Advanced Procedures

COE 205

Computer Organization and Assembly Language

Computer Engineering Department

King Fahd University of Petroleum and Minerals

Presentation Outline

- Stack Parameters
- Local Variables and Stack Frames
- Simplifying the Writing of Procedures
- Recursion
- Creating Multi-Module Programs

Parameter Passing - Revisited

- 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

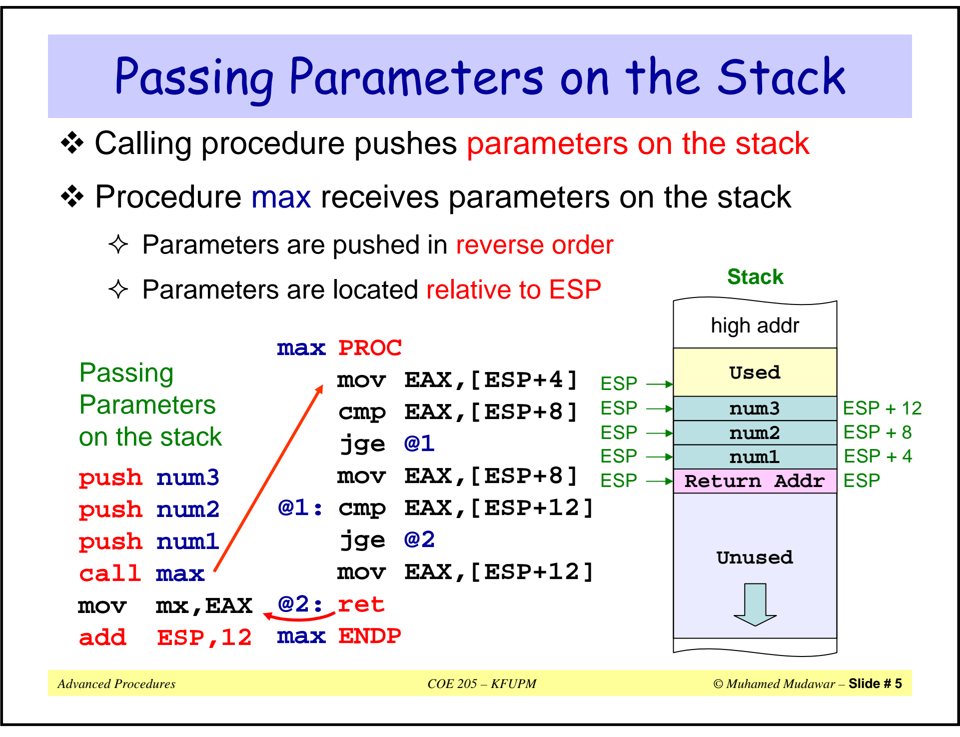
Stack Parameters

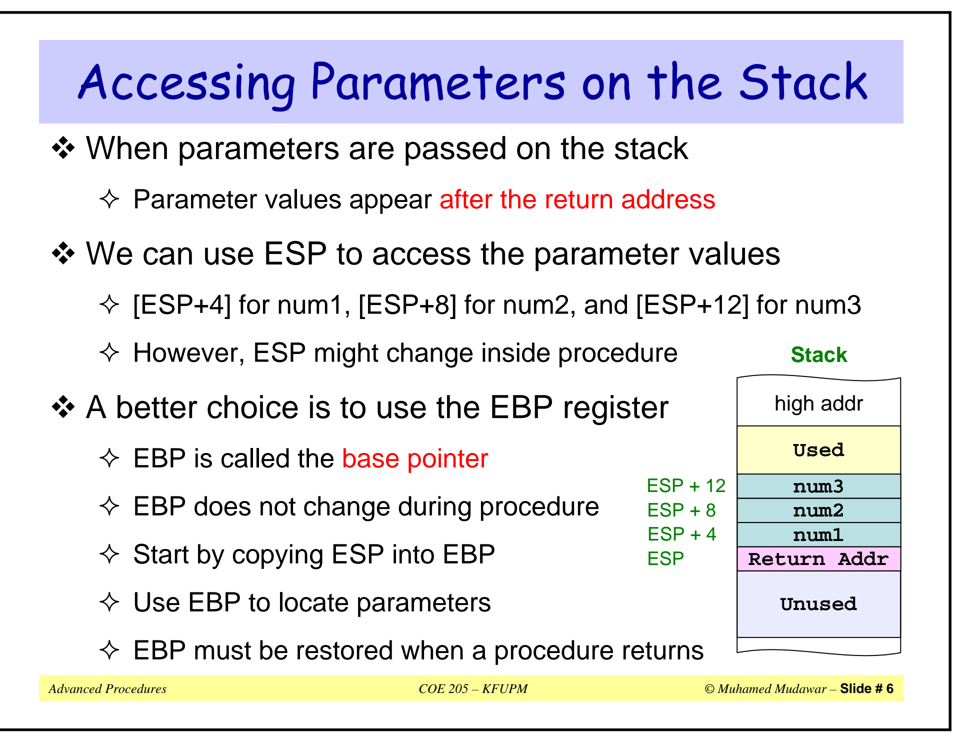
- Consider the following max procedure

```
int max ( int x, int y, int z ) {
    int temp = x;
    if (y > temp) temp = y;
    if (z > temp) temp = z;
    return temp;
}
```

