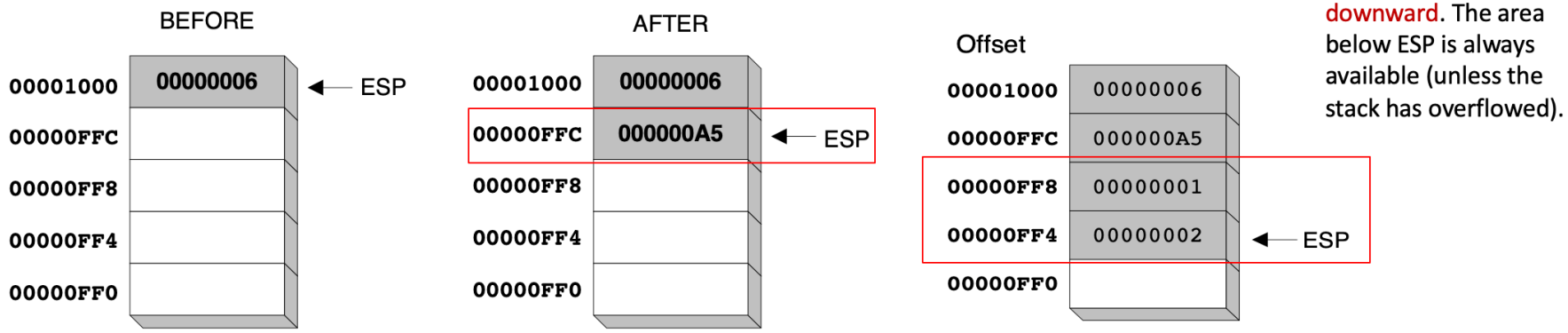
Stack Operations

- Runtime Stack
- **PUSH Operation**
- POP Operation
- Using PUSH and POP
  - Example: Reversing a String
- Related Instructions

PUSH Operation

- **A 32-bit push operation**
  - **decrements the stack pointer by 4**
  - and **copies** a value into the location **pointed to by the stack pointer**.

```
PUSH reg/mem16
PUSH reg/mem32
PUSH imm32
```



The stack grows downward. The area below ESP is always available (unless the stack has overflowed).

ESP = stack pointer (Top)

EBP = Base Pointe (Bottom)

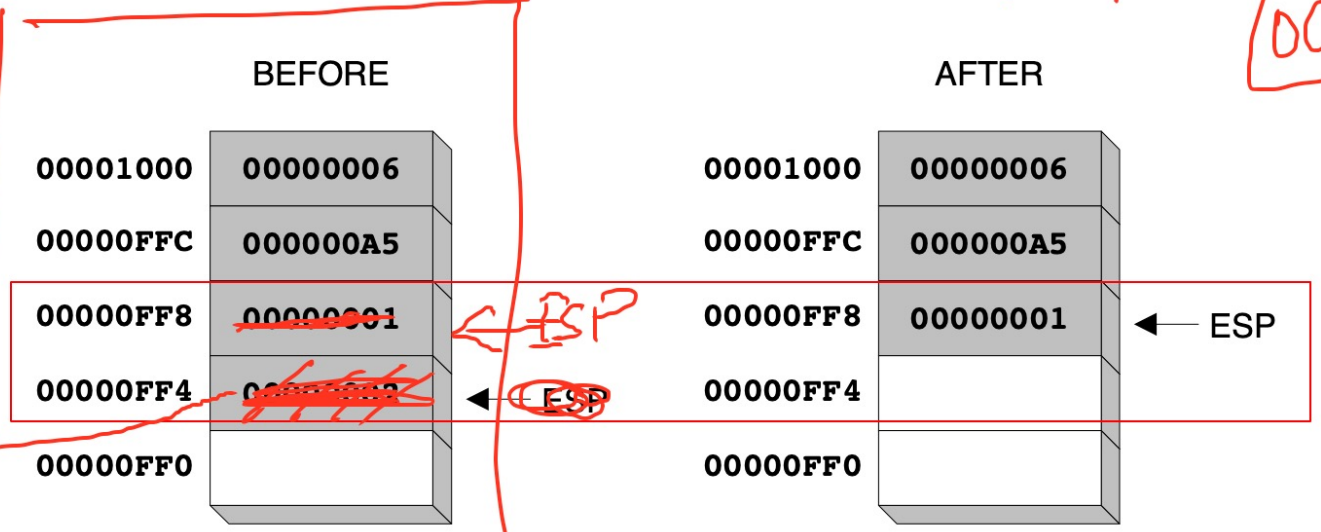
Stack Operations

- Runtime Stack
- PUSH Operation
- **POP Operation**
- Using PUSH and POP
  - Example: Reversing a String
- Related Instructions

