Assembly Language for x86 Processors

7th Edition

Kip R. Irvine

Chapter 8: Advanced Procedures

Chapter Overview

- **Stack Frames**
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

Stack Frames

- Stack Parameters
- Local Variables
- ENTER and LEAVE Instructions
- LOCAL Directive
- WriteStackFrame Procedure

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 program pushes arguments on the stack and calls the procedure.
  - The called procedure pushes EBP on the stack, and sets EBP(基址指针寄存器) to ESP(栈指针寄存器).
  - If local variables are needed, a constant is subtracted from ESP to make room on the stack.

Stack Parameters

- More convenient than register parameters
- Two possible ways of calling DumpMem. Which is easier?

```
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

;Push varible, address
```

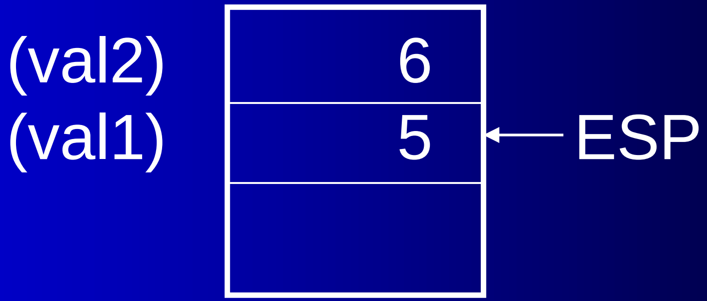
Passing Arguments by Value

- Push argument values on stack
  - (Use only 32-bit values in protected mode to keep the stack aligned)
- Call the called-procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called-procedure did not remove them

Example

```
.data
val1   DWORD  5
val2   DWORD  6

.code
push val2
push val1
```



Stack prior to CALL
Constants can be pushed

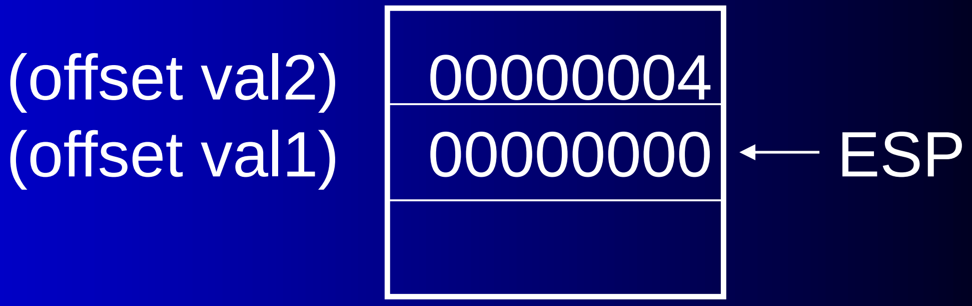
Passing by Reference

- Push the offsets of arguments on the stack
- Call the procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called procedure did not remove them

Example

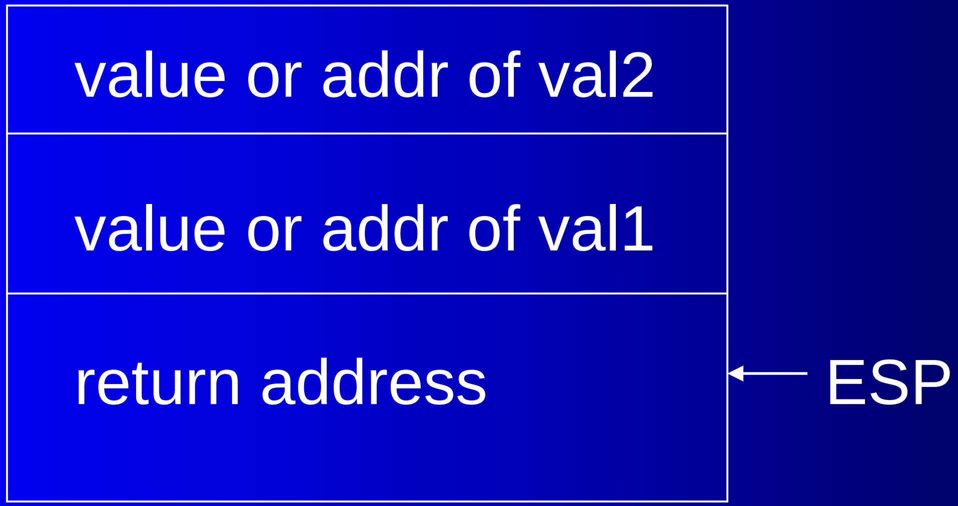
```
.data
val1   DWORD  5
val2   DWORD  6

.code
push OFFSET val2
push OFFSET val1
```



Stack prior to CALL

Stack after the CALL



Passing an Array by Reference (1 of 2)