Calling procedure: `mx = max(num1, num2, num3)`

Register Parameters	Stack Parameters
`mov eax, num1`	`push num3` } Reverse Order
`mov ebx, num2`	`push num2`
`mov ecx, num3`	`push num1`
`call max`	`call max`
`mov mx, eax`	`mov mx, eax`


Passing Parameters on the Stack
Calling procedure pushes parameters on the stack
Procedure max receives parameters on the stack
Parameters are pushed in reverse order
Parameters are located relative to ESP
Passing Parameters on the stack
push num3
push num2
push num1
call max
mov mx,EAX
add ESP,12
max PROC
mov EAX,[ESP+4]
cmp EAX,[ESP+8]
jge @1
mov EAX,[ESP+8]
@1: cmp EAX,[ESP+12]
jge @2
mov EAX,[ESP+12]
@2: ret
max ENDP
Stack
high addr
Used
num3
num2
num1
Return Addr
Unused
ESP
ESP + 12
ESP + 8
ESP + 4
Advanced Procedures
COE 205 – KFUPM
© Muhamed Mudawar – Slide # 5


Accessing Parameters on the Stack
When parameters are passed on the stack
Parameter values appear after the return address
We can use ESP to access the parameter values
[ESP+4] for num1, [ESP+8] for num2, and [ESP+12] for num3
However, ESP might change inside procedure
A better choice is to use the EBP register
EBP is called the base pointer
EBP does not change during procedure
Start by copying ESP into EBP
Use EBP to locate parameters
EBP must be restored when a procedure returns
Stack
high addr
Used
num3
num2
num1
Return Addr
Unused
ESP + 12
ESP + 8
ESP + 4
ESP
Advanced Procedures
COE 205 – KFUPM
© Muhamed Mudawar – Slide # 6

Using the Base Pointer Register

- ❖ EBP is used to locate parameters on the stack
- ❖ Like any other register, EBP must be saved before use

Passing Parameters on the stack

```
push num3
push num2
push num1
call max
mov  mx,EAX
add  ESP,12
```

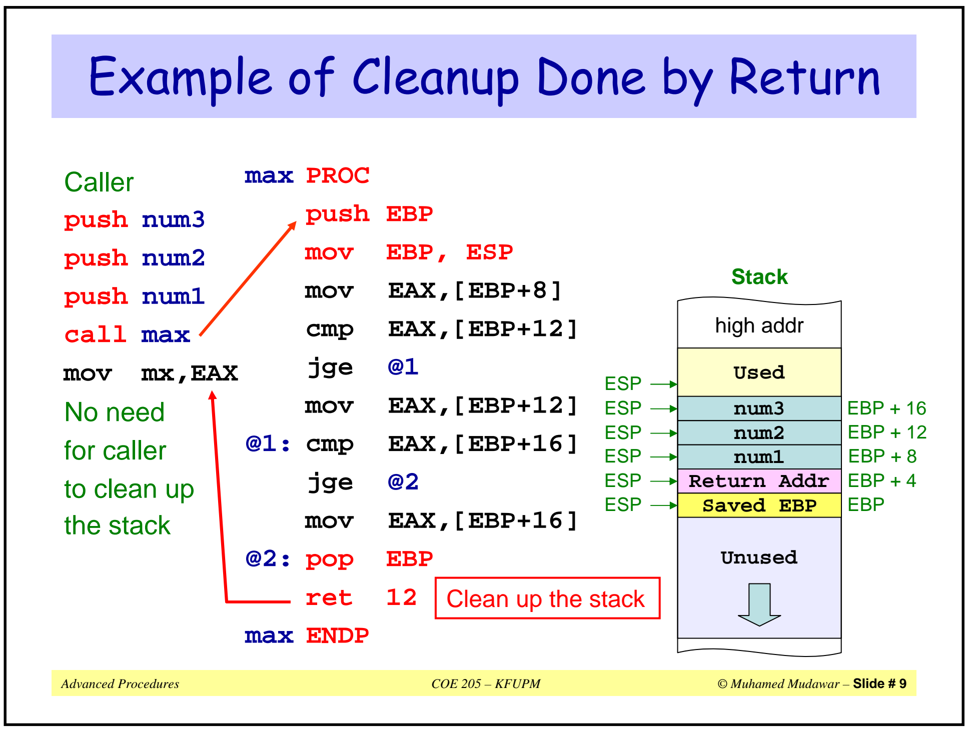
```
max PROC
     push EBP
     mov  EBP, ESP
     mov  EAX,[EBP+8]
     cmp  EAX,[EBP+12]
     jge  @1
     mov  EAX,[EBP+12]
@1:  cmp  EAX,[EBP+16]
     jge  @2
     mov  EAX,[EBP+16]
@2:  pop  EBP
     ret
max ENDP
```

Stack

ESP	Stack	EBP offset
	high addr	
ESP →	Used	
ESP →	num3	EBP + 16
ESP →	num2	EBP + 12
ESP →	num1	EBP + 8
ESP →	Return Addr	EBP + 4
ESP →	Saved EBP	EBP
	Unused	

Who Should Clean up the Stack?