POP Operation

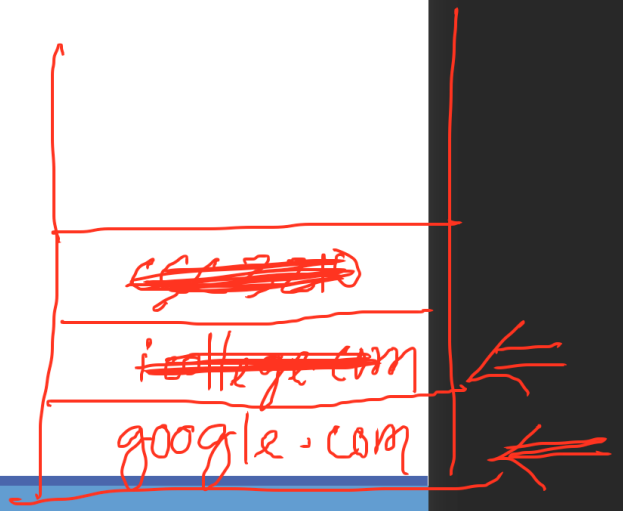
- **Copies** value at stack[ESP] **into** a register or variable.
- **Adds *n*** to ESP, where ***n*** is either **2 bytes or 4 bytes**
  - value of ***n*** depends on the attribute of the operand receiving the data

```
POP  reg/mem16
POP  reg/mem32
```



Stack Applications

- A **stack** makes
  - a convenient **temporary save area for registers** when they (ex. ECX) are used for more than one purpose
  - After they are modified, **they can be restored** to their original values
- When the CALL instruction executes,
  - the **CPU saves the current subroutine's return address** on the stack
- When calling a subroutine,
  - you **pass input values called arguments** by **pushing** them on the stack
- The stack provides temporary storage for local variables inside subroutines



Click to add notes

Stack Operations

- Runtime Stack
- PUSH Operation
- POP Operation
- **Using PUSH and POP**
  - Example: Reversing a String
- Related Instructions

Using PUSH and POP

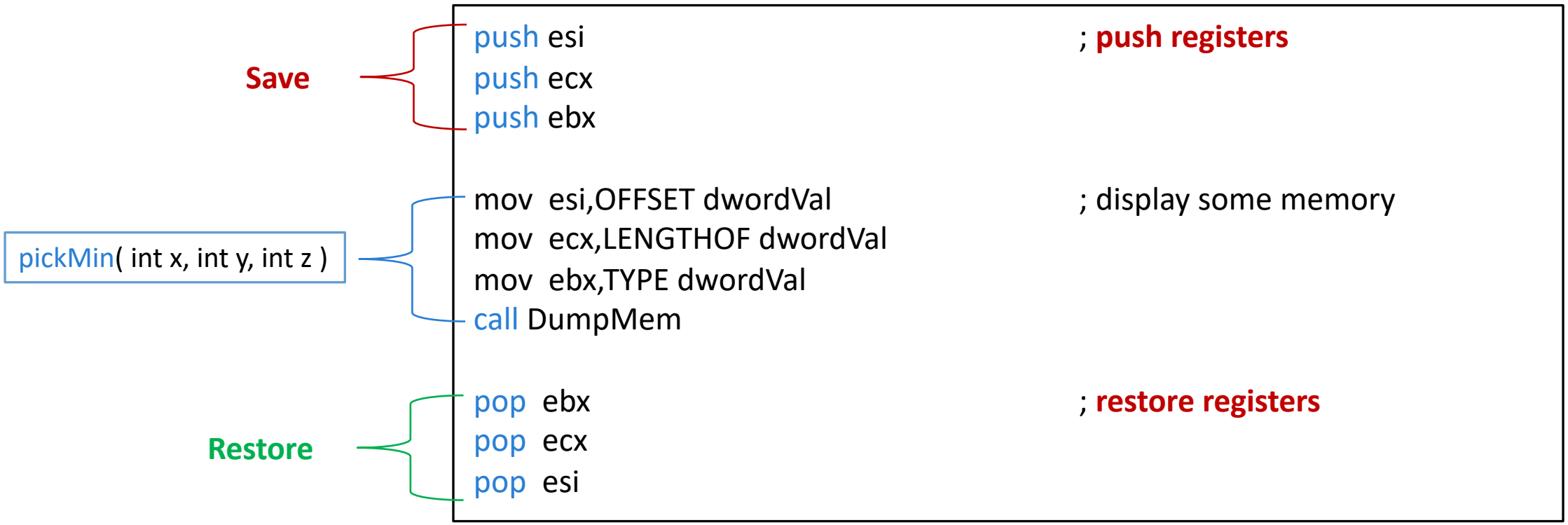
- **Save** and **restore** **registers** **when** they **contain important values**.
- PUSH and POP instructions **occur in the opposite order**

```
.data
array DWORD 1,2,3,4,5,6,7,8,9,0Ah,0Bh
.code
main PROC
    mov   esi,OFFSET array          ; starting OFFSET
    mov   ecx,LENGTHOF array        ; number of units
    mov   ebx,TYPE array            ; doubleword format
    call  DumpMem
```