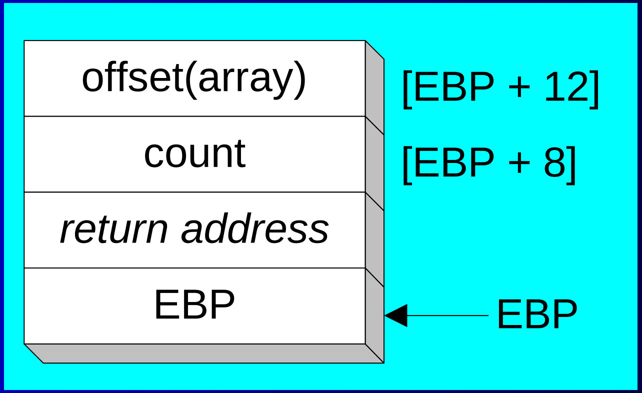
- The ArrayFill procedure fills an array with 16-bit random integers
- The calling program passes the address of the array, along with a count of the number of array elements:

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

Passing an Array by Reference (2 of 2)

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]
     .
     .
```



ESI points to the beginning of the array, so it's easy to use a loop to access each array element. View the complete program.

Accessing Stack Parameters (C/C++)

- C and C++ functions access stack parameters using constant offsets from EBP[1].
  - Example: [ebp + 8]
- EBP is called the base pointer or frame pointer because it holds the base address of the stack frame.
- EBP does not change value during the function.
- EBP must be restored to its original value when a function returns.

[1] BP in Real-address mode

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.

Who removes parameters from the stack?

Caller (C) ...... or ...... Called-procedure (STDCALL):

```
push val2
push val1
call AddTwo
add   esp,8
```

```
AddTwo PROC
    push  ebp
    mov   ebp,esp
    mov   eax,[ebp+12]
    add   eax,[ebp+8]

    pop   ebp
    ret   8
```

(Covered later: The MODEL directive specifies calling conventions)

```
Main proc
      Call example
……
Main endp


Example proc
      push val2
      push val1
      call AddTwo
      Ret
Example endp
```

Your turn . . .

- Create a procedure named Difference that subtracts the first argument from the second one. Following is a sample call:

```
push 14                         ; first argument
push 30                         ; second argument
call Difference                 ; EAX = 16
```

```
Difference PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp + 8]       ; second argument
    sub  eax,[ebp + 12]      ; first argument
    pop  ebp
    ret  8
Difference ENDP
```

Passing 8-bit and 16-bit Arguments

- Cannot push 8-bit values on stack
- Pushing 16-bit operand may cause page fault or ESP alignment problem
  - incompatible with Windows API functions
- Expand smaller arguments into 32-bit values, using MOVZX or MOVSX:

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

Passing Multiword Arguments

- Push high-order values on the stack first; work backward in memory
- Results in little-endian ordering of data
- Example:

```
.data
longVal DQ 1234567800ABCDEFh
.code
  push  DWORD PTR longVal + 4     ; high doubleword
  push  DWORD PTR longVal         ; low doubleword
  call  WriteHex64
```

Saving and Restoring Registers

- Push registers on stack just after assigning ESP to EBP
  - local registers are modified inside the procedure

```
MySub PROC
  push  ebp
  mov   ebp,esp
  push  ecx              ; save local registers
  push  edx
```

Stack Affected by USES Operator

```
MySub1 PROC USES ecx edx
   ret
MySub1 ENDP
```

- USES operator generates code to save and restore registers:

```
MySub1 PROC
   push  ecx
   push  edx

   pop   edx
   pop   ecx
ret
```

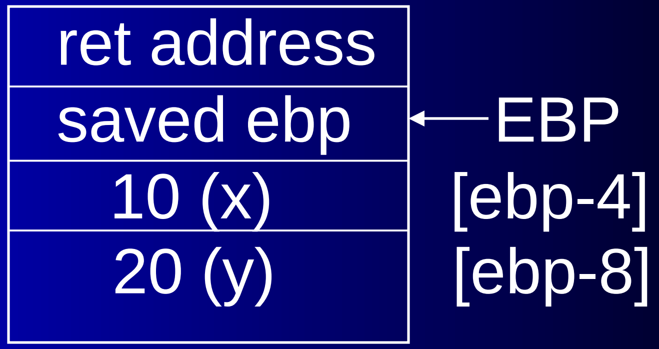
Local Variables

- Only statements within subroutine can view or modify local variables
- Storage used by local variables is released when subroutine ends
- local variable name can have the same name as a local variable in another function without creating a name clash
- Essential when writing recursive procedures, as well as procedures executed by multiple execution threads

Creating LOCAL Variables

Example - create two DWORD local variables:

Say: int x=10, y=20;