- ❖ When returning for a procedure call …
  - ✧ Who should remove parameters and clean up the stack?
- ❖ Clean-up can be done by the calling procedure
  - ✧ `add ESP,12      ; will clean up stack`
- ❖ Clean-up can be done also by the called procedure
  - ✧ We can specify an optional integer in the `ret` instruction
  - ✧ `ret 12          ; will return and clean up stack`
- ❖ Return instruction is used to clean up stack
  - ✧ `ret n           ; n is an integer constant`
  - ✧ Actions taken
    - EIP = [ESP]
    - ESP = ESP + 4 + *n*



Register versus Stack Parameters

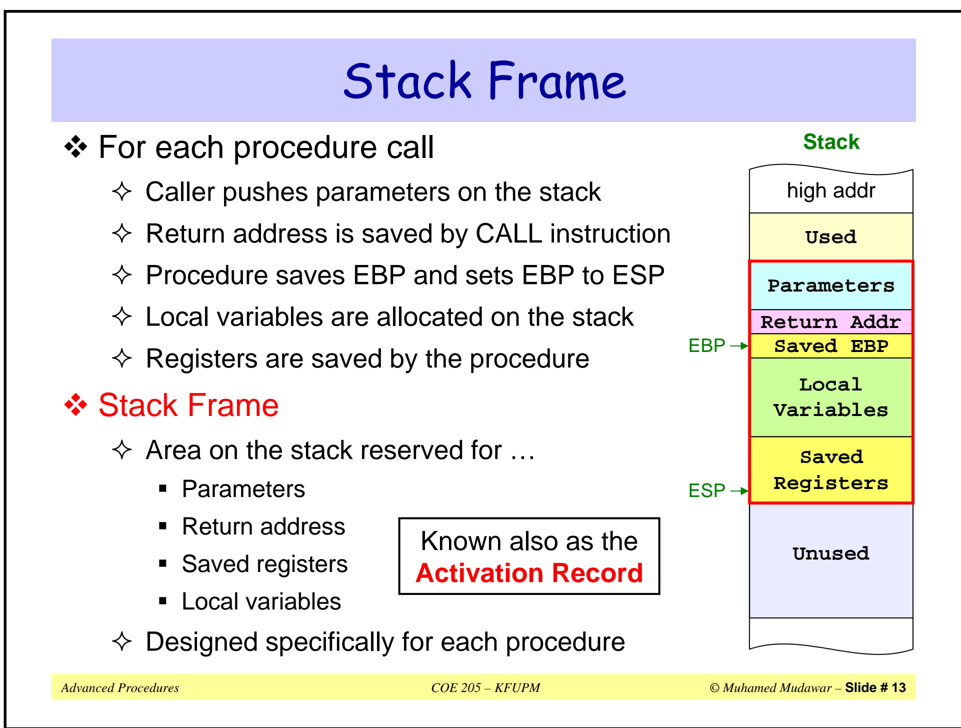
- Passing Parameters in Registers
  - Pros: Convenient, easier to use, and faster to access
  - Cons: Only few parameters can be passed
    - A small number of registers are available
    - Often these registers are used and need to be saved on the stack
    - Pushing register values on stack negates their advantage
- Passing Parameters on the Stack
  - Pros: Many parameters can be passed
    - Large data structures and arrays can be passed
  - Cons: Accessing parameters is not simple
    - More overhead and slower access to parameters

Next . . .

- ❖ Stack Parameters
- ❖ Local Variables and Stack Frames
- ❖ Simplifying the Writing of Procedures
- ❖ Recursion
- ❖ Creating Multi-Module Programs

Local Procedure Variables

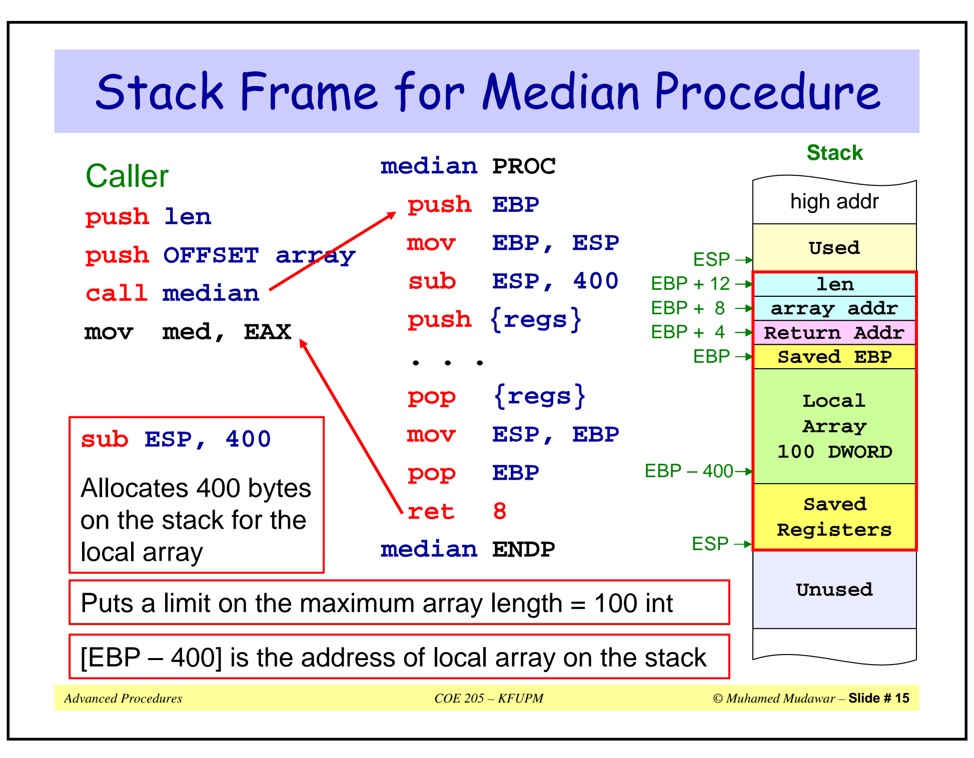
- ❖ Local procedure variables are dynamic in nature
  - ✧ Come into existence when the procedure is invoked
  - ✧ Disappear when the procedure terminates
- ❖ Cannot reserve space for local variables in data segment
  - ✧ Because such space allocation is static
    - ▪ Remains active even after returning from the procedure call
  - ✧ Also because it does not work with recursive procedures
- ❖ Local variables can be stored in registers or on the stack
  - ✧ Registers are best used for local variables when ...
    - ▪ Variables are small in size and frequently used (e.g. loop counter)
  - ✧ Local variables are stored on the stack when …
    - ▪ They are large in size (e.g. arrays) or cannot fit in registers
    - ▪ Recursive calls are made by the procedure