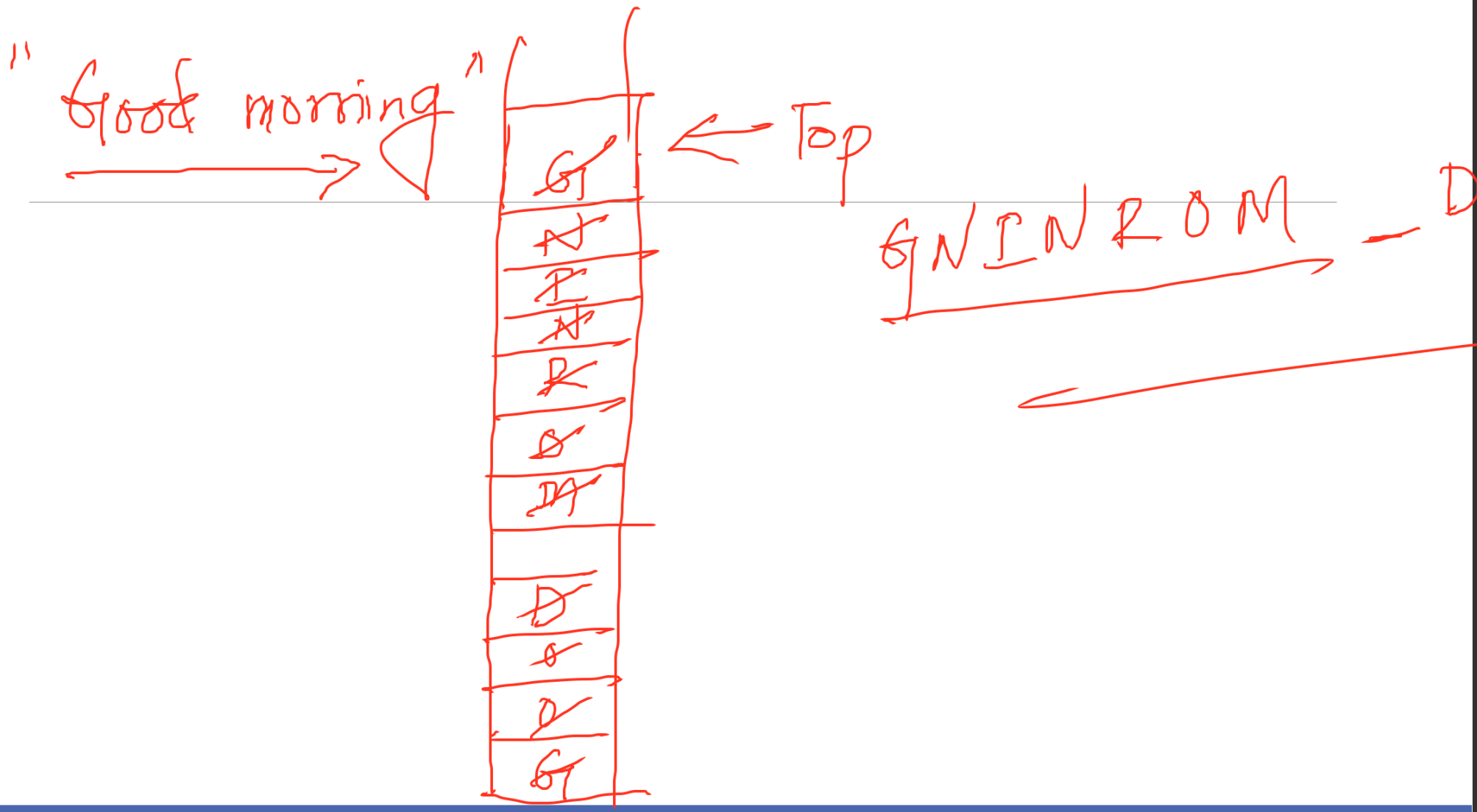
pickMin(int x, int y, int z)