```
MySub PROC
   push     ebp
   mov      ebp,esp
   sub      esp,8               ;create 2 DWORD variables

   mov      DWORD PTR [ebp-4],10 ; initialize x=10
   mov      DWORD PTR [ebp-8],20 ; initialize y=20
```

LEA Instruction

- LEA returns offsets of direct and indirect operands
  - OFFSET operator only returns constant offsets
- LEA required when obtaining offsets of stack parameters & local variables
- 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

Suppose you have a Local variable at [ebp-8]

And you need the address of that local variable in ESI

You cannot use this:

```
mov esi, OFFSET [ebp-8]        ; error
```

Use this instead:

```
lea esi,[ebp-8]
```

- LEA指令返回间接操作数的偏移地址
- 间接操作数可能使用一个或多个寄存器，因此其偏移值是在运行时计算的。
- OFFSET获取在编译时就已经知道的地址
- OFFSET操作符不能获取堆栈参数的地址

ENTER Instruction

- ENTER instruction creates stack frame for a called procedure
  - pushes EBP on the stack
  - sets EBP to the base of the stack frame
  - reserves space for local variables
  - Example:

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

  - Equivalent to:

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

LEAVE Instruction

Terminates the stack frame for a procedure.

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

Equivalent operations

```
push   ebp
mov    ebp,esp
sub    esp,8      ; 2 local DWORDs
```

```
mov    esp,ebp   ; free local space
pop    ebp
```

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
```

Using LOCAL

Examples:

```
LOCAL flagVals[20]:BYTE       ; array of bytes

LOCAL pArray:PTR WORD         ; pointer to an array

myProc PROC,                  ; procedure
    LOCAL t1:BYTE,            ; local variables
```

LOCAL Example (1 of 2)

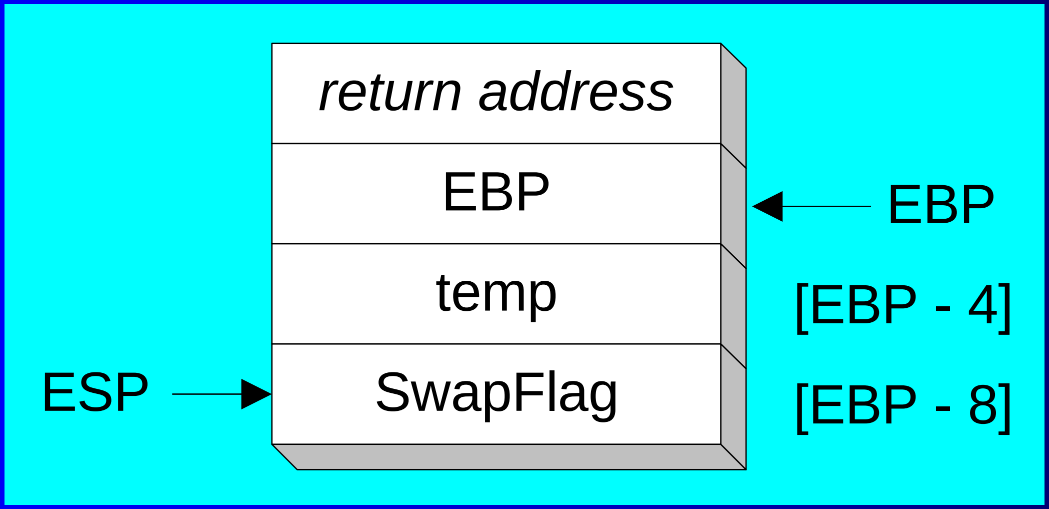
```
BubbleSort PROC
    LOCAL temp:DWORD, SwapFlag:BYTE
    . . .
    ret
BubbleSort ENDP
```

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
```

LOCAL Example (2 of 2)

Diagram of the stack frame for the BubbleSort procedure:

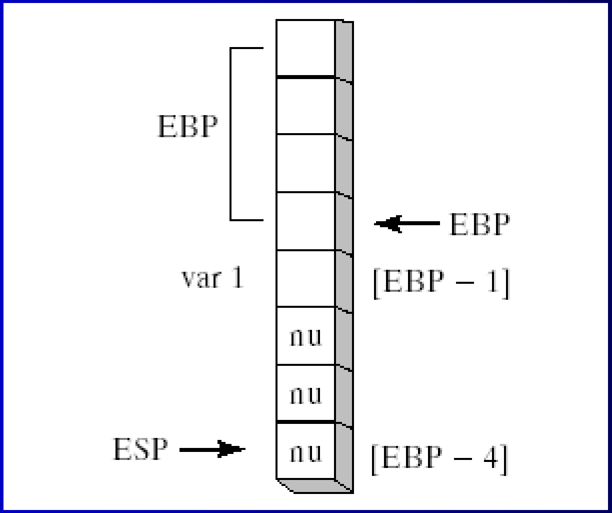


Non-Doubleword Local Variables

- 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

Local Byte Variable

