CS 271
Computer Architecture & Assembly Language

Lecture 13

Parameter Passing using Stack

Array

Random Number

2/15/22, Tuesday

Oregon State University

Odds and Ends

- Program 5 posted

Lecture Topics:

- Passing Parameters on the System Stack

- Introduction to Arrays
- Arrays as Reference Parameters
- Display an Array Sequentially
- “Random” Numbers

Passing Parameters on the System Stack

Recall: RET Instruction

ret ⇔ pop EIP

- Pops stack into the instruction pointer (EIP)
- Syntax:
  - **RET**
  - **RET n**
- Optional operand *n* causes *n* to be added to the stack pointer after EIP is assigned a value
  - Equivalent to popping the return address **and n additional bytes** off the stack

Recall: Stack Frame

- Also known as an activation record
- Area of the stack used for a procedure's return address, passed parameters, saved registers, and local variables

- Created by the following steps:
  - Calling program pushes arguments onto the stack and calls the procedure
  - The called procedure pushes EBP onto the stack, and sets EBP to ESP

base pointer

Recall: Addressing Modes

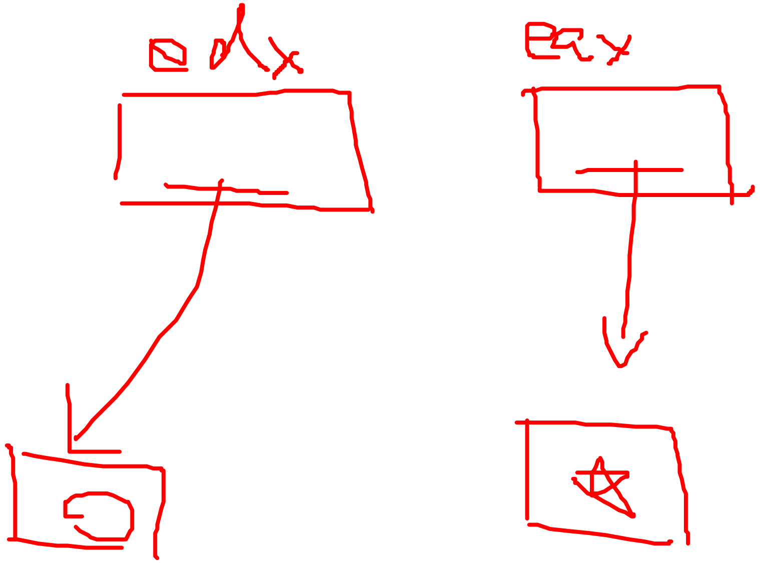
• Immediate	Constants, literal, absolute address
• Direct	Contents of referenced memory address
• Register	Contents of register
• Register indirect	Access memory through address in a register
• Indexed	Array name using element “distance” in register
• Base-indexed	Start address in one register; offset in another, add and access memory
• Stack	Memory area specified and maintained as a stack; Stack pointer in ESP register
• Offset	Memory address; may be computed

Recall: Register Indirect Mode

- [reg] means "contents of memory at the address in *reg*"
- It is OK to add a constant (named or literal)
  - Example: mov [edx+12], eax

- We have used register indirect with **esp** to reference the value at the top of the system stack

- Note: register indirect is a **memory reference**
  - There are no memory-memory instruction
  - E.g., mov [edx], [eax] is WRONG!



