Advanced Procedures

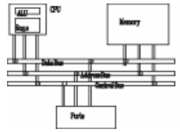
Computer Organization and Assembly Languages
Yung-Yu Chuang
2005/12/4

with slides by Kip Irvine

Overview

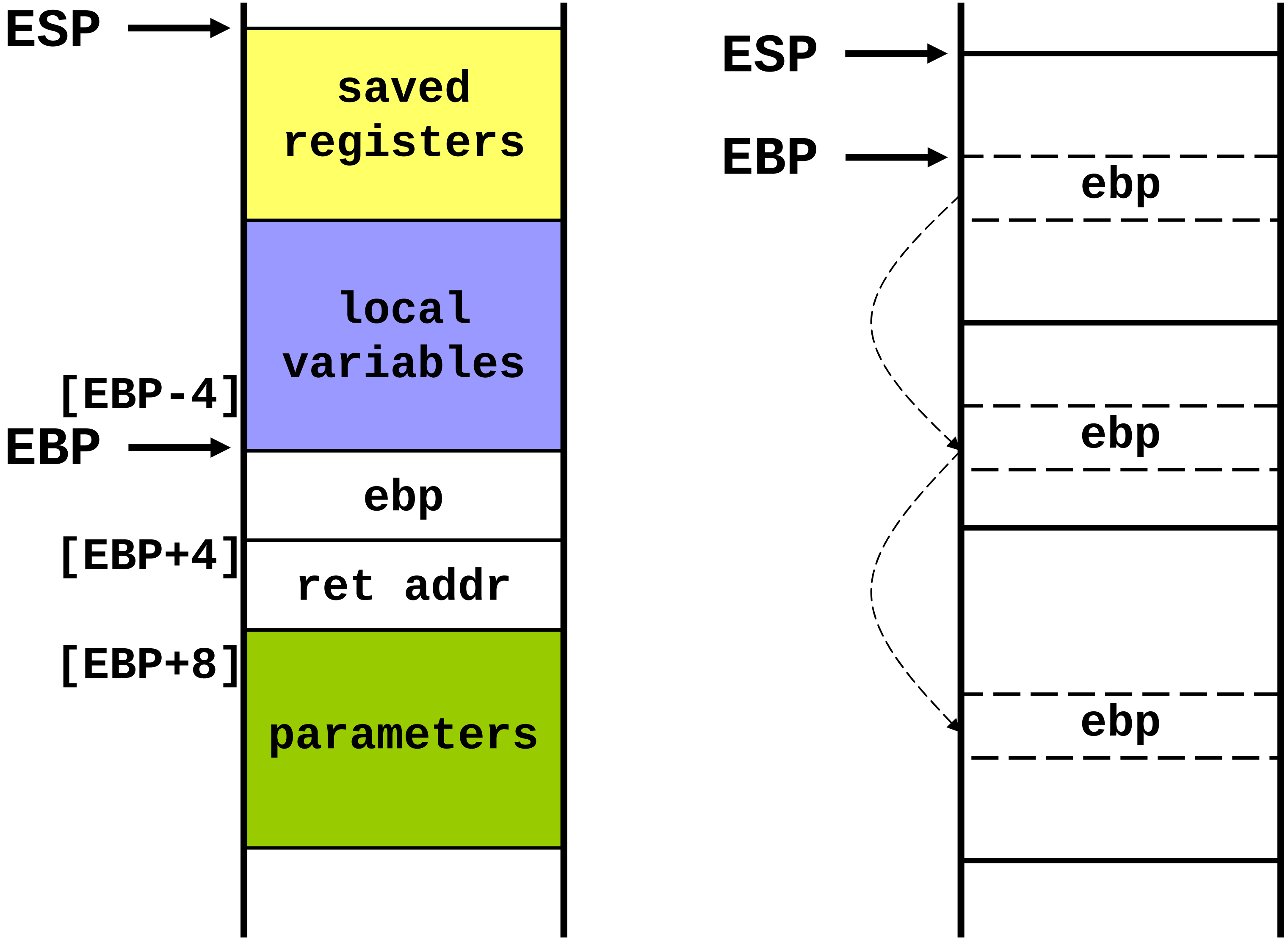
- Stack Frames (a communication protocol between high-level-language procedures)
- Stack Parameters (passing by value, passing by reference, memory model and language specifiers)
- Local Variables (creating and initializing on the stack, scope and lifetime, LOCAL)
- Recursion
- Related directives: INVOKE, PROC, PROTO
- Creating Multimodule Programs

Stack frame



- Also known as an activation record
- Area of the stack set aside for a procedure's return address, passed parameters, saved registers, and local variables
- Created by the following steps:
  – Calling procedure pushes *arguments* on the stack and calls the procedure.
  – The subroutine is called, causing the *return address* to be pushed on the stack.
  – The called procedure pushes *EBP* on the stack, and *sets EBP to ESP*.
  – If *local variables* are needed, a constant is subtracted from ESP to make room on the stack.
  – The *registers needed to be saved* are pushed.