```
Example1 PROC
    LOCAL var1:BYTE
    mov al,var1                  ; [EBP - 1]
    ret
Example1 ENDP
```



WriteStackFrame Procedure

- Displays contents of current stack frame
  - Prototype:

```
WriteStackFrame PROTO,
  numParam:DWORD,      ; number of passed parameters
  numLocalVal: DWORD,  ; number of DWordLocal variables
  numSavedReg: DWORD   ; number 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
```

The Microsoft x64 Calling Convention

- CALL subtracts 8 from RSP
- First four parameters are placed in RCX, RDX, R8, and R9. Additional parameters are pushed on the stack.
- Parameters less than 64 bits long are not zero extended
- Return value in RAX if <= 64 bits
- Caller must allocate at least 32 bytes of shadow space so the subroutine can copy parameter values

The Microsoft x64 Calling Convention

- Caller must align RSP to 16-byte boundary
- Caller must remove all parameters from the stack after the call
- Return value larger than 64 bits must be placed on the runtime stack, with RCX pointing to it
- RBX, RBP, RDI, RSI, R12, R14, R14, and R15 registers are preserved by the subroutine; all others are not.

What's Next

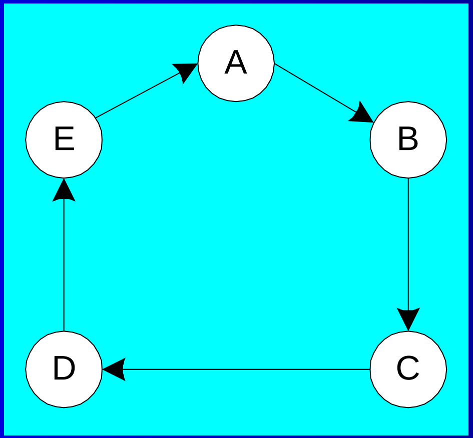
- Stack Frames
- **Recursion**
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

Recursion

- What is Recursion?
- Recursively Calculating a Sum
- Calculating a Factorial

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:



Recursively Calculating a Sum

The CalcSum procedure recursively calculates the sum of an array of integers. Receives: ECX = count. Returns: EAX = sum

```
CalcSum PROC
    cmp ecx,0                     ; check counter value
    jz L2                         ; quit if zero
    add eax,ecx                   ; otherwise, add to sum
    dec ecx                       ; decrement counter
    call CalcSum                  ; recursive call
L2: ret
CalcSum ENDP
```

Stack frame:

Pushed On Stack	ECX	EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

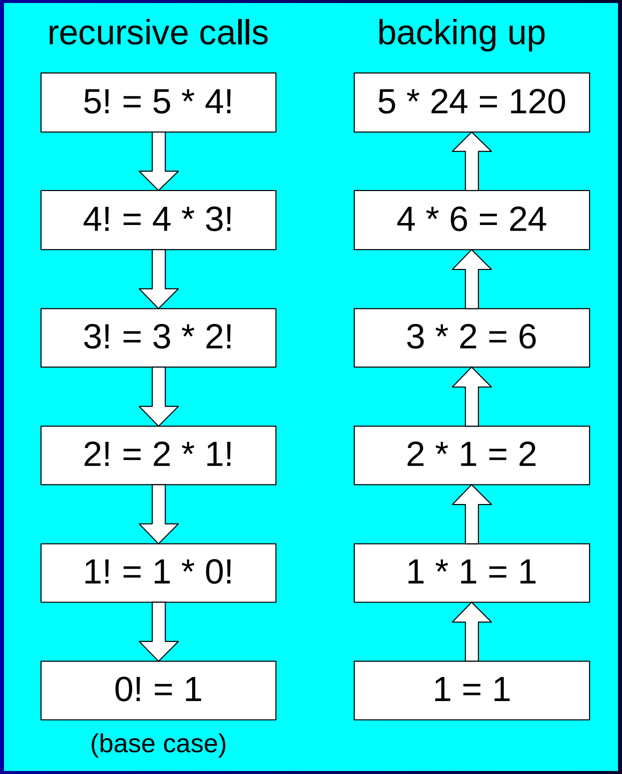
View the complete program

Calculating a Factorial (1 of 3)

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