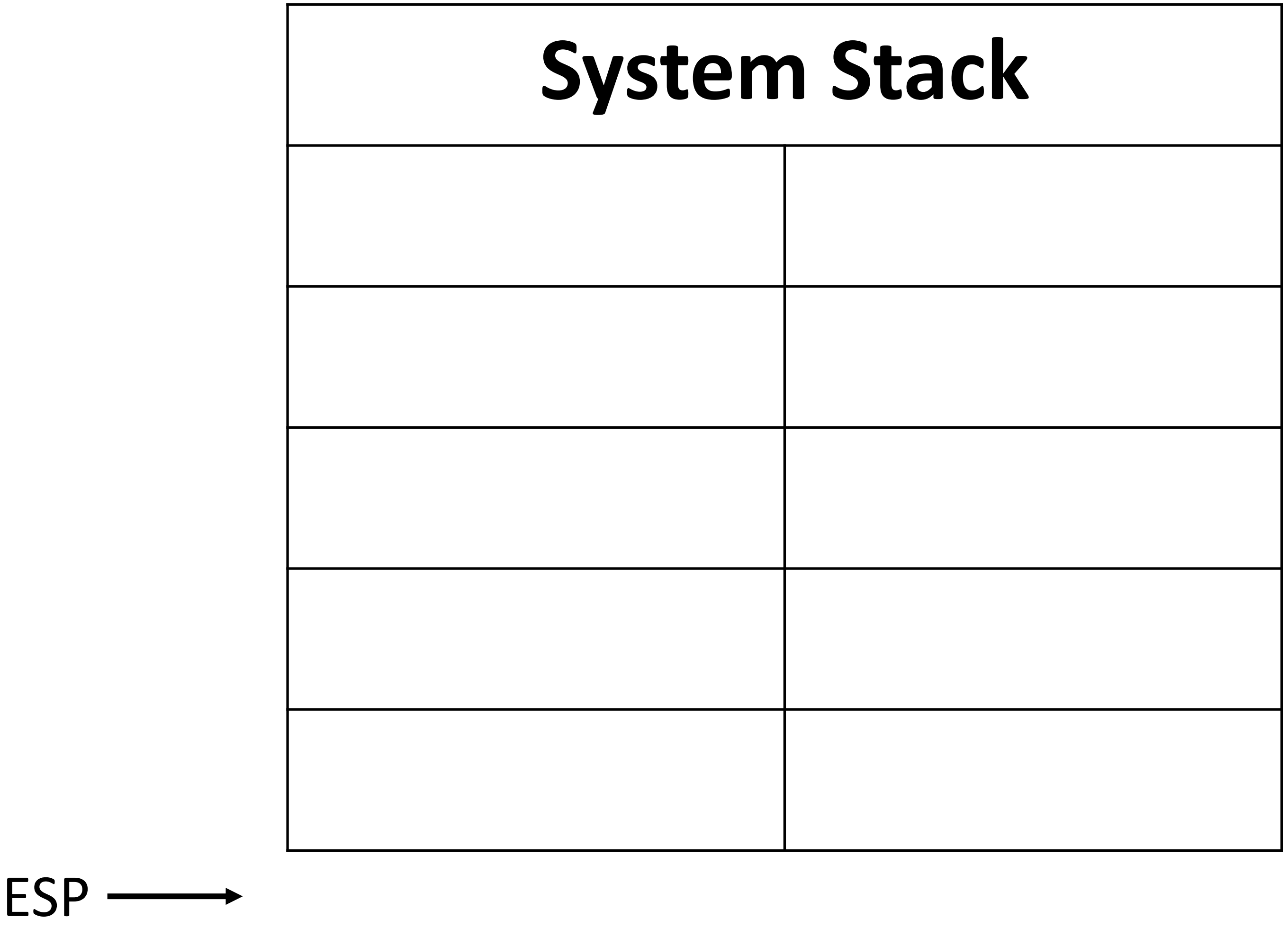
Recall: Explicit Access to Stack Parameters

- 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 is (should be) set to the base address of the stack frame
- EBP should not change value during the procedure
- EBP must be restored to its original value when the procedure returns
- Remember that the return address is pushed onto the stack **after** the parameters are pushed
- Programmer is responsible for managing the stack.

Stack Frame Example

```
.data
x  DWORD    175
y  DWORD    37
Z  DWORD    ?

.code
main   PROC
   push      x
   push      y
   push      OFFSET z
   call      SumTwo
```



Stack Frame Example

```
.data
x  DWORD    175
y  DWORD    37
Z  DWORD    ?

.code
main   PROC
   push      x
   push      y
   push      OFFSET z
   call      SumTwo
```

	System Stack	
ESP →	[ESP]	175

Stack Frame Example

```
.data
x DWORD    175
y DWORD    37
Z DWORD    ?

.code
main   PROC
   push      x
   push      y
   push      OFFSET z
   call      SumTwo
```

ESP →

System Stack	
[ESP]	37
[ESP + 4]	175

Stack Frame Example

Note: @ means "address of"

```
.data
x  DWORD     175
y  DWORD     37
z  DWORD     ?

.code
main   PROC
   push      x
   push      y
   push      OFFSET z
   call      SumTwo
```

	System Stack	
ESP →	[ESP]	@ z
	[ESP + 4]	37
	[ESP + 8]	175

Stack Frame Example

Note: @ means "address of"

```
.data
x  DWORD     175
y  DWORD     37
z  DWORD     ?
SumTwo ( x, y, &z ) ;
.code
main   PROC
   push      x
   push      y
   push      OFFSET z
➡  call      SumTwo
  (addr._)
```

ESP → [ESP]