Stack frame


ESP
saved
registers
local
variables
[EBP-4]
EBP
ebp
[EBP+4]
ret addr
[EBP+8]
parameters
ESP
EBP
ebp
ebp
ebp

Explicit access to stack parameters

- A procedure can explicitly access stack parameters using constant offsets from **EBP**.
  - Example: **`[ebp + 8]`**
- **EBP** is often called the base pointer or frame pointer because it holds the base address of the stack frame.
- **EBP** does not change value during the procedure.
- **EBP** must be restored to its original value when a procedure returns.

Parameters

- Two types: register parameters and stack parameters.
- Stack parameters are more convenient than register parameters.

```
pushad
mov esi,OFFSET array
mov ecx,LENGTHOF array
mov ebx,TYPE array
call DumpMem
popad
```

register parameters

```
push TYPE array
push LENGTHOF array
push OFFSET array
call DumpMem
```

stack parameters

Parameters

call by value	call by reference

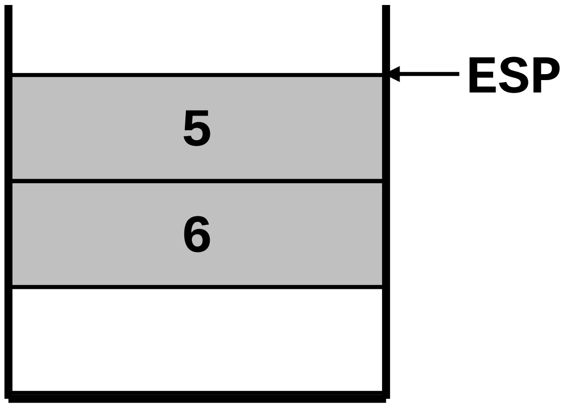
```
int sum=AddTwo(a, b);
```

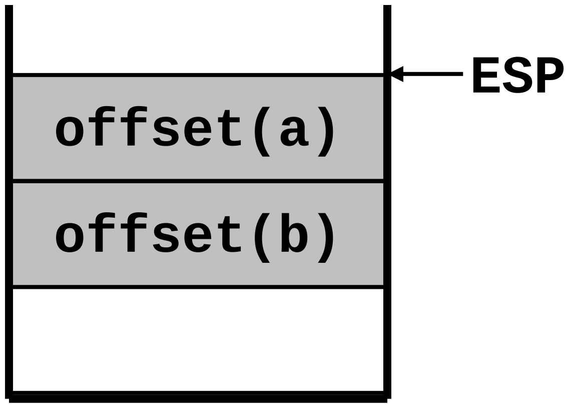
```
int sum=AddTwo(&a, &b);
```

```
.date
a   DWORD   5
b   DWORD   6
```

```
push b
push a
call AddTwo
```

```
push OFFSET b
push OFFSET a
call AddTwo
```

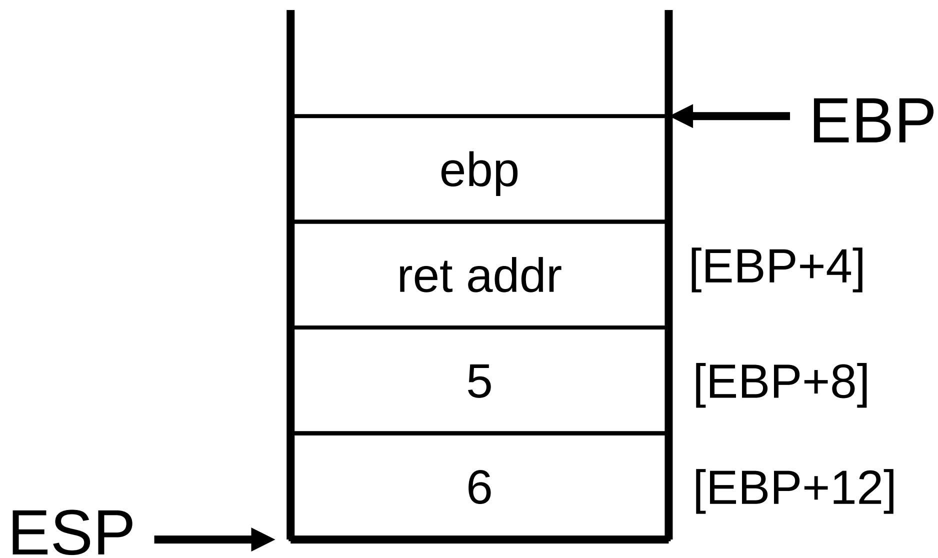




Stack frame example

```
.data
sum DWORD ?
.code
   push 6              ; second argument
   push 5              ; first argument
   call AddTwo         ; EAX = sum
   mov  sum,eax        ; save the sum
```

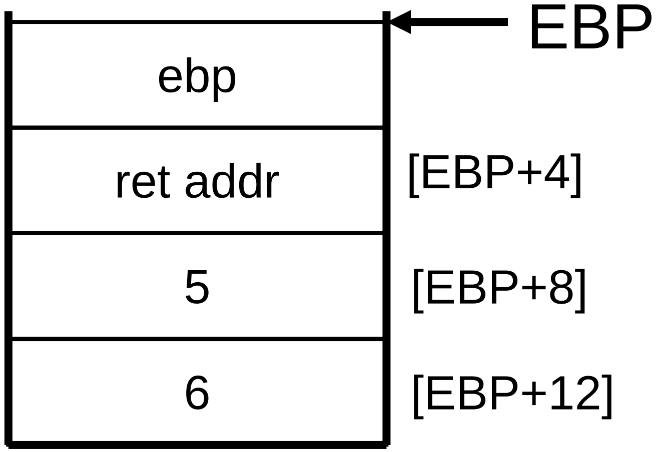
```
AddTwo PROC
   push ebp
   mov  ebp,esp
   .
   .
```



Stack frame example

```
AddTwo PROC
   push ebp
   mov ebp,esp            ; base of stack frame
   mov eax,[ebp + 12] ; second argument (6)
   add eax,[ebp + 8]  ; first argument (5)
   pop ebp
   ret 8                  ; clean up the stack
AddTwo ENDP              ; EAX contains the sum
```

Who should be responsible to remove arguments? It depends on the language model.