```
int function 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 a Factorial (2 of 3)

```
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

; Instructions from this point on execute when each
; recursive call returns.

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

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

See the program listing

- main PROC

  push 12; calc 12!

  call Factorial; calculate factorial (eax)

  ReturnMain:

  call WriteDec; display it

  call Crlf

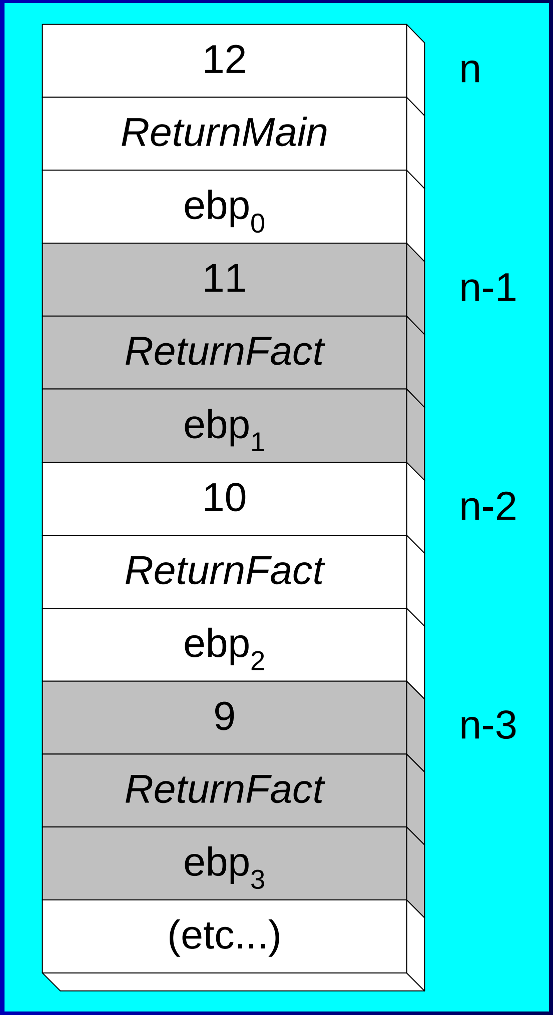
  exit
- main ENDP

Calculating a Factorial (3 of 3)

Suppose we want to calculate 12!

This diagram shows the first few stack frames created by recursive calls to Factorial

Each recursive call uses 12 bytes of stack space.



What's Next

- Stack Frames
- Recursion
- **INVOKE, ADDR, PROC, and PROTO**
- Creating Multimodule Programs
- Java Bytecodes

INVOKE, ADDR, PROC, and PROTO

- INVOKE Directive
- ADDR Operator
- PROC Directive
- PROTO Directive
- Parameter Classifications
- Example: Exchaning Two Integers
- Debugging Tips

Not in 64-bit mode!

INVOKE Directive

- In 32-bit mode, 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]
```

Not in 64-bit mode!

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
```

Not in 64-bit mode!

PROC Directive (1 of 2)

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

  label PROC 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.

PROC Directive (2 of 2)

- Alternate format permits parameter list to be on one or more separate lines:

  label PROC**,** ← comma required

  paramList

- The parameters can be on the same line . . .

  param-1:type-1, param-2:type-2, . . ., param-n:type-n

- Or they can be on separate lines:

  param-1:type-1,

  param-2:type-2,

  . . .,

  param-n:type-n

AddTwo Procedure (1 of 2)

- The AddTwo procedure receives two integers and returns their sum in EAX.

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

    mov eax,val1
    add eax,val2

    ret
AddTwo ENDP
```

PROC Examples (2 of 3)

FillArray receives a pointer to an array of bytes, a single byte fill value that will be copied to each element of the array, and the size of the array.

```
FillArray PROC,
    pArray:PTR BYTE, fillVal:BYTE
    arraySize:DWORD

    mov ecx,arraySize
    mov esi,pArray
    mov al,fillVal
L1: mov [esi],al
    inc esi
    loop L1
    ret
FillArray ENDP
```

PROC Examples (3 of 3)

```
Swap PROC,
    pValX:PTR DWORD,
    pValY:PTR DWORD
    . . .
Swap ENDP
```

```
ReadFile PROC,
    pBuffer:PTR BYTE
    LOCAL fileHandle:DWORD
    . . .
ReadFile ENDP
```