System Stack	
[ESP]	return @
[ESP + 4]	@ z
[ESP + 8]	37
[ESP + 12]	175

Stack Frame Example

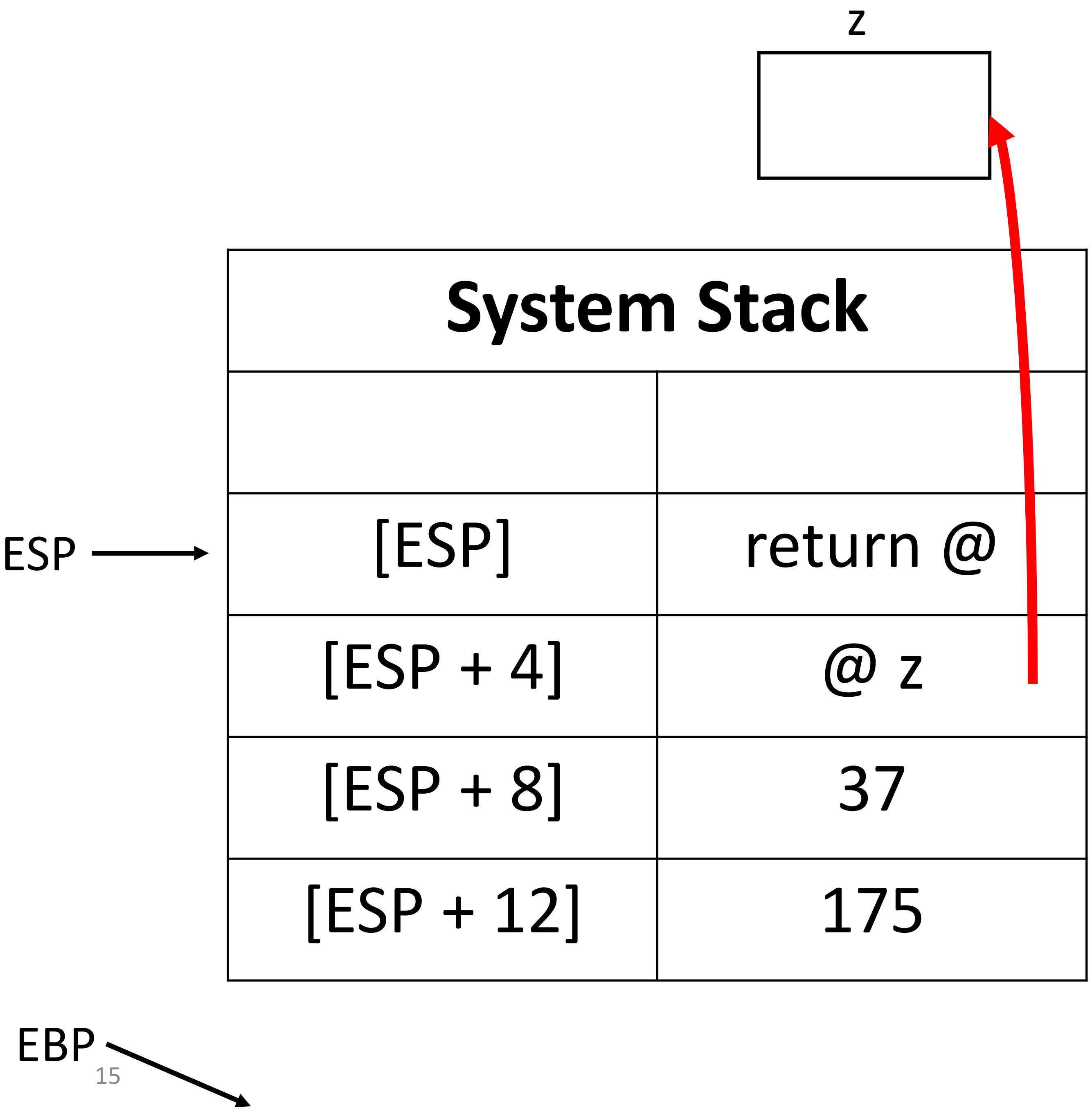
```
SumTwo    PROC
    push  ebp
    mov   ebp, esp
    mov   eax, [ebp+16]
    ;175 in eax

    add   eax, [ebp+12]
    ;175+37 = 212 in eax

    mov   ebx, [ebp+8]
    ;@z in ebx

    mov   [ebx], eax
    ;store 212 in z

    pop   ebp
    ret   12
SumTwo    ENDP
```