RET Instruction

- *Return from subroutine*
- Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.
- Syntax:
  - **RET**
  - **RET** *n*
- Optional operand *n* causes *n* bytes to be added to the stack pointer after EIP (or IP) is assigned a value.

Passing arguments by reference

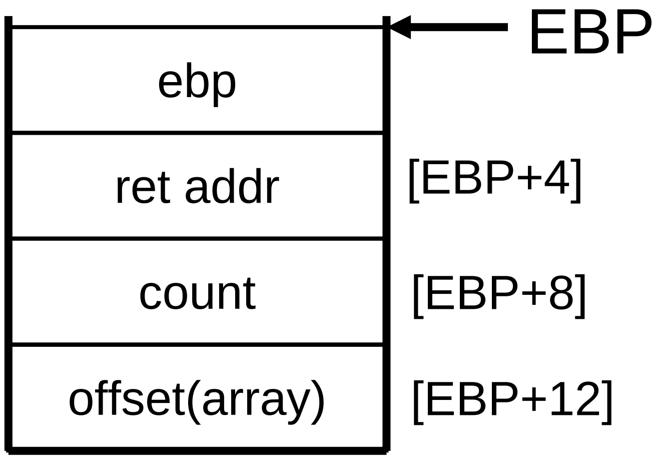
- The **`ArrayFill`** procedure fills an array with 16-bit random integers
- The calling program passes the address of the array, along with a count of the number of array elements:

```
.data
count = 100
array WORD count DUP(?)
.code
   push OFFSET array
   push COUNT
   call ArrayFill
```

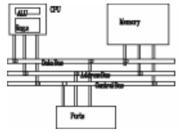
Passing arguments by reference

`ArrayFill` can reference an array without knowing the array's name:

```
ArrayFill PROC
   push ebp
   mov  ebp,esp
   pushad
   mov  esi,[ebp+12]
   mov  ecx,[ebp+8]
   .
   .
```



Passing 8-bit and 16-bit arguments



- When passing stack argements, it is best to push 32-bit operands to keep ESP aligned on a doubleword boundary.

```
Uppercase PROC
    push ebp
    mov  ebp, esp
    mov  al, [ebp+8]
    cmp  al, 'a'
    jb   L1
    cmp  al, 'z'
    ja   L1
    sub  al, 32
L1: pop  ebp
    ret  4
Uppercase ENDP
```

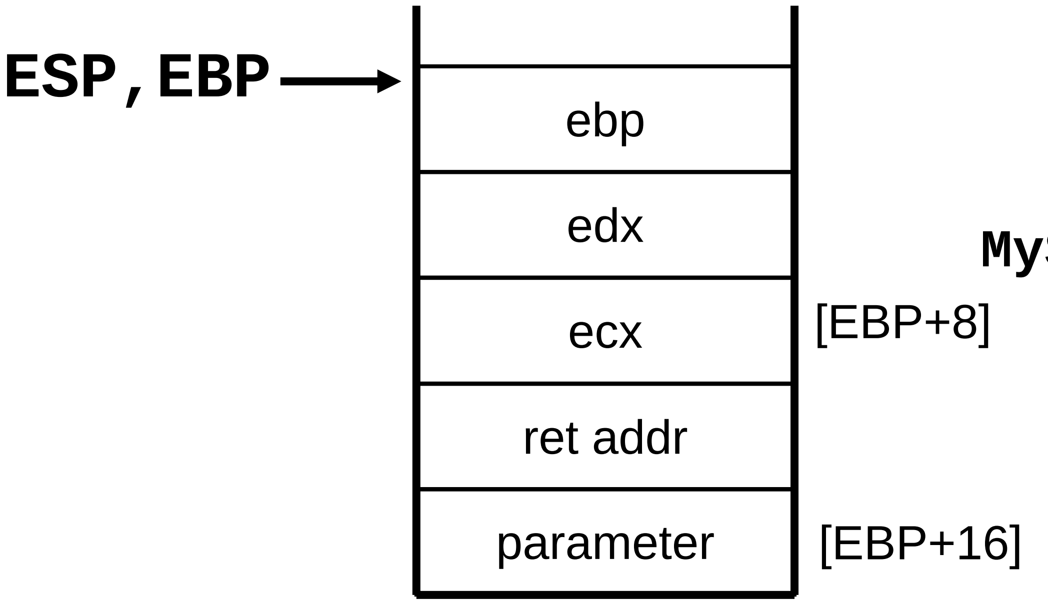
```
push  'x'  ; error
Call  Uppercase
```

```
.data
charVal  BYTE  'x'
.code
movzx eax, charVal
push  eax
Call  Uppercase
```

