Procedure

*Computer Organization and Assembly Languages*

*Yung-Yu Chuang*

*with slides by Kip Irvine*
Overview

- Stack Operations
- Defining and Using Procedures
- Stack frames, parameters and local variables
- Recursion
- Related directives
Stack operations
Stacks

- LIFO (Last-In, First-Out) data structure.
- push/pop operations
- You probably have had experiences on implementing it in high-level languages.
- Here, we concentrate on *runtime stack*, directly supported by hardware in the CPU. It is essential for calling and returning from procedures.
Runtime stack

- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) *: point to the top of the stack usually modified by CALL, RET, PUSH and POP

* SP in Real-address mode
PUSH and POP instructions

• **PUSH** syntax:
  - PUSH \( r/m16 \)
  - PUSH \( r/m32 \)
  - PUSH \( imm32 \)

• **POP** syntax:
  - POP \( r/m16 \)
  - POP \( r/m32 \)
**PUSH operation (1 of 2)**

- A push operation decrements the stack pointer by 2 or 4 (depending on operands) and copies a value into the location pointed to by the stack pointer.
**PUSH operation (2 of 2)**

- The same stack after pushing two more integers:

```
PUSH 01h
00000001
000000A5
00000006

PUSH 02h
00000002
00000001
000000A5
00000006
```
**POP operation**

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

\[
\begin{array}{c|c}
0\text{FEC} & 0\text{FF8} \\
0\text{FF0} & 0\text{FFC} \\
0\text{FF4} & \\
\hline
\text{ESP} & \\
\end{array}
\]

\[
\begin{array}{c|c}
0\text{FEC} & 0\text{FF8} \\
0\text{FF0} & 0\text{FFC} \\
0\text{FF4} & \\
\hline
\text{ESP} & \\
\end{array}
\]

\[
\begin{array}{c|c}
0\text{FEC} & 0\text{FF8} \\
0\text{FF0} & 0\text{FFC} \\
0\text{FF4} & \\
\hline
\text{ESP} & \\
\end{array}
\]

\[
\begin{array}{c|c}
0\text{FEC} & 0\text{FF8} \\
0\text{FF0} & 0\text{FFC} \\
0\text{FF4} & \\
\hline
\text{ESP} & \\
\end{array}
\]

**POP EAX**

\[
\begin{array}{c|c}
0\text{FEC} & 0\text{FF8} \\
0\text{FF0} & 0\text{FFC} \\
0\text{FF4} & \\
\hline
\text{ESP} & \\
\end{array}
\]

\[
\begin{array}{c|c}
0\text{FEC} & 0\text{FF8} \\
0\text{FF0} & 0\text{FFC} \\
0\text{FF4} & \\
\hline
\text{ESP} & \\
\end{array}
\]

**EAX=00000002**
When to use stacks

- Temporary save area for registers
- To save return address for CALL
- To pass arguments
- Local variables
- Applications which have LIFO nature, such as reversing a string
Example of using stacks

Save and restore registers when they contain important values. Note that the `PUSH` and `POP` instructions are in the opposite order:

```
push esi          ; push registers
push ecx
push ebx

mov esi,OFFSET dwordVal  ; starting OFFSET
mov ecx,LENGTHOF dwordVal; number of units
mov ebx,TYPE dwordVal ;size of a doubleword
call DumpMem        ; display memory

pop ebx          ; opposite order
pop ecx
pop esi
```
Example: 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
```
Example: reversing a string

.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
Example: reversing a string

; 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

exit
main ENDP
END main
Related instructions

- **PUSHFD** and **POPFD**
  - push and pop the EFLAGS register
  - **LAHF**, **SAHF** are other ways to save flags

- **PUSHAD** pushes the 32-bit general-purpose registers on the stack in the following 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
Example

MySub  PROC
    pushad
    ...
    ; modify some register
    ...
    popad
    ret
MySub  ENDP

Do not use this if your procedure uses registers for return values
Defining and using procedures
Creating Procedures

- Large problems can be divided into smaller tasks to make them more manageable
- A procedure is the ASM equivalent of a Java or C++ function
- Following is an assembly language procedure named sample:

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

A named block of statements that ends with a return.
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.

For example, a procedure of drawing lines could assume that display adapter is already in graphics mode.
Example: SumOf procedure

;-------------------------------
SumOf PROC
;
; 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
;
;-------------------------------
    add eax, ebx
    add eax, ecx
    ret
SumOf ENDP
CALL and RET instructions

• The **CALL** instruction calls a procedure
  - pushes offset of next instruction on the stack
  - copies the address of the called procedure into **EIP**

• The **RET** instruction returns from a procedure
  - pops top of stack into **EIP**

• We used **jl** and **jr** in our toy computer for **CALL** and **RET**, **BL** and **MOV PC, LR** in ARM.
CALL-RET example (1 of 2)

00000025 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 (2 of 2)

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

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

Stack

EIP
Local and global labels

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