System Stack	
[ESP]	return @
[ESP + 4]	@ z
[ESP + 8]	37
[ESP + 12]	175

Stack Frame Example

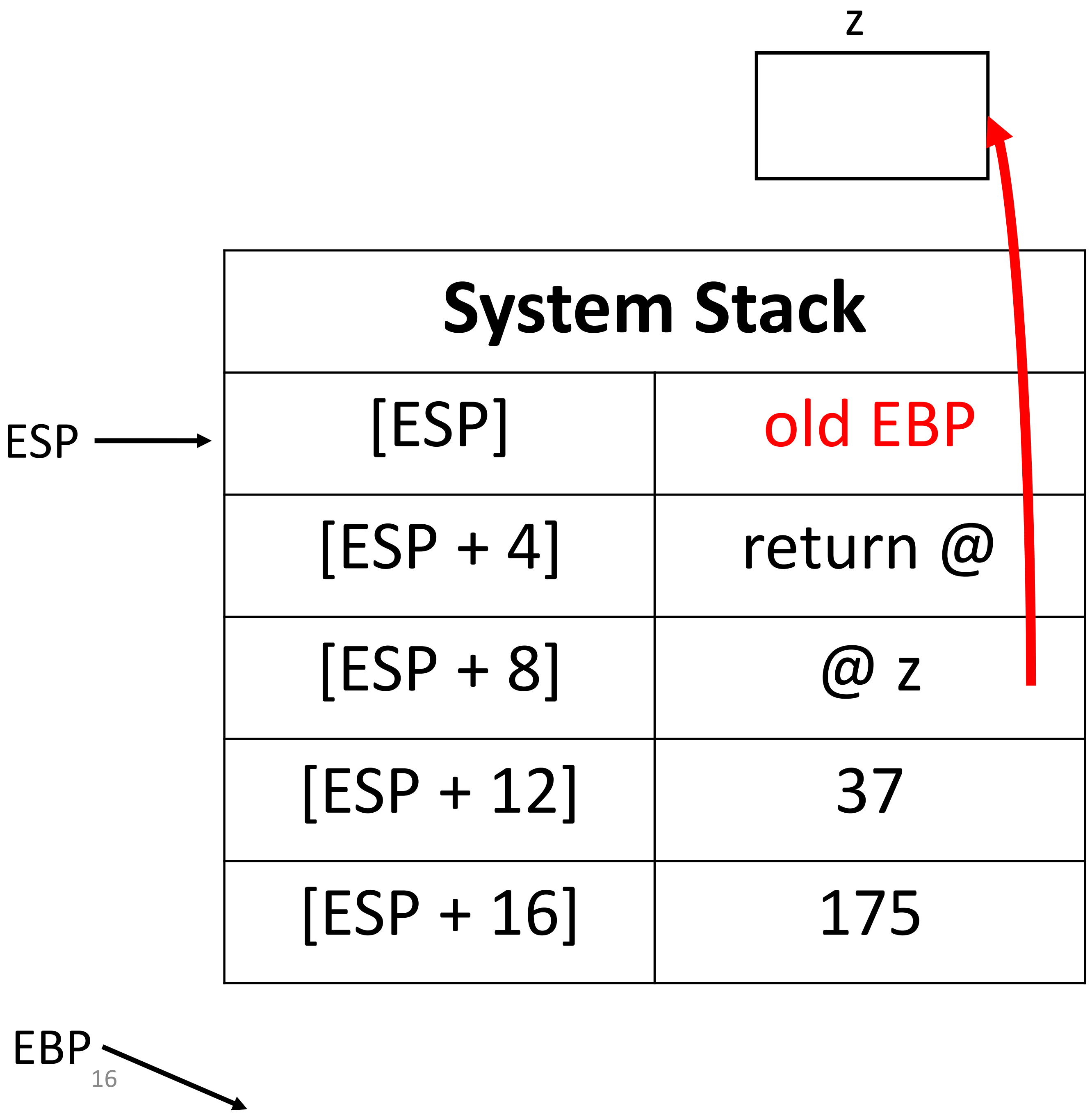
```
SumTwo    PROC
    push  ebp
    mov   ebp, esp
    mov   eax, [ebp+16]
    ;175 in eax

    add   eax, [ebp+12]
    ;175+37 = 212 in eax

    mov   ebx, [ebp+8]
    ;@z in ebx

    mov   [ebx], eax
    ;store 212 in z

    pop   ebp
    ret   12
SumTwo    ENDP
```