Saving and restoring registers

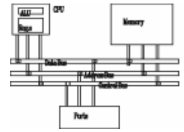
- When using stack parameters, avoid **USES**.

```
MySub2 PROC USES ecx, edx
  push ebp
  mov  ebp, esp
  mov  eax, [ebp+8]
  pop  ebp
  ret  4
MySub2 ENDP
```




```
MySub2 PROC
  push ecx
  push edx
  push ebp
  mov  ebp, esp
  mov  eax, [ebp+8]
  pop  ebp
  pop  edx
  pop  ecx
  ret  4
MySub2 ENDP
```

Local variables



- The variables defined in the data segment can be taken as *static global variables*.

  visibility=the whole program

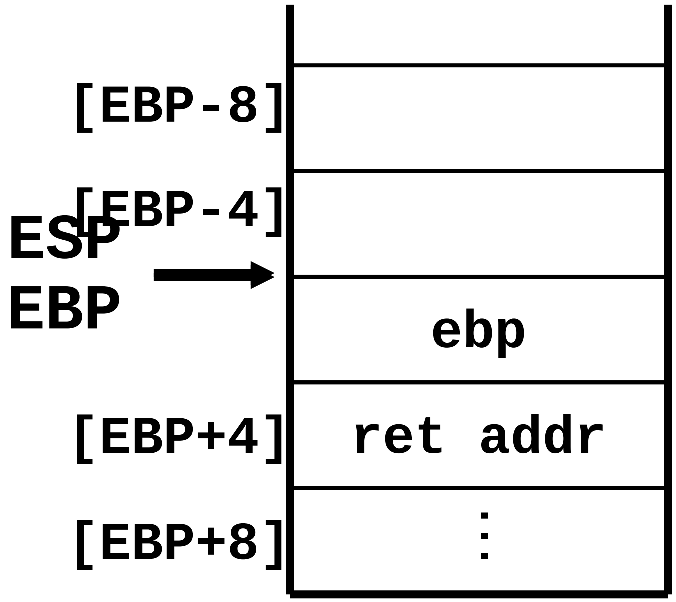
  lifetime=program duration

- A local variable is created, used, and destroyed within a single procedure (block)
- Advantages of local variables:
  - Restricted access: easy to debug, less error prone
  - Efficient memory usage
  - Same names can be used in two different procedures
  - Essential for recursion

Creating local variables

- Local variables are created on the runtime stack, usually above EBP.
- To explicitly create local variables, subtract their total size from ESP.

```
MySub PROC
   push ebp
   mov  ebp,esp
   sub  esp,8
   mov  [ebp-4],123456h
   mov  [ebp-8],0
   .
   .
```

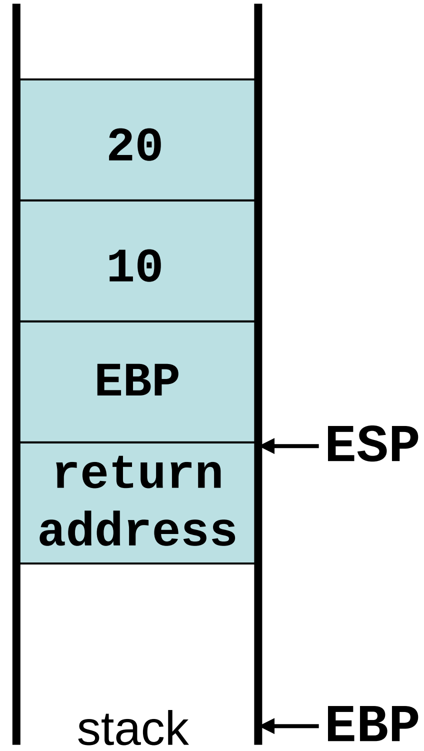


Local variables

- They can’t be initialized at assembly time but can be assigned to default values at runtime.

```
void MySub()
{
  int X=10;
  int Y=20;
  ...
}
```

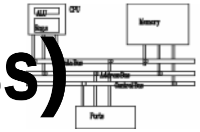
```
MySub PROC
  push ebp
  mov  ebp, esp
  sub  esp, 8
  mov  DWORD PTR [ebp-4], 10
  mov  DWORD PTR [ebp-8], 20
  ...
  mov  esp, ebp
  pop  ebp
  ret
MySub ENDP
```



Local variables

```
X_local EQU DWORD PTR [ebp-4]
Y_local EQU DWORD PTR [ebp-8]

MySub PROC
  push ebp
  mov  ebp, esp
  sub  esp, 8
  mov  X_local, 10
  mov  Y_local, 20
  ...
  mov  esp, ebp
  pop  ebp
  ret
MySub ENDP
```

LEA instruction (load effective address)



- The **LEA** instruction returns offsets of both direct and indirect operands.
  - **OFFSET** operator can only return constant offsets.
- **LEA** is required when obtaining the offset of a stack parameter or local variable. For example:

```
CopyString 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
```