```plaintext
main PROC
  jmp L2
L1::
  exit
main ENDP

sub2 PROC
L2:
  jmp L1
  ret
sub2 ENDP
```

; error!
; global label

; local label
; ok
Procedure parameters (1 of 3)

• A good procedure might be usable in many different programs

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

• General registers can be used to pass parameters
The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names:

```
ArraySum PROC
    mov esi,0 ; array index
    mov eax,0 ; set the sum to zero
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
```
This version returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

```
ArraySum PROC
    ; Recevies: ESI points to an array of doublewords,
    ;           ECX = number of array elements.
    ; Returns:  EAX = sum
    ;-----------------------------------------------
    push esi
    push 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
    pop ecx
    pop esi
    ret
ArraySum ENDP
```
Calling ArraySum

.data
array DWORD 10000h, 20000h, 30000h, 40000h
thesum DWORD ?

.code
main PROC
    mov esi, OFFSET array
    mov ecx, LENGTHOF array
    call ArraySum
    mov theSum, eax
**USES operator**

- Lists the registers that will be saved (to avoid side effects) (return register shouldn’t be saved)

```
ArraySum PROC USES esi ecx
    mov eax,0 ; set the sum to zero
    ...

MASM generates the following code:
ArraySum PROC
    push esi
    push ecx
    ...
    pop ecx
    pop esi
    ret
ArraySum ENDP
```
Stack frames, parameters and local variables
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

[EBP-4] EBP → local variables

[EBP+4] ebp

[EBP+8] ret addr

[EBP+8] parameters

ESP →

EBP → ebp

ebp

ebp

33
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
```

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

register parameters  
stack parameters
Parameters

**call by value**

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

**call by reference**

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

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

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

```plaintext
push offset(b)
push offset(a)
call AddTwo
```

ESP

ESP
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
.
.
ret addr
5
6

EBP
ESP
[EBP+4]
[EBP+8]
[EBP+12]
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:

```assembly
.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]
    .
    .
```

Diagram:

```
EBP

<table>
<thead>
<tr>
<th>ebp</th>
<th>[EBP+4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret addr</td>
<td>[EBP+8]</td>
</tr>
<tr>
<td>count</td>
<td>[EBP+12]</td>
</tr>
<tr>
<td>offset(array)</td>
<td></td>
</tr>
</tbody>
</table>
```
Passing 8-bit and 16-bit arguments

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

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

ESP,EBP →

ebp
edx
ecx
ret addr
parameter

[EBP+8]
[EBP+16]
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
    .
    .
    ret addr
```

Diagram:

[EIP]  [EBP+8]
|             |
[EIP-4]  [EBP+4]
|             |
[EIP-8]  [EBP-4]
|     esp     |
|     ebp     |
|     .        |
|     .        |
|     .        |
Local variables

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

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

```assembly
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 at run time.
  - **OFFSET** only returns constant offsets (assemble time).
- **LEA** is required when obtaining the offset of a stack parameter or local variable. For example:

```assembly
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 
```


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 are they 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  bl, SwapFlag
mov  eax, [ebp-4]
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...
Recursion
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:

```c
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

ebp
ret Factorial
0
::
ebp
ret Factorial
11
ebp
ret main
12
Related directives
.MODEL directive

- .MODEL directive specifies a program's memory model and model options (language-specifier).
- Syntax:
  \[
  \text{.MODEL } \text{memorymodel} [,\text{modeloptions}]\]
- \text{memorymodel} can be one of the following:
  - tiny, small, medium, compact, large, huge, or flat
- \text{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:
  
  \[
  \text{INVOKE } \text{procedureName} \ [, \ \text{argumentList}]
  \]

- **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:

```plaintext
.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:
  
  \[ label \text{ PROC} \ [\text{attributes}] \ [\text{USES}] \ \text{paramList} \]

• \text{paramList} is a list of parameters separated by commas. Each parameter has the following syntax:

  \[ \text{paramName:type} \]

  \text{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: \text{foo PROC C USES eax, param1:DWORD}
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

```assembly
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
• 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
```
Multimodule programs
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
  sub1 PROC PUBLIC

- EXTERN sub1@0:PROC

- PUBLIC count, SYM1
  SYM1=10
  .data
  count DWORD 0

- EXTERN name:type
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
```
TITLE Integer Summation Program

INCLUDE sum.inc

.code
main PROC
  call Clnscr

  INVOKE PromptForIntegers, 
    ADDR prompt1, 
    ADDR array, 
    Count 

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