System Stack	
[ESP]	old EBP
[ESP + 4]	return @
[ESP + 8]	@ z
[ESP + 12]	37
[ESP + 16]	175

Stack Frame Example

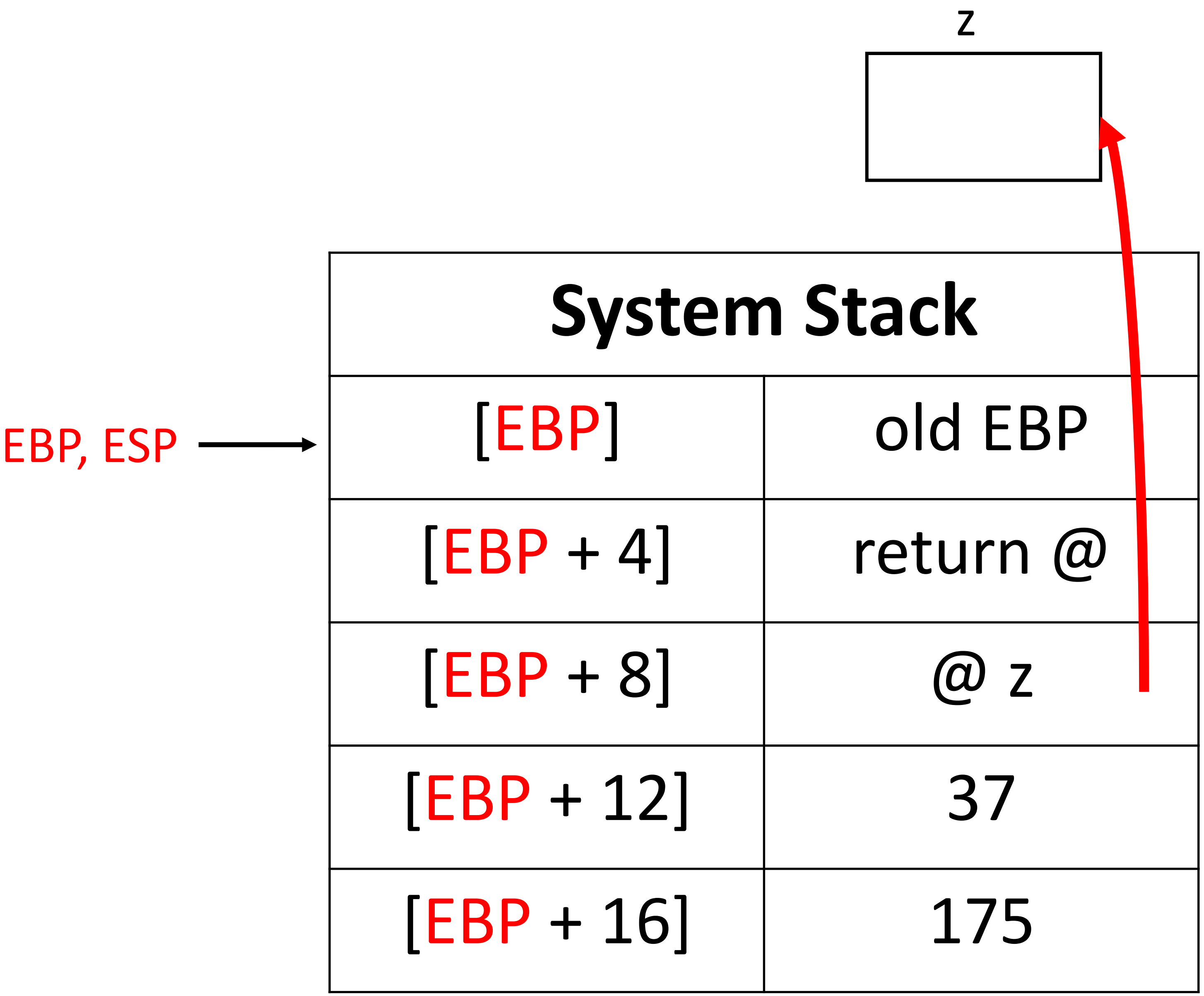
```
SumTwo    PROC
    push  ebp
    mov   ebp, esp
    mov   eax, [ebp+16]
    ;175 in eax

    add   eax, [ebp+12]
    ;175+37 = 212 in eax

    mov   ebx, [ebp+8]
    ;@z in ebx

    mov   [ebx], eax
    ;store 212 in z

    pop   ebp
    ret   12
SumTwo    ENDP
```