LEA example

```
void makeArray()
{
  char myString[30];
  for (int i=0; i<30; i++)
    myString[i]='*';
}
```

```
makeArray PROC
    push ebp
    mov  ebp, esp
    sub  esp, 32
    lea  esi, [ebp-30]
    mov  ecx, 30
L1: mov  BYTE PTR [esi], '*'
    inc  esi
    loop L1
    add  esp 32
    pop  ebp
    ret
makeArray ENDP
```

ENTER and LEAVE

- **ENTER** instruction creates stack frame for a called procedure
  - pushes EBP on the stack **push ebp**
  - set EBP to the base of stack frame **mov ebp, esp**
  - reserves space for local variables **sub esp, n**
- **ENTER nbytes, nestinglevel**
  - **nbytes** (for local variables) is rounded up to a multiple of 4 to keep ESP on a doubleword boundary
  - **nestinglevel**: 0 for now

```
MySub PROC
   enter 8,0
```

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

ENTER and LEAVE

- **LEAVE** reverses the action of a previous **ENTER** instruction.

```
MySub PROC
  enter 8, 0
  .
  .
  .
  .
  leave
  ret
MySub ENDP
```

```
MySub PROC
  push ebp
  mov  ebp, esp
  sub  esp, 8
  .
  .
  mov  esp, ebp
  pop  ebp
  ret
MySub ENDP
```

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*

Example:

```
MySub PROC
   LOCAL var1:BYTE, var2:WORD, var3:SDWORD
```

MASM-generated code

```
BubbleSort PROC
   LOCAL temp:DWORD, SwapFlag:BYTE
   . . .
   ret
BubbleSort ENDP
```

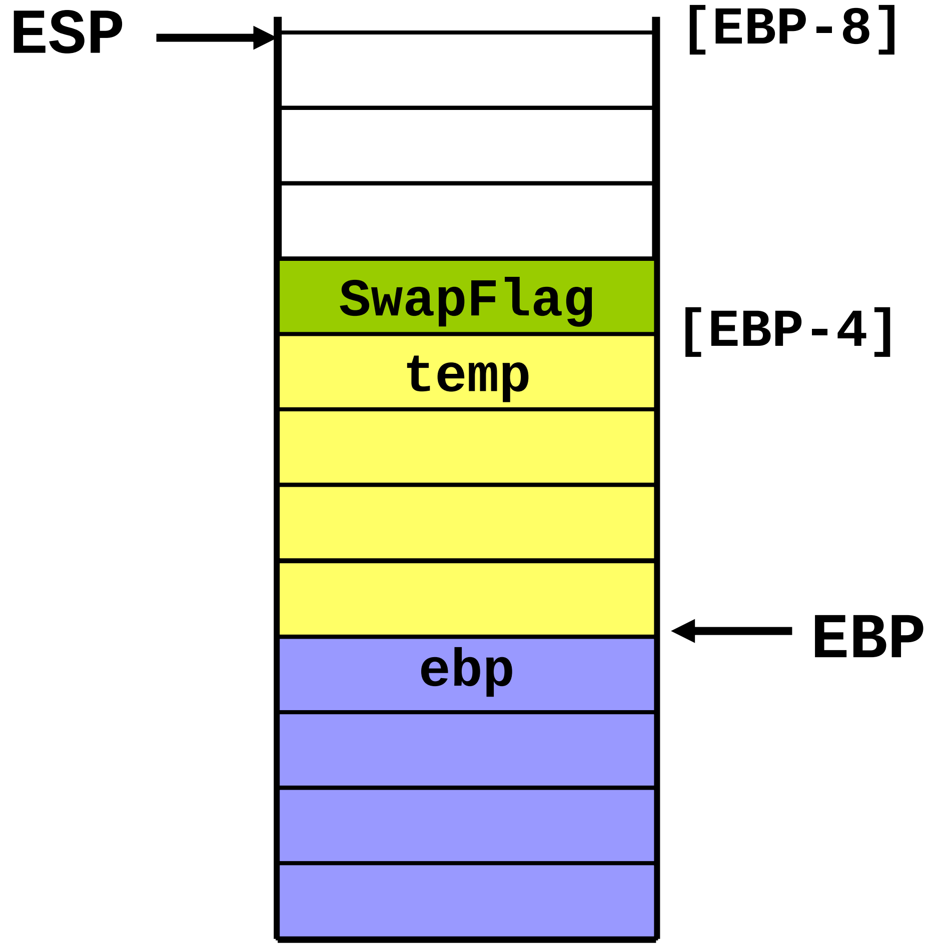
MASM generates the following code:

```
BubbleSort PROC
   push ebp
   mov  ebp,esp
   add  esp,0FFFFFFF8h ; add -8 to ESP
   . . .
   mov  esp,ebp
   pop  ebp
   ret
BubbleSort ENDP
```

Non-Doubleword Local Variables

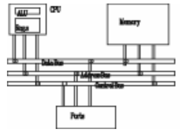
- Local variables can be different sizes
- How created in the stack by **LOCAL** directive:
  - 8-bit: assigned to next available byte
  - 16-bit: assigned to next even (word) boundary
  - 32-bit: assigned to next doubleword boundary

MASM-generated code




```
mov   eax, temp          →    mov   eax, [ebp-4]
mov   bl, SwapFlag            mov   bl, [ebp-5]
```