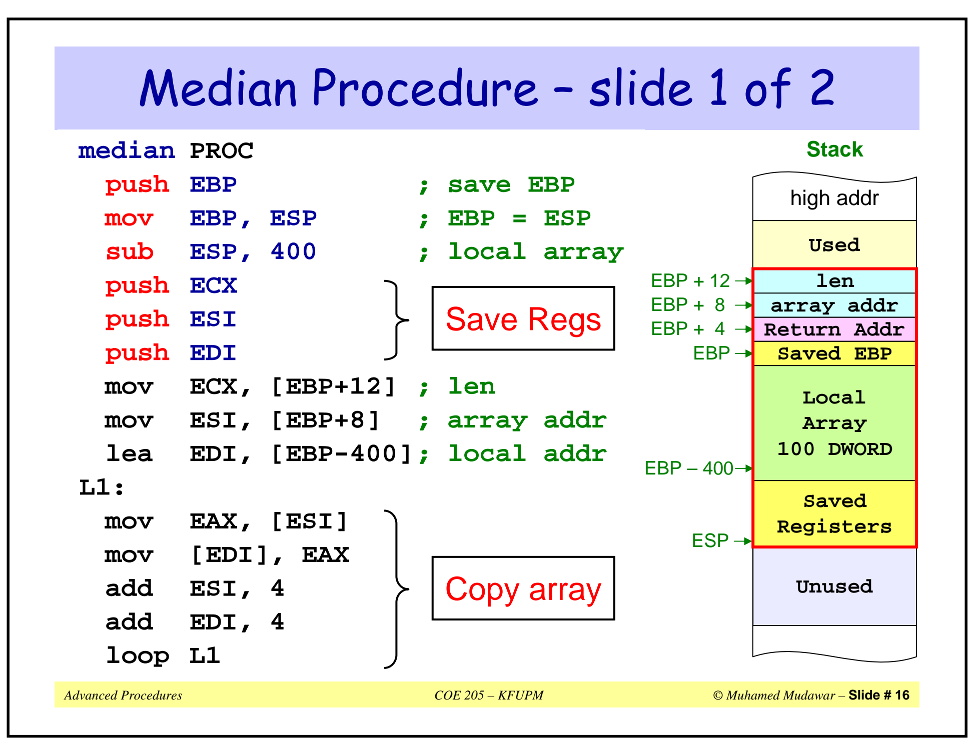


Example on Local Variables

- Consider the following procedure: **`median`**
- To compute the median of an array of integers
  - First, copy the array (to avoid modifying it) into a local array
  - Second, sort the local array
  - Third, find the integer value at the middle of the sorted array

```
int median (int array[], int len) {
    int local[100];          // local array (100 int)
    for (i=0; i<len; i++)
      local[i] = array[i]; // Copy the array
    bubbleSort(local,len); // Sort the local array
    return local[len/2];    // Return middle element
}
```


Stack Frame for Median Procedure
Caller
push len
push OFFSET array
call median
mov med, EAX
median PROC
push EBP
mov EBP, ESP
sub ESP, 400
push {regs}
. . .
pop {regs}
mov ESP, EBP
pop EBP
ret 8
median ENDP
sub ESP, 400
Allocates 400 bytes on the stack for the local array
Puts a limit on the maximum array length = 100 int
[EBP – 400] is the address of local array on the stack
Stack
high addr
Used
ESP
EBP + 12
EBP + 8
EBP + 4
EBP
len
array addr
Return Addr
Saved EBP
Local Array 100 DWORD
EBP – 400
Saved Registers
ESP
Unused
Advanced Procedures
COE 205 – KFUPM
© Muhamed Mudawar – Slide # 15


Median Procedure - slide 1 of 2
median PROC
push EBP ; save EBP
mov EBP, ESP ; EBP = ESP
sub ESP, 400 ; local array
push ECX
push ESI
push EDI
Save Regs
mov ECX, [EBP+12] ; len
mov ESI, [EBP+8] ; array addr
lea EDI, [EBP-400]; local addr
L1:
mov EAX, [ESI]
mov [EDI], EAX
add ESI, 4
add EDI, 4
loop L1
Copy array
Stack
high addr
Used
EBP + 12
EBP + 8
EBP + 4
EBP
len
array addr
Return Addr
Saved EBP
Local Array 100 DWORD
EBP – 400
Saved Registers
ESP
Unused
Advanced Procedures
COE 205 – KFUPM
© Muhamed Mudawar – Slide # 16