System Stack	
[EBP]	old EBP
[EBP + 4]	return @
[EBP + 8]	@ z
[EBP + 12]	37
[EBP + 16]	175

Stack Frame Example

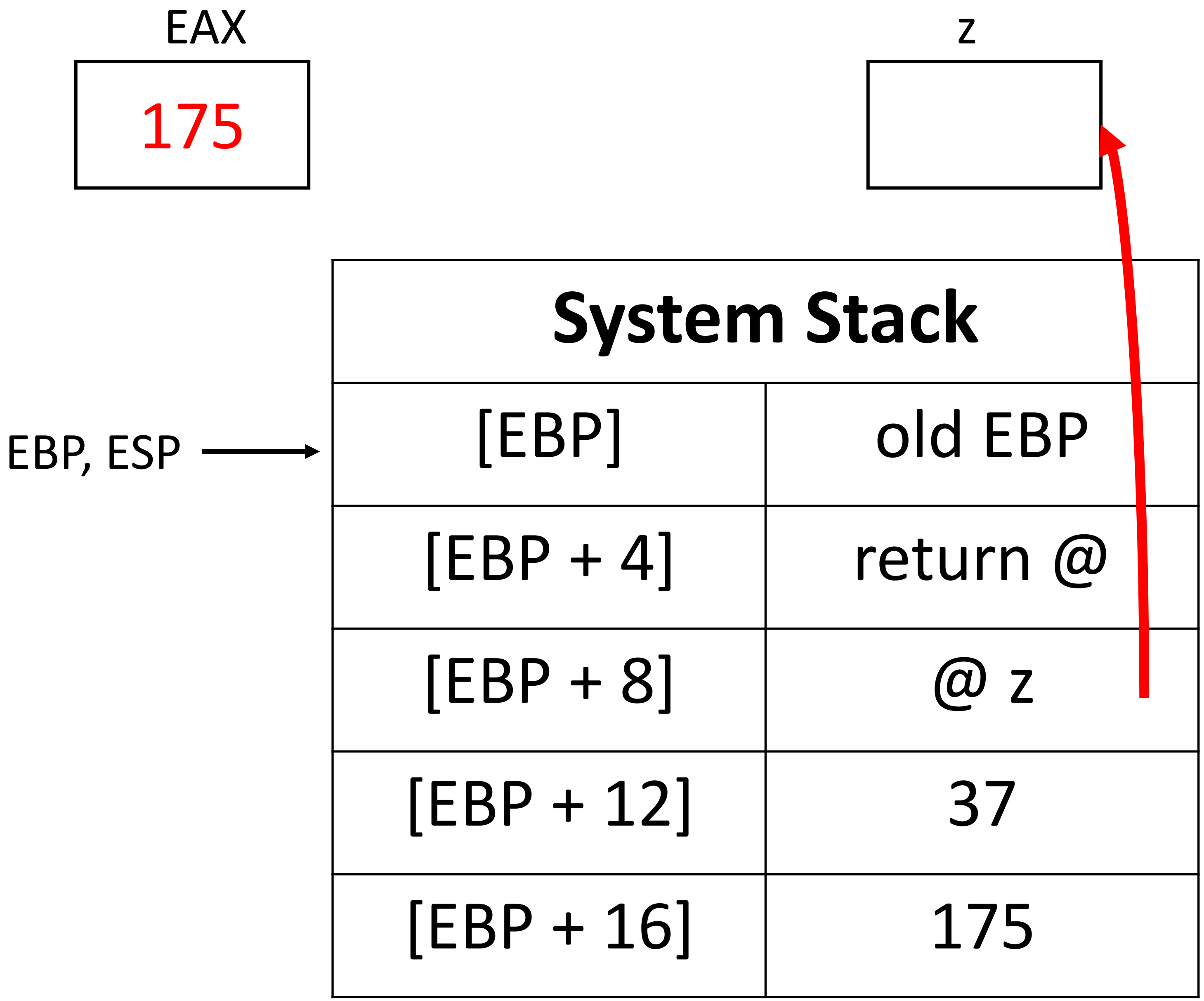
```
SumTwo   PROC
   push  ebp
   mov   ebp, esp
   mov   eax, [ebp+16]
   ;175 in eax

   add   eax, [ebp+12]
   ;175+37 = 212 in eax

   mov   ebx, [ebp+8]
   ;@z in ebx

   mov   [ebx], eax
   ;store 212 in z

   pop   ebp
   ret   12
SumTwo   ENDP
```



System Stack	
[EBP]	old EBP
[EBP + 4]	return @
[EBP + 8]	@ z
[EBP + 12]	37
[EBP + 16]	175