PROTO Directive

- Creates a procedure prototype
- Syntax:
  - *label* PROTO *paramList*
- Parameter list not permitted in 64-bit mode
- 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
```

Parameters are not permitted in 64-bit mode.

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 is a pointer to a variable containing input that will be both used and modified by the procedure.
  - The variable passed by the calling program is modified.

Trouble-Shooting Tips

- Save and restore registers when they are modified by a procedure.
  - Except a register that returns a function result
- When using INVOKE, be careful to pass a pointer to the correct data type.
  - For example, MASM cannot distinguish between a DWORD argument and a PTR BYTE argument.
- Do not pass an immediate value to a procedure that expects a reference parameter.
  - Dereferencing its address will likely cause a general-protection fault.

What's Next

- Stack Frames
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- **Creating Multimodule Programs**
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

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

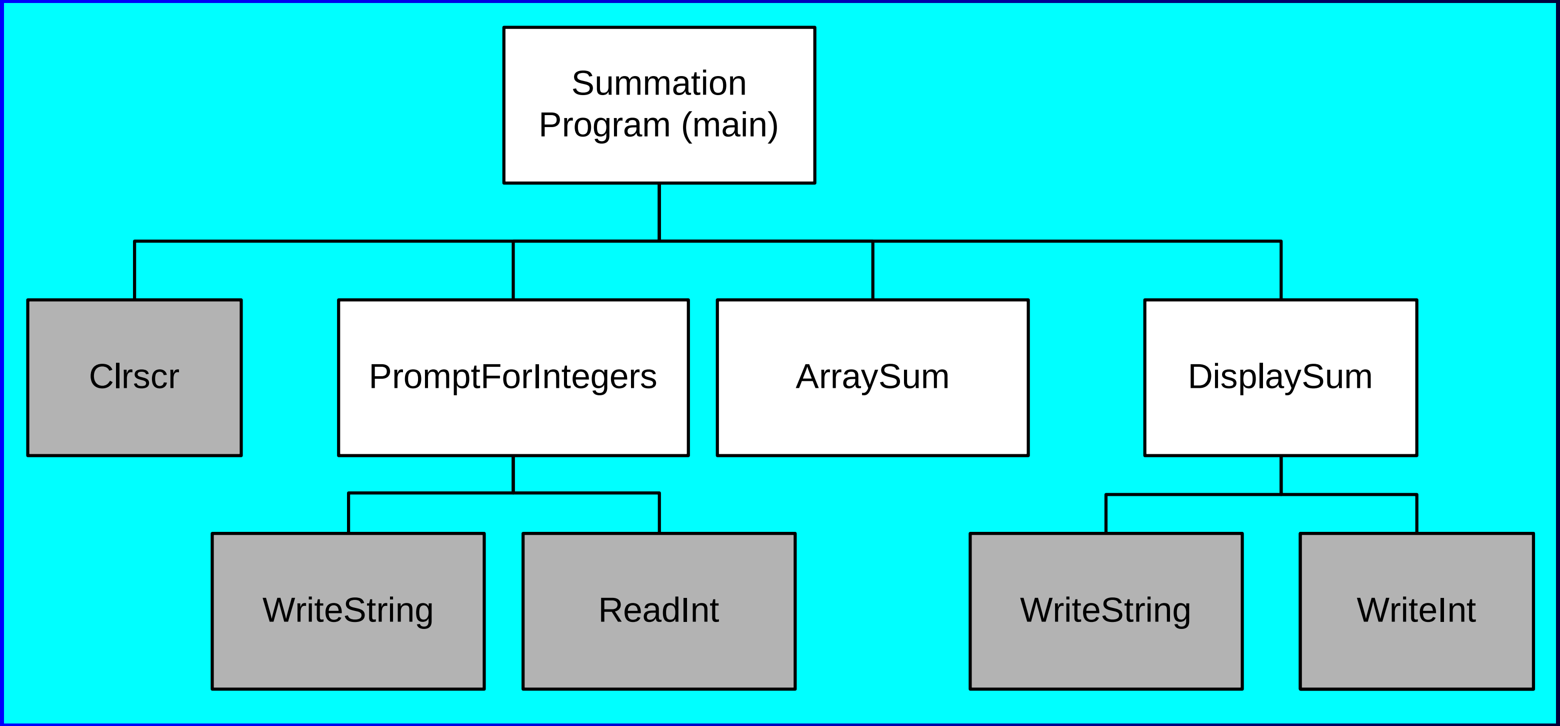
- Large programs are easier to write, maintain, and debug when divided into separate source code modules.
- When changing a line of code, only its enclosing module needs to be assembled again. Linking assembled modules requires little time.
- A module can be a container for logically related code and data (think object-oriented here...)
  - 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

Example: ArraySum Program

- Let's review the ArraySum program from Chapter 5.



Each of the four white rectangles will become a module. This will be a 32-bit application.

Sample Program output

```
Enter a signed integer: -25

Enter a signed integer: 36

Enter a signed integer: 42

The sum of the integers is: +53
```

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
```

Inspect Individual Modules

- Main
- PromptForIntegers
- ArraySum
- DisplaySum

What's Next

- Stack Frames
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- **Java Bytecodes (optional)**

Java Bytecodes

- Stack-oriented instruction format
  - operands are on the stack
  - instructions pop the operands, process, and push result back on stack
- Each operation is atomic
- Might be be translated into native code by a *just in time* compiler

Java Virual Machine (JVM)

- Essential part of the Java Platform
- Executes compiled bytecodes
  - machine language of compiled Java programs

Java Methods

- Each method has its own stack frame
- Areas of the stack frame:
  - local variables
  - operands
  - execution environment