DumpMem Writes a block of memory to the console window in hexadecimal

Using PUSH and POP



- **DumpMem** Writes a block of memory to the console window in hexadecimal


"Good morning"
Top
GNINROM_DOOG

Example1: Nested Loop

- When creating a nested loop, push the outer loop counter **before** entering the inner 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
        ;
        ;
        loop L2                 ; repeat the inner loop

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

Example2: Reversing a String

```
; Reversing a String                (RevStr.asm)

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO,dwExitCode:DWORD

.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) - 1

.code
main PROC
; Push the name on the stack.
     mov   ecx,nameSize
     mov   esi,0

L1:  movzx eax,aName[esi]              ; get character
     push  eax                         ; push on stack
     inc   esi
     loop  L1

; Pop the name from the stack, in reverse,
; and store in the aName array.
     mov   ecx,nameSize
     mov   esi,0

L2:  pop   eax                         ; get character
     mov   aName[esi],al               ; store in string
     inc   esi
     loop  L2
```

Why **must** **each character be put in EAX** before it is pushed?

Because only **doubleword** (32-bit) or **word** (16-bit) values can be pushed on the stack.

Attendance!

Stack Operations

- Runtime Stack
- PUSH Operation
- POP Operation
- Using PUSH and POP
  - Example: Reversing a String
- **Related Instructions**

Related Instructions

- **PUSHFD** and **POPFD**
  - push and pop the **EFLAGS register**
- **PUSHAD** pushes the **32-bit general-purpose registers on the stack**
  - order: **EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI**
- **POPAD** pops **the same registers off the stack in reverse order**
  - **PUSHA** and **POPA** do the same for **16-bit** registers

Outline

- Stack Operations
- **Defining and Using Procedures**
- Linking to an External Library
- The Irvine32 Library

Defining and Using Procedures

- **Creating Procedures**
- **Documenting Procedures**
- **Example: SumOf Procedure**
- CALL and RET Instructions
- Nested Procedure Calls
- Local and Global Labels
- Procedure Parameters

Creating Procedures

- **A procedure** is the ASM equivalent of a Java or C++ **function/method**
- A procedure is **declared** using the **PROC** and **ENDP** directives
- It must be **assigned a name** (a valid identifier)
- **When** you **create a procedure** other than your **program's startup procedure (main)**:
  - End it with a **RET** instruction
  - **RET** forces the CPU **to return to the location** from where the procedure was called **(address of call next instruction)**

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

```
main PROC
.
.
main ENDP
```

Documenting Procedures

- **Suggested documentation for each procedure:**
  - **A description** of all tasks accomplished by the procedure
  - **Receives**: A list of input parameters; state their usage and requirements
  - **Returns:** A description of values returned by the procedure
  - **Requires:** Optional list of requirements called preconditions that 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: SumOf Procedure

```
;-----------------------------------------------------
; Calculates and returns the sum of three 32-bit integers.
; Receives: EAX, EBX, ECX, the three integers. May be
; signed or unsigned.
; Returns: EAX = sum, and the status flags (Carry,
; Overflow, etc.) are changed.
; Requires: nothing
;-----------------------------------------------------

SumOf PROC
    add eax,ebx
    add eax,ecx
    ret
SumOf ENDP
```

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

Defining and Using Procedures

- Creating Procedures
- Documenting Procedures
- Example: SumOf Procedure
- **CALL and RET Instructions**
- Nested Procedure Calls
- Local and Global Labels
- Procedure Parameters

CALL and RET Instructions

- The **CALL** instruction calls a procedure:
  1) **pushes** offset of next instruction on the **stack**
  2) **copies** the address of the called procedure into **EIP**
- The **RET** instruction **returns** from a procedure:
  - **pops** top of stack into EIP 😕

```
main PROC
    call MySub
    mov eax,ebx
    .
    .
main ENDP

MySub PROC
    mov eax,edx
    .
    .
    ret
MySub ENDP
```

CALL-RET Example

0000025 is the **offset** of the instruction immediately following the **CALL** instruction

00000040 is the offset of the first instruction inside MySub

```
main PROC
    00000020 call MySub
    00000025 mov eax,ebx
    .
    .
main ENDP

MySub PROC
    00000040 mov eax,edx
    .
    .
    ret
MySub ENDP
```