Median Procedure - slide 2 of 2

```
; Call sort procedure to sort local array
; Parameters are passed on the stack
  push DWORD PTR [EBP+12]
  lea  EDI, [EBP-400]
  push EDI   ; local array address
  call sort  ; sort local array
  mov  ESI, [EBP+12]     ; len
  shr  ESI, 1            ; ESI = len/2
  mov  EAX, [EDI+ESI*4] ; local[len/2]
  pop  EDI  }
  pop  ESI  }  Restore Regs
  pop  ECX  }
  mov  ESP, EBP          ; free local
  pop  EBP               ; restore EBP
  ret  8                 ; return & cleanup
median ENDP
```


Next . . .

- ❖ Stack Parameters
- ❖ Local Variables and Stack Frames
- ❖ Simplifying the Writing of Procedures
- ❖ Recursion
- ❖ Creating Multi-Module Programs

Simplifying the Writing of Procedures

- PROC directive
  - Specifies registers to be saved and restored
  - Specifies parameters
- LOCAL directive
  - Declares local variables
- PROTO directive
  - Specifies procedure prototypes
- INVOKE directive
  - Simplifies procedure calls

MASM provides useful directives to simplify the writing of procedures

PROC Directive

- Declares a procedure with an optional list of parameters
- Syntax:

```
procName PROC [USES reglist], paramList
```

- **paramList** is list of parameters separated by commas

```
param1:type1, param2:type2, . . .
```

- Each parameter has the following syntax

```
paramName:type
```

- **type** must either be one of the standard ASM types

  BYTE, SBYTE, WORD, SWORD, DWORD, … etc.

  Or it can be a pointer to one of these types

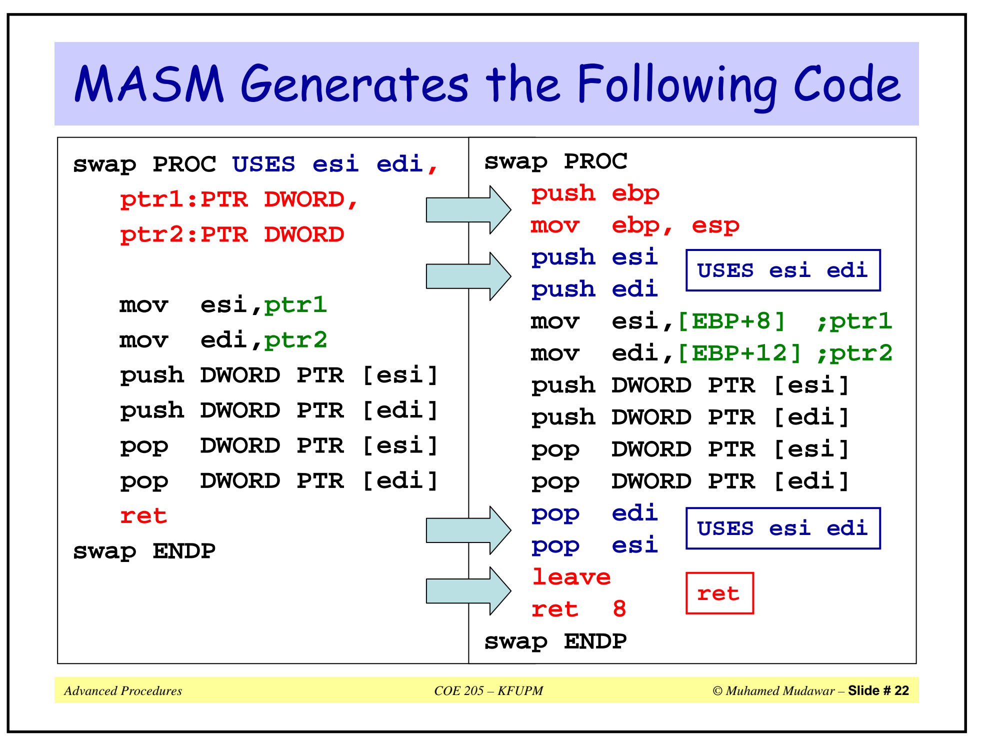
PROC Example

- Swap Procedure: exchanges two 32-bit integer variables
- Two stack parameters: **ptr1** and **ptr2**

```
swap PROC USES esi edi ,                    
    ptr1:PTR DWORD,          ; pointer to 1st integer
    ptr2:PTR DWORD           ; pointer to 2nd integer
    mov   esi,ptr1           ; get pointers
    mov   edi,ptr2
    push DWORD PTR [esi]     ; push first integer
    push DWORD PTR [edi]     ; push second integer
    pop  DWORD PTR [esi]     ; replace first integer
    pop  DWORD PTR [edi]     ; replace second integer
    ret
swap ENDP
```

comma is required

parameter list



ENTER and LEAVE Instructions

- ENTER instruction creates stack frame for a procedure
  - Pushes EBP on the stack `push ebp`
  - Sets EBP to the base of the stack frame `mov ebp,esp`
  - Reserves space for local variables `sub esp,nbytes`
- Example:

```
myproc PROC
  enter 8, 0
```