Reserving stack space



- .STACK 4096
- **`Sub1`** calls **`Sub2`**, **`Sub2`** calls **`Sub3`**, how many bytes will you need in the stack?

```
Sub1 PROC
  LOCAL array1[50]:DWORD ; 200 bytes

Sub2 PROC
  LOCAL array2[80]:WORD  ; 160 bytes

Sub3 PROC
  LOCAL array3[300]:WORD ; 300 bytes
```

660+8(ret addr)+saved registers…

WriteStackFrame Procedure

- Displays contents of current stack frame

```
WriteStackFrame PROTO,
  numParam:DWORD, ; # of passed parameters
  numLocalVal: DWORD, ; # of DWordLocal
                      ; variables
  numSavedReg: DWORD  ; # of saved registers
```

WriteStackFrame Example

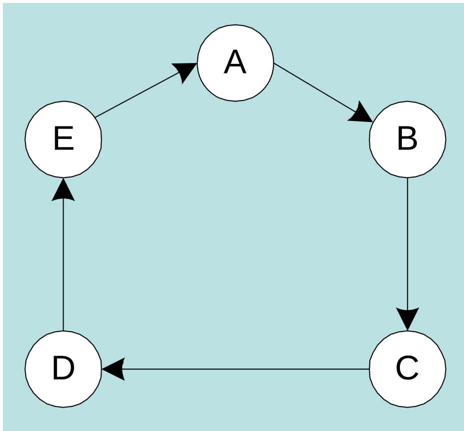
```
main PROC
    mov eax, 0EAEAEAEAh
    mov ebx, 0EBEBEBEBh
    INVOKE aProc, 1111h, 2222h
    exit
main ENDP

aProc PROC USES eax ebx,
    x: DWORD, y: DWORD
    LOCAL a:DWORD, b:DWORD
    PARAMS = 2
    LOCALS = 2
    SAVED_REGS = 2
    mov a,0AAAAh
    mov b,0BBBBh
    INVOKE WriteStackFrame, PARAMS, LOCALS, SAVED_REGS
```

WriteStackFrame Example

```
Stack Frame
00002222  ebp+12 (parameters)
00001111  ebp+8 (parameters)
00401083  ebp+4 (return address)
0012FFF0  ebp+0 (saved ebp) ← ebp
0000AAAA  ebp-4 (local variable)
0000BBBB  ebp-8 (local variable)
EAEAEAEA  ebp-12 (saved register)
EBEBEBEB  ebp-16 (saved register) ← esp
```

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:

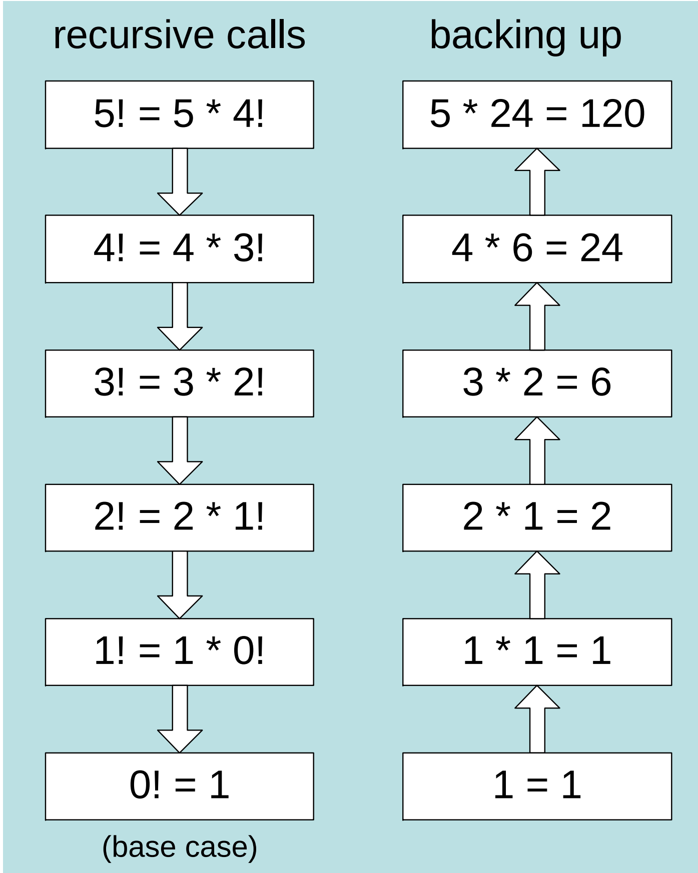


Calculating a factorial

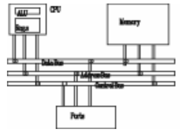
This function calculates the factorial of integer *n*.
A new value of *n* is saved in each stack frame:

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

```
      factorial(5);
```