Stack Frame Example

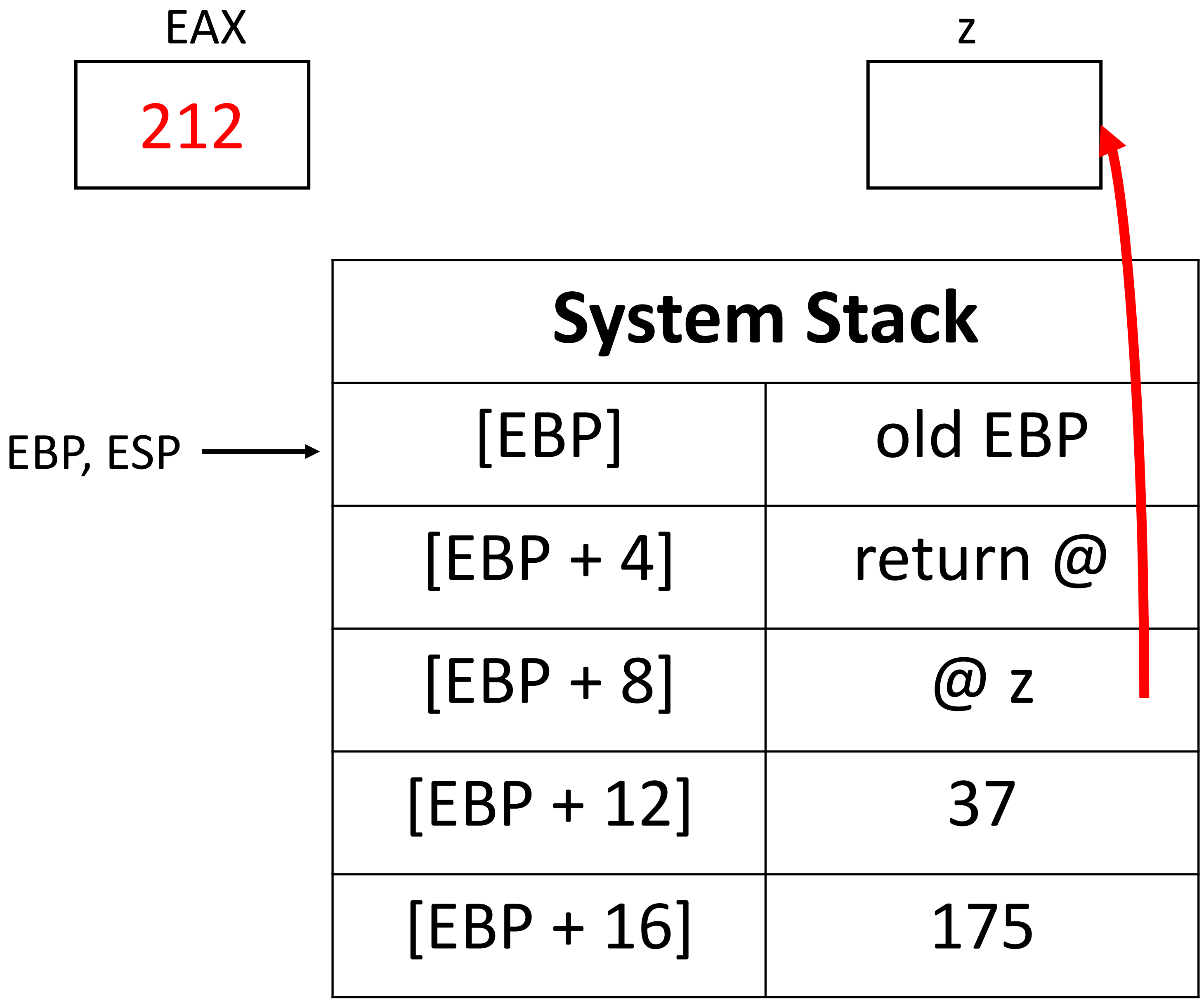
```
SumTwo    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, [ebp+16]
    ;175 in eax

    add    eax, [ebp+12]
    ;175+37 = 212 in eax

    mov    ebx, [ebp+8]
    ;@z in ebx

    mov    [ebx], eax
    ;store 212 in z

    pop    ebp
    ret    12
SumTwo    ENDP
```



System Stack	
[EBP]	old EBP
[EBP + 4]	return @
[EBP + 8]	@ z
[EBP + 12]	37
[EBP + 16]	175