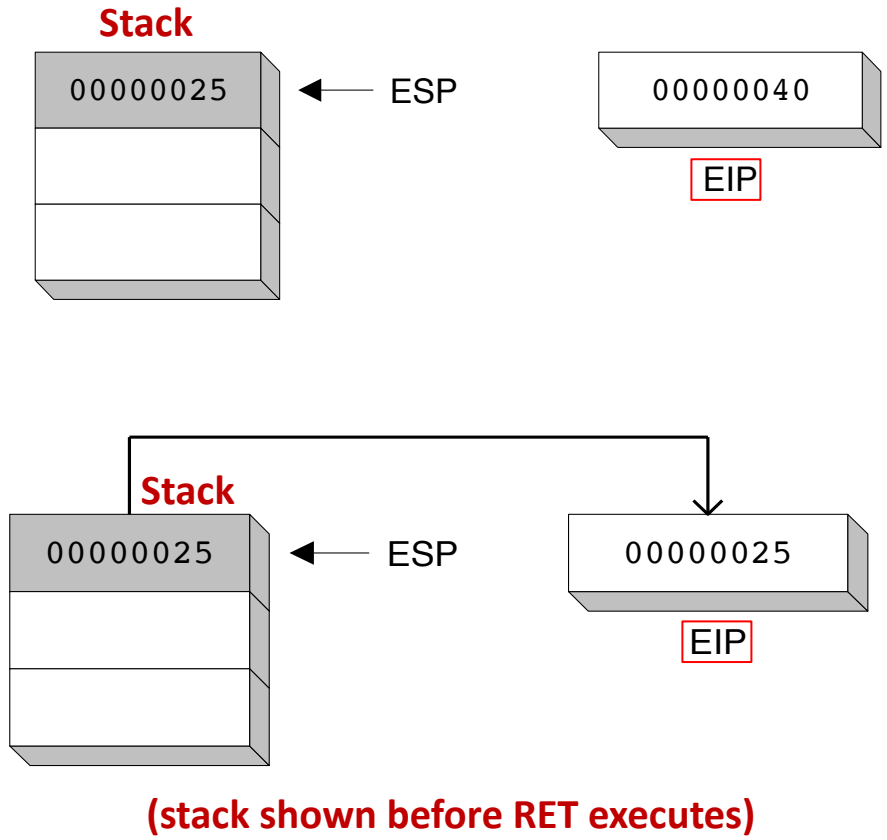
CALL-RET Example

The **CALL** instruction **pushes 00000025** onto the **stack**, and **loads 00000040** into **EIP**

The **RET** instruction **pops 00000025** from the **stack** into **EIP**

```
main PROC
     00000020 call MySub
     00000025 mov eax,ebx
     .
     .
main ENDP

MySub PROC
     00000040 mov eax,edx
     .
     .
     ret
MySub ENDP
```



Defining and Using Procedures

- Creating Procedures
- Documenting Procedures
- Example: SumOf Procedure
- CALL and RET Instructions
- **Nested Procedure Calls**
- **Local and Global Labels**
- Procedure Parameters