Calculating a factorial

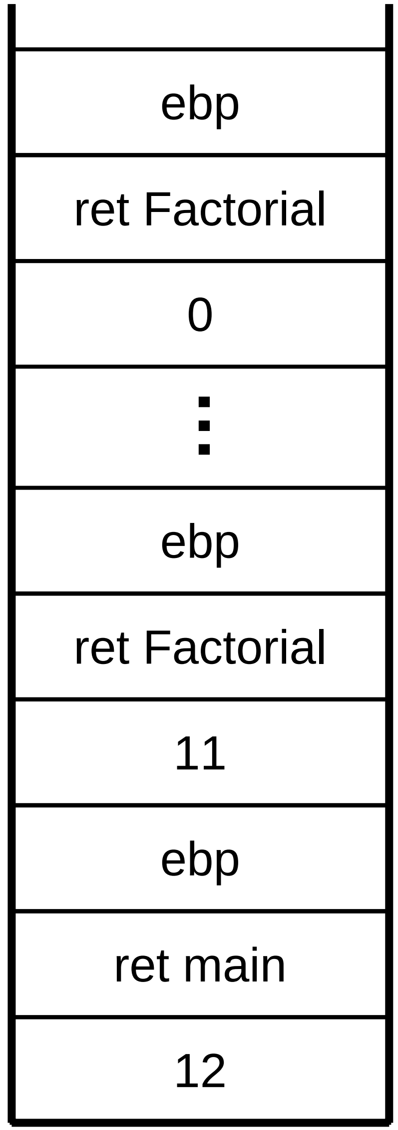


```
Factorial PROC
   push ebp
   mov  ebp,esp
   mov  eax,[ebp+8]      ; get n
   cmp  eax,0            ; n > 0?
   ja   L1               ; yes: continue
   mov  eax,1            ; no: return 1
   jmp  L2
L1:dec  eax
   push eax              ; Factorial(n-1)
   call Factorial

ReturnFact:
   mov  ebx,[ebp+8]        ; get n
   mul  ebx                ; edx:eax=eax*ebx

L2:pop  ebp              ; return EAX
   ret  4                ; clean up stack
Factorial ENDP
```

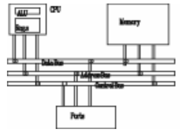
Calculating a factorial

```
push 12
call Factorial
```

```
Factorial PROC
   push ebp
   mov  ebp,esp
   mov  eax,[ebp+8]
   cmp  eax,0
   ja   L1
   mov  eax,1
   jmp  L2
L1:dec  eax
   push eax
   call Factorial

ReturnFact:
   mov  ebx,[ebp+8]
   mul  ebx

L2:pop  ebp
   ret  4
Factorial ENDP
```



.MODEL directive



- **.MODEL** directive specifies a program's memory model and model options (language-specifier).
- Syntax:

  .MODEL *memorymodel* [,*modeloptions*]

- ***memorymodel*** can be one of the following:
  – tiny, small, medium, compact, large, huge, or flat
- ***modeloptions*** includes the language specifier:
  – procedure naming scheme
  – parameter passing conventions
- **.MODEL flat, STDCALL**

Memory models

- A program's memory model determines the number and sizes of code and data segments.
- Real-address mode supports tiny, small, medium, compact, large, and huge models.
- Protected mode supports only the flat model.

Small model: code < 64 KB, data (including stack) < 64 KB.
All offsets are 16 bits.

Flat model: single segment for code and data, up to 4 GB.
All offsets are 32 bits.

Language specifiers

- STDCALL (used when calling Windows functions)
  - procedure arguments pushed on stack in reverse order (right to left)
  - called procedure cleans up the stack
  - **`_name@nn`** (for example, **`_AddTwo@8`**)
- C
  - procedure arguments pushed on stack in reverse order (right to left)
  - calling program cleans up the stack (variable number of parameters such as **`printf`**)
  - _name (for example, **`_AddTwo`**)
- PASCAL
  - arguments pushed in forward order (left to right)
  - called procedure cleans up the stack
- BASIC, FORTRAN, SYSCALL

INVOKE directive

- The **`INVOKE`** directive is a powerful replacement for Intel's **`CALL`** instruction that lets you pass multiple arguments
- Syntax:

  `INVOKE` ***`procedureName`*** **`[,`** ***`argumentList`*** **`]`**
- ***`ArgumentList`*** is an optional comma-delimited list of procedure arguments
- Arguments can be:
  - immediate values and integer expressions
  - variable names
  - address and ADDR expressions
  - register names

INVOKE examples

```
.data
byteVal BYTE 10
wordVal WORD 1000h
.code
   ; direct operands:
   INVOKE Sub1,byteVal,wordVal

   ; address of variable:
   INVOKE Sub2,ADDR byteVal

   ; register name, integer expression:
   INVOKE Sub3,eax,(10 * 20)

   ; address expression (indirect operand):
   INVOKE Sub4,[ebx]
```

INVOKE example

```
.data
val1 DWORD 12345h
val2 DWORD 23456h
.code
  INVOKE AddTwo, val1, val2

push val1
push val2
call AddTwo
```

ADDR operator

- Returns a near or far pointer to a variable, depending on which memory model your program uses:
  - Small model: returns 16-bit offset
  - Large model: returns 32-bit segment/offset
  - Flat model: returns 32-bit offset
- Simple example:

```
.data
myWord WORD ?
.code
INVOKE mySub,ADDR myWord
```

ADDR example

```
.data
Array DWORD 20 DUP(?)
.code
...
INVOKE Swap, ADDR Array, ADDR [Array+4]
```

```
push OFFSET Array+4
push OFFSET Array
Call Swap
```

PROC directive

- The **PROC** directive declares a procedure with an optional list of named parameters.
- Syntax:

  label **PROC [attributes] [USES] paramList**

- ***paramList*** is a list of parameters separated by commas. Each parameter has the following syntax:

  paramName:*type*

  type must either be one of the standard ASM types (BYTE, SBYTE, WORD, etc.), or it can be a pointer to one of these types.