(8 = nbytes, 0 = level)

is equivalent to

```
myproc PROC
  push ebp
  mov  ebp, esp
  sub  esp, 8
```

- LEAVE instruction is equivalent to

```
mov esp, ebp
pop ebp
```


Language Specifier

- .MODEL directive
  - Specifies program's memory model and language specifier

    `.MODEL MemoryModel [, ModelOptions]`
- Example: `.MODEL flat, stdcall`
- Language specifier specifies
  - Procedure naming scheme
  - Parameter passing conventions
  - Options: `STDCALL, C, FORTRAN, SYSCALL, etc.`
- We are using the STDCALL language specifier
  - Procedure arguments are pushed on stack in reverse order
  - Called procedure cleans up the stack

LOCAL Directive

- The LOCAL directive declares a list of local variables
  - Immediately follows the PROC directive
  - Each variable is assigned a type
  - Syntax: LOCAL varlist
- Syntax:

```
LOCAL var1:type1, var2:type2, . . .
```

- Example:

```
myproc PROC
  LOCAL  var1:DWORD,        ; var1 is a DWORD
         var2:WORD,         ; var2 is a WORD
         var3[20]:BYTE      ; array of 20 bytes
```


LOCAL Example

- Given **myproc** procedure

```
myproc PROC
  LOCAL  var1:DWORD,
         var2:WORD,
         var3[20]:BYTE
  mov eax, var1
  mov bx,  var2
  mov dl,  var3
  . . .
  ret
myproc ENDP
```

- MASM generates:

```
myproc PROC
  push ebp
  mov  ebp, esp
  add  esp, -28
  mov eax, [EBP-4]
  mov bx,  [EBP-6]
  mov dl,  [EBP-26]
  . . .
  leave
  ret
myproc ENDP
```

More on Local Variables

- Local variables can be of different sizes
- Created on the stack by the LOCAL directive:
  - 8-bit: assigned to next available byte
  - 16-bit: assigned to next even (word) boundary
  - 32-bit: assigned to next doubleword boundary

```
Example PROC
    LOCAL v1:BYTE,
          v2:DWORD,
          v3:WORD
    . . .
Example ENDP
```


PROTO Directive

- Creates a procedure prototype
- Syntax:

```
procName PROTO paramList
```

- Uses the same parameter list that appears in procedure
- Prototypes are required for …
  - Procedures called by INVOKE
  - Calling external procedures
- Standard configuration:
  - PROTO appears at top of the program listing
  - Procedure implementation occurs later in the program

PROTO Examples

- Prototype for the **ArraySum** procedure:

```
ArraySum PROTO,
   arrayPtr: PTR DWORD,   ; array pointer
   arrayLen: DWORD        ; array length
```

- Prototype for the **swap** procedure:

```
swap PROTO,
   ptr1:PTR DWORD,        ; 1st int pointer
   ptr2:PTR DWORD         ; 2nd int pointer
```


INVOKE Directive

- INVOKE is a powerful replacement for CALL instruction
- Lets you pass multiple arguments
- Syntax: **`INVOKE procName paramList`**
- **`ParamList`** is an optional list of procedure arguments
- MASM requires that every procedure called by the INVOKE directive to have a prototype
- Arguments can be:
  - Immediate values and integer expressions
  - Variable names
  - Addresses
  - Register names

INVOKE Examples

- Consider the following procedure prototypes:

```
ArraySum PROTO, arrayPtr:PTR DWORD, arrayLen:DWORD
swap PROTO, ptr1:PTR DWORD, ptr2:PTR DWORD
```

- We can use INVOKE to call these procedures:

```
INVOKE ArraySum,ADDR array,ecx
```

MASM Generates

```
push ecx
push OFFSET array
call ArraySum
```

ADDR is the Address operator

```
INVOKE swap, ADDR var1, esi
```

MASM Generates

```
push esi
push OFFSET var1
call swap
```


Rewriting the Median Procedure - 1

```
median PROC USES ECX ESI EDI,
  arrayptr:PTR SDWORD,
  len: DWORD

  local sorted[100]:SDWORD

  mov   ECX, len
  mov   ESI, arrayptr
  lea   EDI, sorted
L1:
  mov   EAX, [ESI]
  mov   [EDI], EAX
  add   ESI, 4
  add   EDI, 4
  loop  L1
```

MASM Generates

```
  push  EBP
  mov   EBP, ESP
  sub   ESP, 400
  push  ECX
  push  ESI
  push  EDI
  mov   ECX, [EBP+12]
  mov   ESI, [EBP+8]
  lea   EDI, [EBP-400]
L1:
  mov   EAX, [ESI]
  mov   [EDI], EAX
  add   ESI, 4
  add   EDI, 4
  loop  L1
```

Copy Array

Rewriting the Median Procedure - 2

```
; Call sort procedure

  lea   EDI, sorted

  INVOKE sort, edi, len

  mov   ESI, len
  shr   ESI, 1
  mov   EAX, [EDI+ESI*4]

  ret

median ENDP
```

MASM Generates

```
  lea   EDI, [EBP-400]
  push  [EBP+12]
  push  EDI
  call  sort
  mov   ESI, [EBP+12]
  shr   ESI, 1
  mov   EAX, [EDI+ESI*4]
  pop   EDI
  pop   ESI
  pop   ECX
  leave
  ret   8
median ENDP
```


More on LEA Instruction

- LEA returns address of both direct and indirect operands
  - Used for obtaining address of a stack parameter or local variable
- OFFSET operator can only return constant offsets
  - Offsets of global variables declared in the data segment
- Example:

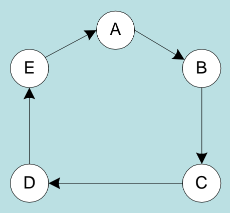
```
example PROC, count:DWORD
    LOCAL temp[20]:BYTE

    mov edi, OFFSET count       ; invalid operand
    mov esi, OFFSET temp        ; invalid operand
    lea edi, count              ; ok
    lea esi, temp               ; ok
```

Next . . .