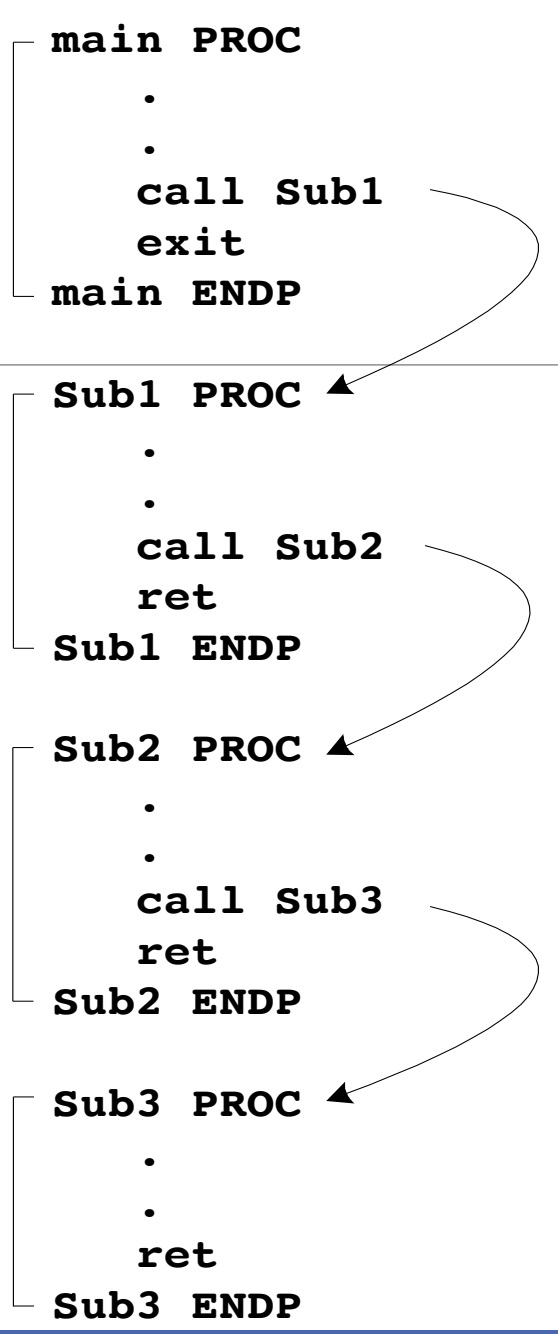
Nested Procedure Calls

1. By the time **Sub3** is called, **the stack** contains all **three** return addresses (**main, sub1, sub2**):

2. After the **return**,
   - **ESP** points to the next-highest stack entry.

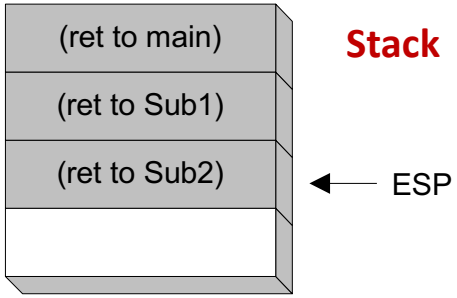
3. Finally, when **Sub1** **returns**, **stack[ESP]** is **popped** into the instruction pointer, and execution resumes in **main**:

Stack

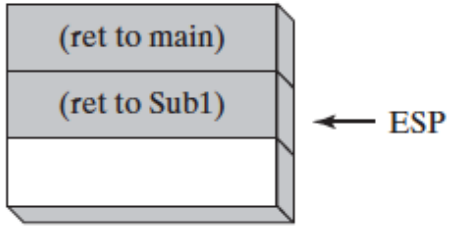


Nested Procedure Calls

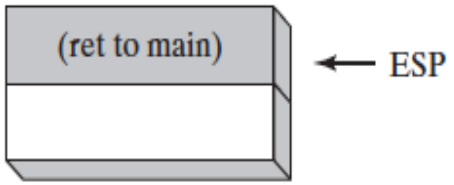
1. By the time **Sub3** is called, **the stack** contains all **three** return addresses (**main, sub1, sub2**):



2. After the **return**,
   - **ESP** points to the next-highest stack entry.



3. Finally, when **Sub1 returns**, **stack[ESP]** is **popped** into the instruction pointer, and execution resumes in **main**:




```
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
```

Nested Procedure Calls

- **The stack** proves itself a useful device for
  - **remembering information**, including nested procedure calls
- **Stack structures** are used in situations where programs **must retrace their steps in a specific order**

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
```

Defining and Using Procedures

- Creating Procedures
- Documenting Procedures
- Example: SumOf Procedure
- CALL and RET Instructions
- Nested Procedure Calls
- Local and Global Labels
- **Procedure Parameters**

Procedure Parameters

- **Why parameters?**
  - A good **procedure might be usable** in many different programs
    - **but not if it refers to specific variable names**

    Example: **calculating the sum of an integer array**:
    - it's **not a good idea to include references to specific variable names** inside the procedure
    - If you did, **the procedure could only be used with one array**

- **Parameters help to make procedures flexible** because **parameter values can change at runtime**

```
ArraySum PROC
        myarray  DWORD 10000h,20000h,30000h,40000h,50000h
        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
```

What if you wanted to calculate the sum of two or three arrays within the same program?

Procedure Parameters

Does not use parameters

```
ArraySum PROC
        myarray  DWORD 10000h,20000h,30000h,40000h,50000h
        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
```

Not a good idea to include references to specific variable names inside the procedure

Uses parameters

```
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
```

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

Procedure Parameters: Passing Register Arguments to Procedures

Main

```
; Testing the ArraySum procedure (TestArraySum.asm)

.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD

.data
array DWORD 10000h,20000h,30000h,40000h,50000h
theSum DWORD ?