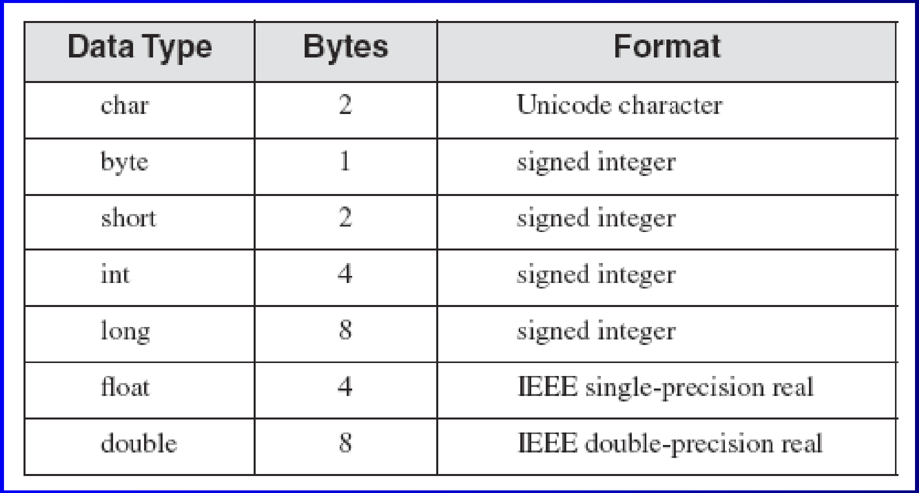
Bytecode Instruction Format

- 1-byte opcode
  - iload, istore, imul, goto, etc.
- zero or more operands

- Disassembling Bytecodes
  - use javap.exe, in the Java Development Kit (JDK)

Primitive Data Types

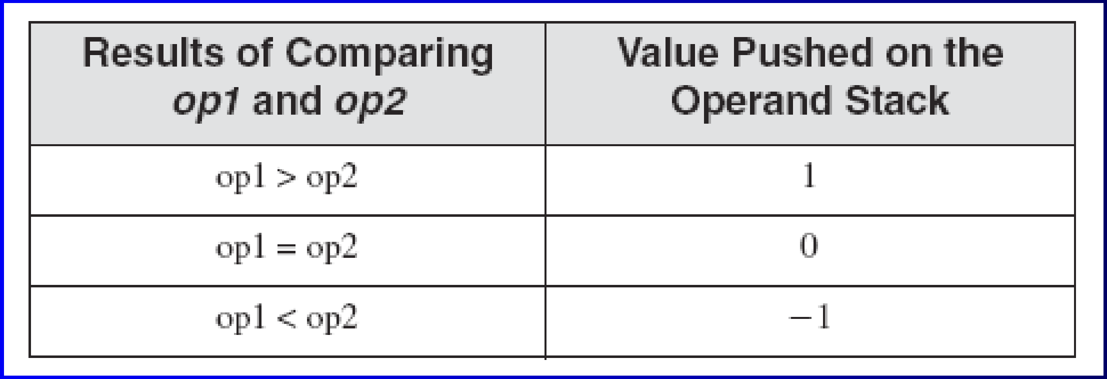
- Signed integers are in twos complement format, stored in big-endian order



Data Type	Bytes	Format
char	2	Unicode character
byte	1	signed integer
short	2	signed integer
int	4	signed integer
long	8	signed integer
float	4	IEEE single-precision real
double	8	IEEE double-precision real

JVM Instruction Set

- Comparison Instructions pop two operands off the stack, compare them, and push the result of the comparison back on the stack
- Examples: fcmp and dcmp



Results of Comparing *op1* and *op2*	Value Pushed on the Operand Stack
op1 > op2	1
op1 = op2	0
op1 < op2	–1

JVM Instruction Set

- Conditional Branching
  - jump to label if st(0) <= 0

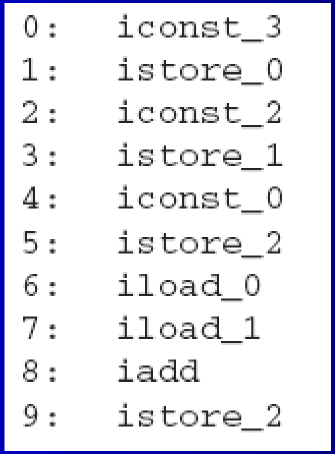
    ```
    ifle label
    ```
- Unconditional Branching
  - call subroutine