- ❖ Stack Parameters
- ❖ Local Variables and Stack Frames
- ❖ Simplifying the Writing of Procedures
- ❖ Recursion
- ❖ Creating Multi-Module Programs

What is Recursion?

- ❖ The process created when . . .
  - ✧ A procedure calls itself
  - ✧ Procedure A calls procedure B, which in turn calls procedure A
- ❖ Using a graph in which each node is a procedure and each edge is a procedure call, recursion forms a cycle

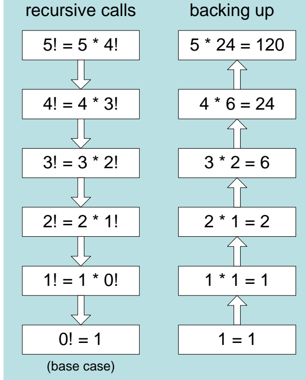


Calculating Factorial (1 of 3)

This function calculates the factorial of integer *n*

A new value of *n* is saved in each stack frame

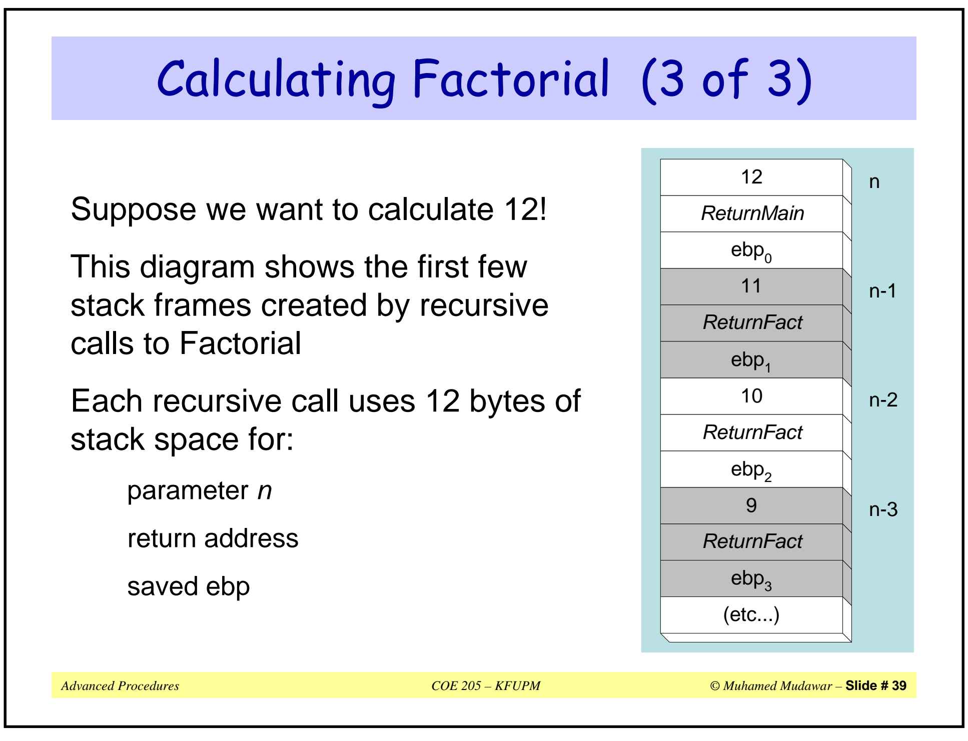
```
int factorial(int n)
{
    if(n == 0)
      return 1;
    else
      return n * factorial(n-1);
}
```



As each call instance returns, the product it returns is multiplied by the previous value of *n*

Calculating Factorial (2 of 3)

```
Factorial PROC
     push ebp
     mov  ebp,esp
     mov  eax,[ebp+8]         ; eax = n
     cmp  eax,0               ; n > 0?
     ja   L1                  ; yes: continue
     mov  eax,1               ; no: return 1
     jmp  L2
L1:  dec  eax
     push eax                 ; Factorial(n-1)
     call Factorial
ReturnFact:
     mov  ebx,[ebp+8]         ; get n
     mul  ebx                 ; eax = eax * ebx
L2:  pop  ebp                 ; return EAX
     ret  4                   ; clean up stack
Factorial ENDP
```



Next . . .