.code
main PROC
    mov   esi,OFFSET array     ; ESI points to array
    mov   ecx,LENGTHOF array   ; ECX = array count
    call  ArraySum             ; calculate the sum
    mov   theSum,eax           ; returned in EAX

    INVOKE ExitProcess,0
main ENDP
```

Subroutines **must preserve all registers**, except for **eax**, **ecx**, and **edx**, which **can be changed across** Subroutines **call**, and **esp**, which must be updated according to the calling convention.

Subroutine

```
;-----------------------------------------------
; ArraySum
;
; Calculates the sum of an array of 32-bit integers.
; Receives: ESI = the array offset
;           ECX = number of elements in the array
; Returns:  EAX = sum of the array elements
;-----------------------------------------------
ArraySum PROC
    push  esi                  ; save ESI, ECX
    push  ecx
    mov   eax,0                ; set the sum to zero

L1: add   eax,[esi]            ; add each integer to sum
    add   esi,TYPE DWORD       ; point to next integer

    loop  L1                   ; repeat for array size

    pop   ecx                  ; restore ECX, ESI
    pop   esi
    ret                        ; sum is in EAX
ArraySum ENDP
```

Procedure Parameters: Passing Register Arguments to Procedures

After the **CALL** statement, we have the option of **copying the sum in EAX** to a variable.

```
.data
theSum  DWORD  ?
.code
main PROC
    mov   eax,10000h                ; argument
    mov   ebx,20000h                ; argument
    mov   ecx,30000h                ; argument
    call  Sumof                     ; EAX = (EAX + EBX + ECX)
    mov   theSum,eax                ; save the sum
```

```
;-----------------------------------------------------
; sumof
;
; Calculates and returns the sum of three 32-bit integers.
; Receives: EAX, EBX, ECX, the three integers. May be
;           signed or unsigned.
; Returns:  EAX = sum
-----------------------------------------------------
SumOf PROC
    add   eax,ebx
    add   eax,ecx
    ret
SumOf ENDP
```

When not to push a register

- The **sum** of the three registers is stored in **EAX** on line **(3)**,
- **but** the **POP** instruction **replaces it** with the starting value of **EAX** on line **(4)**:

```
SumOf PROC                    ; sum of three integers
      push eax                ; 1
      add eax,ebx             ; 2
      add eax,ecx             ; 3
      pop eax                 ; 4
      ret
SumOf ENDP
```

causing the procedure's return value to be lost

Outline

- Stack Operations
- Defining and Using Procedures
- **Linking to an External Library**
- The Irvine32 Library

Linking to an External Library

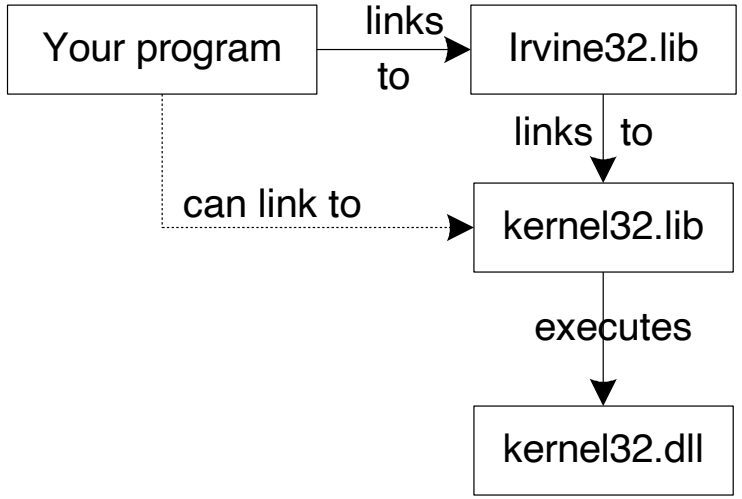
- What is a Link Library?
- How the Linker Works

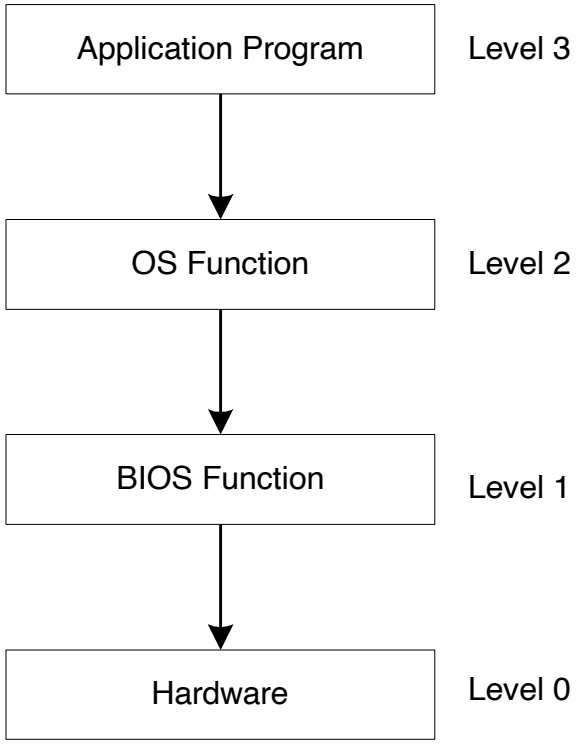
What is a Link Library?

- A **file containing procedures** that have been compiled into machine code
  - constructed from one or more **OBJ files**
- To **build** a **library**, . . .
  - start with **one or more ASM source files**
  - **assemble** each into an **OBJ** file
  - **create** an empty library file (extension .LIB)
  - add the **OBJ file(s)** to the library file, using the Microsoft LIB utility

How The Linker Works

- Your **programs link to Irvine32.lib using the linker command** inside a batch file named make32.bat.
- Notice the two **LIB files**: Irvine32.lib, and kernel32.lib
  - the latter is part of the Microsoft *Win32 Software Development Kit (SDK)*





Chapter2

Outline

- Stack Operations
- Defining and Using Procedures
- Linking to an External Library
- **The Irvine32 Library**

Calling Irvine32 Library Procedures

- **Call** each procedure using the **CALL instruction**
- **Some procedures** require **input arguments**
- The **INCLUDE directive copies** in the procedure prototypes (**declarations**)
- The following example **displays "1234" on the console**:

```
INCLUDE Irvine32.inc
.code
    mov   eax,1234h              ; input argument
    call  WriteHex               ; show hex number
    call  Crlf                   ; end of line
```

WriteHex PROTO

Next, a Call instruction executes WriteHex:

Call WriteHex

Some Library Procedures

- **Clrscr** - Clears console, locates **cursor** at upper left corner
- **Crlf** - Writes end of line sequence to standard output
- **DumpMem** - Writes block of memory to standard output in hex
- **DumpRegs** – Displays general-purpose registers and flags (hex)
- **ReadChar** - Reads a single character from standard input
- **ReadHex** - Reads 32-bit hexadecimal integer from keyboard
- **ReadDec** - Reads 32-bit unsigned decimal integer from keyboard
- **ReadInt** - Reads 32-bit signed decimal integer from keyboard
- **ReadString** - Reads string from ***stdin***, terminated by [Enter]

Some Library Procedures

- **WriteChar** - **Writes** a single character to standard output
- **WriteDec** - **Writes** unsigned 32-bit integer in decimal format
- **WriteHex** - **Writes** an unsigned 32-bit integer in hexadecimal format
- **WriteHexB** – **Writes** **byte**, **word**, or **doubleword** in hexadecimal format
- **WriteInt** - **Writes** signed 32-bit integer in decimal format
- **WriteString** - **Writes** null-terminated string to console window

Some Library Procedures: Example 1

- **Clear the screen**, **delay** the program for 500 milliseconds, and **dump the registers and flags**

```
.code
    call Clrscr
    mov  eax,500
    call Delay
    call DumpRegs
```

DumpRegs – Displays general-purpose registers and flags (hex)

Sample output:

```
EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0
```

Some Library Procedures: Example 2

- **Display a null-terminated string** and **move the cursor to the beginning** of the next screen line

```
.data
str1 BYTE "Assembly language is easy!",0

.code
    mov  edx,OFFSET str1
    call WriteString
    call Crlf
```

WriteString - Writes null-terminated string to console window

Some Library Procedures: Example 3

- **Display an unsigned integer in binary**, **decimal**, and **hexadecimal**, each on a separate line.

```
IntVal = 35
.code
    mov  eax,IntVal
    call WriteBin           ; display binary
    call Crlf
    call WriteDec           ; display decimal
    call Crlf
    call WriteHex           ; display hexadecimal
    call Crlf
```

Sample output:

```
0000 0000 0000 0000 0000 0000 0010 0011
35
23
```

Some Library Procedures: Example 4

- **Input a string** from the user
- **EDX** points to the string and **ECX** specifies the maximum number of characters the user is permitted to enter.

```
.data
fileName BYTE 80 DUP(0)

.code
    mov   edx,OFFSET fileName
    mov   ecx,SIZEOF fileName - 1
    call  ReadString
```

ReadString - Reads string from stdin, terminated by [Enter]

A null byte is automatically appended to the string.

Example 5

```
; Adding Three Numbers

Include Irvine32.inc
                ;WriteInt PROTO ; Irvine32 library
                ; Crlf  PROTO
.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword

 .code

main PROC

mov ebx,10 ; pass three parameters, in order
mov ecx,20
mov edx,30

call AddThree ; look for return value in EAX
call WriteInt ; display the number
call Crlf ; output a CR/LF

invoke ExitProcess,0

main ENDP
```

```
AddThree PROC

Mov eax,0

add eax,ebx

add eax,ecx

add eax,edx

ret

AddThree ENDP

end main
```