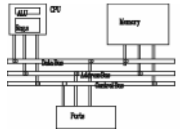
- Example: **foo PROC C USES eax, param1:DWORD**

PROC example

- The AddTwo procedure receives two integers and returns their sum in EAX.
- C++ programs typically return 32-bit integers from functions in EAX.

```
AddTwo PROC,
 val1:DWORD,
 val2:DWORD

 mov eax,val1
 add eax,val2
 ret
AddTwo ENDP
```

```
AddTwo PROC,
 push ebp
 mov  ebp, esp
 mov  eax, dword ptr [ebp+8]
 add  eax, dword ptr [ebp+0Ch]
 leave
 ret 8
AddTwo ENDP
```

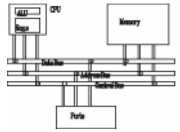


PROC example

```
Read_File PROC USES eax, ebx,
  pBuffer:PTR BYTE
  LOCAL fileHandle:DWORD

  mov  esi, pBuffer
  mov  fileHandle, eax
  .
  .
  ret
Read_File ENDP
```

```
Read_File PROC
  push ebp
  mov  ebp, esp
  add  esp, 0FFFFFFFCh
  push eax
  push ebx
  mov  esi, dword ptr [ebp+8]
  mov  dword ptr [ebp-4], eax
  .
  .
  pop  ebx
  pop  eax
  ret
Read_File ENDP
```

PROTO directive



- Creates a procedure prototype
- Syntax:
  - ***label*** **PROTO** ***paramList***
- Every procedure called by the **INVOKE** directive must have a prototype
- A complete procedure definition can also serve as its own prototype

PROTO directive

- Standard configuration: **PROTO** appears at top of the program listing, INVOKE appears in the code segment, and the procedure implementation occurs later in the program:

```
MySub PROTO       ; procedure prototype

.code
INVOKE MySub      ; procedure call


MySub PROC        ; procedure implementation
   .
   .
MySub ENDP
```

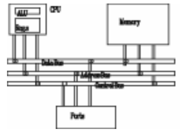
PROTO example

- Prototype for the ArraySum procedure, showing its parameter list:

```
ArraySum PROTO,
   ptrArray:PTR DWORD, ; points to the array
   szArray:DWORD       ; array size
```

```
ArraySum PROC USES esi, ecx,
   ptrArray:PTR DWORD, ; points to the array
   szArray:DWORD       ; array size
```

Parameter classifications



- An input parameter is data passed by a calling program to a procedure.
  - The called procedure is not expected to modify the corresponding parameter variable, and even if it does, the modification is confined to the procedure itself.
- An output parameter is created by passing a pointer to a variable when a procedure is called.
  - The procedure does not use any existing data from the variable, but it fills in a new value before it returns.
- An input-output parameter represents a value passed as input to a procedure, which the procedure may modify.
  - The same parameter is then able to return the changed data to the calling program.

Example: exchanging two integers

The Swap procedure exchanges the values of two 32-bit integers. pValX and pValY do not change values, but the integers they point to are modified.

```
Swap PROC USES eax esi edi,
   pValX:PTR DWORD, ; pointer to first integer
   pValY:PTR DWORD  ; pointer to second integer

   mov esi,pValX    ; get pointers
   mov edi,pValY
   mov eax,[esi]    ; get first integer
   xchg eax,[edi]   ; exchange with second
   mov [esi],eax    ; replace first integer
   ret  ; MASM changes it to ret 8 due to PROC
Swap ENDP
```

Multimodule programs

- A multimodule program is a program whose source code has been divided up into separate ASM files.
- Each ASM file (module) is assembled into a separate OBJ file.
- All OBJ files belonging to the same program are linked using the link utility into a single EXE file.
  - This process is called static linking

Advantages

- Large programs are easier to write, maintain, and debug when divided into separate source code modules.
- When changing a line of code, only its enclosing module needs to be assembled again. Linking assembled modules requires little time.
- A module can be a container for logically related code and data
  - encapsulation: procedures and variables are automatically hidden in a module unless you declare them public

Creating a multimodule program

- Here are some basic steps to follow when creating a multimodule program:
  – Create the main module
  – Create a separate source code module for each procedure or 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

Multimodule programs

- `MySub PROC PRIVATE`

  `Sub PROC PUBLIC`

- `EXTERN sub1@0:PROC`

- `PUBLIC count, SYM1`

  `SYM1=10`

  `.data`

  `Count DWORD 0`

- `EXTERN name:type`

INCLUDE file

The sum.inc file contains prototypes for external functions that are not in the Irvine32 library:

```
INCLUDE Irvine32.inc

PromptForIntegers 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
```

Main.asm

```
TITLE Integer Summation Program

INCLUDE sum.inc

.code
main PROC
   call Clrscr

   INVOKE PromptForIntegers,
      ADDR prompt1,
      ADDR array,
      Count


      ...
      call Crlf
      INVOKE ExitProcess,0
main ENDP
END main
```