- ❖ Stack Parameters
- ❖ Local Variables and Stack Frames
- ❖ Simplifying the Writing of Procedures
- ❖ Recursion
- ❖ Creating Multi-Module Programs

Multi-Module Programs

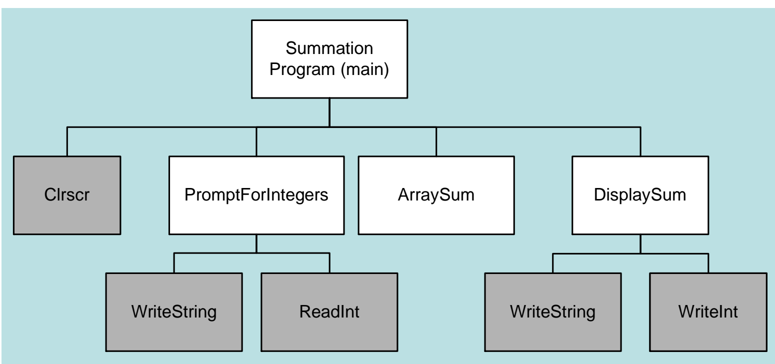
- Large ASM files are hard to manage
- You can divide a program into multiple modules
  - Each module is a separate ASM file
  - Each module is assembled into a separate OBJ file
- The linker combines all OBJ files into a single EXE file
  - This process is called static linking
  - You can also link OBJ files with one or more libraries
- Advantages:
  - A module can be a container for logically related code and data
  - Separate modules are easier to write, maintain, and debug
  - Modules can be reused in different programs if written properly
- PROTO and INVOKE simplify external procedure calls

Creating a Multi-Module Program

- Steps to follow when creating a multi-module program
  - Create the main module
  - Create a separate module for each set of related procedures
  - Create an include file that contains procedure prototypes for external procedures (ones that are called between modules)
  - Use the INCLUDE directive to make your procedure prototypes available to each module

Example: ArraySum Program

❖ Let's review the ArraySum program



Each of the four white rectangles will be a module

Main Module

```
TITLE Array Summation Main Module        (main.asm)
INCLUDE sum.inc

Count = 5     ; modify count to change array length

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

.code
main PROC
   call   Clrscr
   INVOKE ReadArray,  ADDR prompt1, ADDR array, Count
   INVOKE ArraySum,   ADDR array,   Count
   INVOKE DisplaySum, ADDR prompt2, eax
   call   Crlf
   INVOKE ExitProcess,0
main ENDP
END main
```

INCLUDE File

```
; Include file for the ArraySum Program      (sum.inc)

.686
.MODEL  FLAT, STDCALL
.STACK
INCLUDE Irvine32.inc

ReadArray PROTO,
    ptrPrompt:PTR BYTE,          ; prompt string
    ptrArray:PTR DWORD,          ; points to the array
    arraySize:DWORD              ; size of the array

ArraySum PROTO,
    ptrArray:PTR DWORD,          ; points to the array
    count:DWORD                  ; size of the array

DisplaySum PROTO,
    ptrPrompt:PTR BYTE,          ; prompt string
    theSum:DWORD                 ; sum of the array
```


Read Array Module

```
TITLE Read Array Module              (readArray.asm)
INCLUDE sum.inc

.code
ReadArray PROC USES ecx edx esi,
    prompt:PTR BYTE, array:PTR DWORD, arraysize:DWORD

    mov  ecx, arraysize
    mov  edx, prompt                 ; prompt address
    mov  esi, array                  ; array address

L1: call WriteString                 ; display string
    call ReadInt                     ; read integer into EAX
    call Crlf                        ; go to next output line
    mov  [esi],eax                   ; store in array
    add  esi,4                       ; next integer
    loop L1

    ret
ReadArray ENDP
END
```

Array Sum Module

```
TITLE ArraySum Module              (Arraysum.asm)
INCLUDE sum.inc

.code
ArraySum PROC USES ecx esi,
       ptrArray:PTR DWORD,         ; pointer to array
       arraySize:DWORD             ; size of array

       mov  eax, 0                 ; set the sum to zero
       mov  esi, ptrArray
       mov  ecx, arraySize
       cmp  ecx, 0                 ; array size <= 0?
       jle  L2                     ; yes: quit

L1:    add  eax, [esi]             ; add each integer to sum
       add  esi, 4                 ; point to next integer
       loop L1                     ; repeat for array size
L2:    ret                         ; return sum in EAX
ArraySum ENDP
END
```


Display Sum Module

```
TITLE Display Sum Module              (displaySum.asm)

INCLUDE Sum.inc

.code
DisplaySum PROC USES eax edx,
   ptrPrompt: PTR BYTE,               ; prompt string
   theSum: DWORD                      ; the array sum

   mov  edx, ptrPrompt                ; pointer to prompt
   call WriteString
   mov  eax, theSum
   call WriteInt                      ; display EAX
   call Crlf

   ret
DisplaySum ENDP
END
```

Custom Batch File

```
REM  makeArraySum.bat
REM  Batch file Customized for the Array Sum program

@echo off
cls

REM assembling ASM source modules into OBJ files
ML -Zi -c -Fl -coff main.asm displaySum.asm arraySum.asm readArray.asm

if errorlevel 1 goto terminate

REM linking object (OBJ) files to produce executable file
LINK32 main.obj displaySum.obj arraySum.obj readArray.obj irvine32.lib
kernel32.lib /OUT:arraySum.exe /SUBSYSTEM:CONSOLE /DEBUG

:terminate
pause
```


Summary

- Stack parameters
  - More convenient than register parameters
  - Passed by value or by reference
  - ENTER and LEAVE instructions
- Local variables
  - Created on the stack by decreasing the stack pointer
- MASM procedure-related directives
  - PROC, USES, LOCAL, PROTO, and INVOKE
  - Calling conventions (STDCALL, C)
- Recursive procedure calls
- Multi-Module Programs and Custom Make