    ```
    jsr label
    ```

Java Disassembly Examples

- Adding Two Integers

```
int A = 3;
int B = 2;
int sum = 0;
sum = A + B;
```

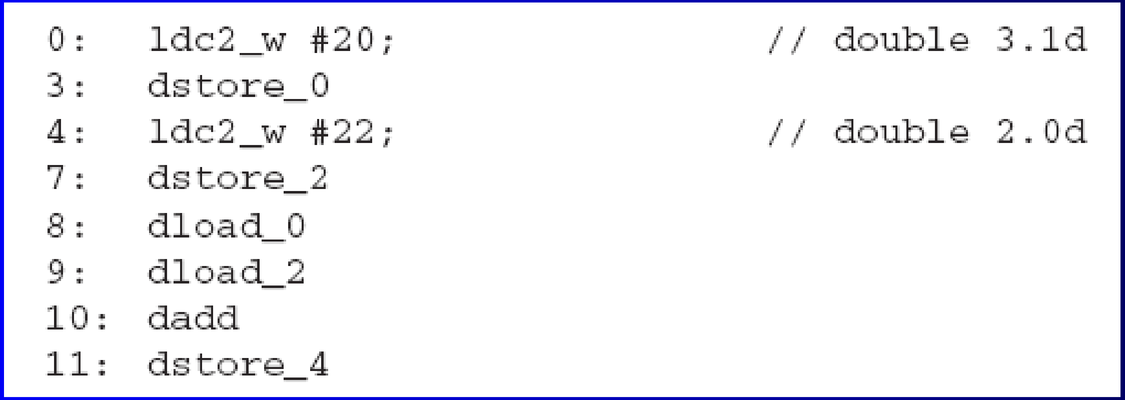


```
0:    iconst_3
1:    istore_0
2:    iconst_2
3:    istore_1
4:    iconst_0
5:    istore_2
6:    iload_0
7:    iload_1
8:    iadd
9:    istore_2
```

Java Disassembly Examples

- Adding Two Doubles

```
double A = 3.1;
double B = 2;
double sum = A + B;
```

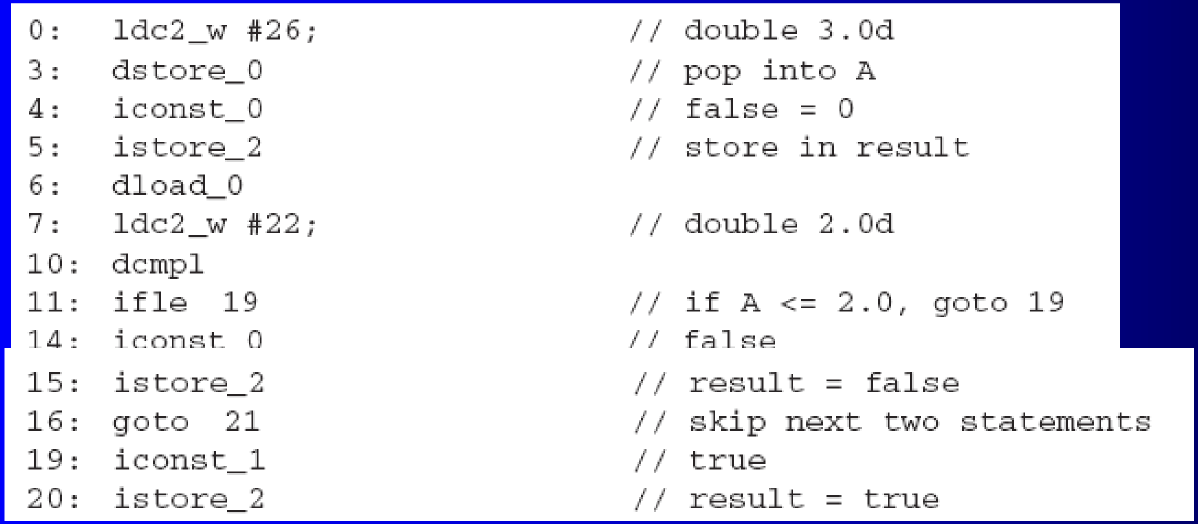


```
0:   ldc2_w #20;                    // double 3.1d
3:   dstore_0
4:   ldc2_w #22;                    // double 2.0d
7:   dstore_2
8:   dload_0
9:   dload_2
10:  dadd
11:  dstore_4
```

Java Disassembly Examples

- Conditional Branch

```
double A = 3.0;
boolean result = false;
if( A > 2.0 )
    result = false;
else
    result = true;
```



```
0:   ldc2_w #26;              // double 3.0d
3:   dstore_0                 // pop into A
4:   iconst_0                 // false = 0
5:   istore_2                 // store in result
6:   dload_0
7:   ldc2_w #22;              // double 2.0d
10:  dcmpl
11:  ifle  19                 // if A <= 2.0, goto 19
14:  iconst_0                 // false
15:  istore_2                 // result = false
16:  goto  21                 // skip next two statements
19:  iconst_1                 // true
20:  istore_2                 // result = true
```

Summary

- Stack parameters
  - more convenient than register parameters
  - passed by value or reference
  - ENTER and LEAVE instructions
- Local variables
  - created on the stack below stack pointer
  - LOCAL directive
- Recursive procedure calls itself
- Calling conventions (C, stdcall)
- MASM procedure-related directives
  - INVOKE, PROC, PROTO
- Java Bytecodes – another approch to programming