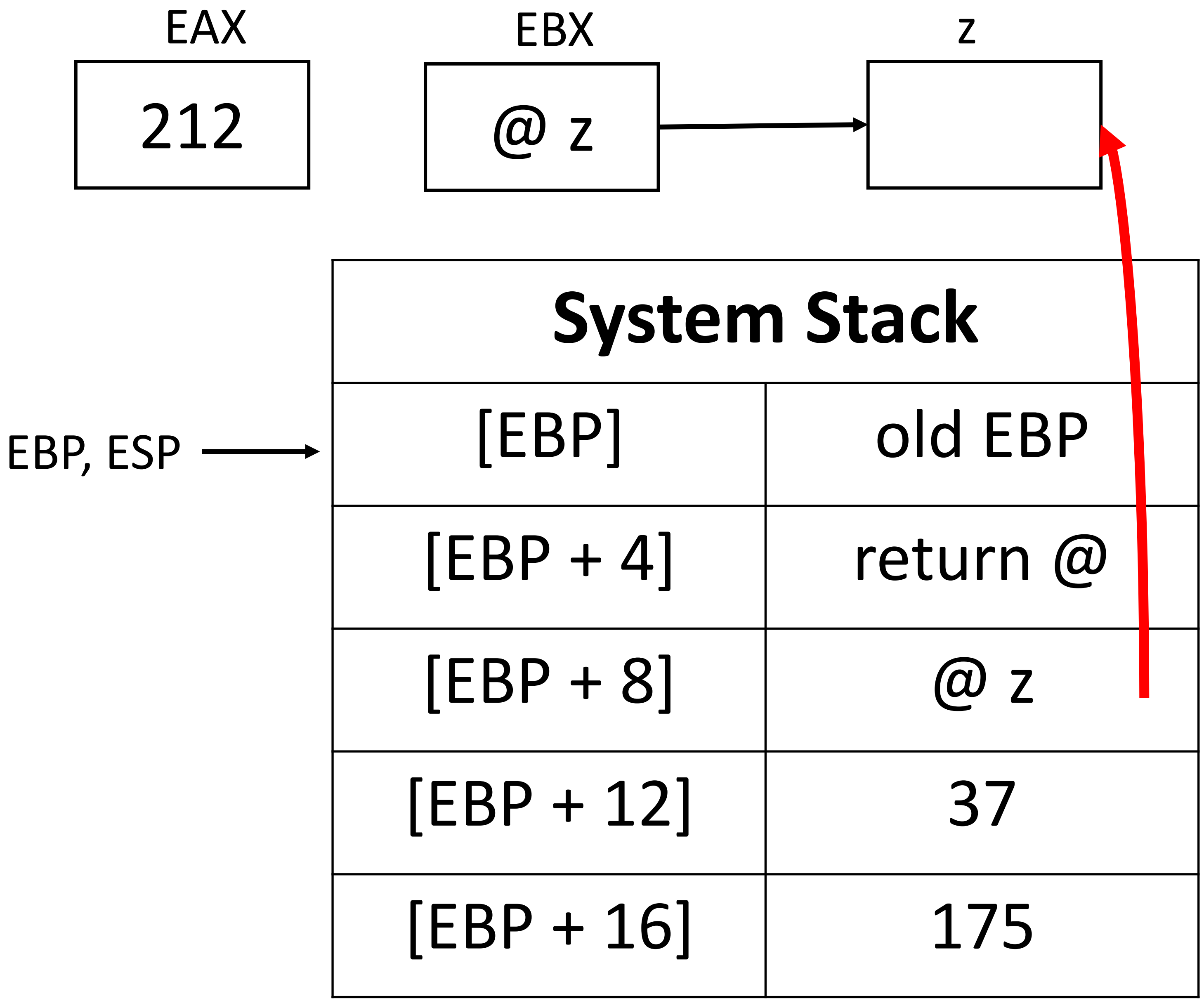
Stack Frame Example

```
SumTwo    PROC
    push  ebp
    mov   ebp, esp
    mov   eax, [ebp+16]
    ;175 in eax

    add   eax, [ebp+12]
    ;175+37 = 212 in eax

    mov   ebx, [ebp+8]
    ;@z in ebx
        *ebx = eax
    mov   [ebx], eax
    ;store 212 in z

    pop   ebp
    ret   12
SumTwo    ENDP
```



System Stack	
[EBP]	old EBP
[EBP + 4]	return @
[EBP + 8]	@ z
[EBP + 12]	37
[EBP + 16]	175

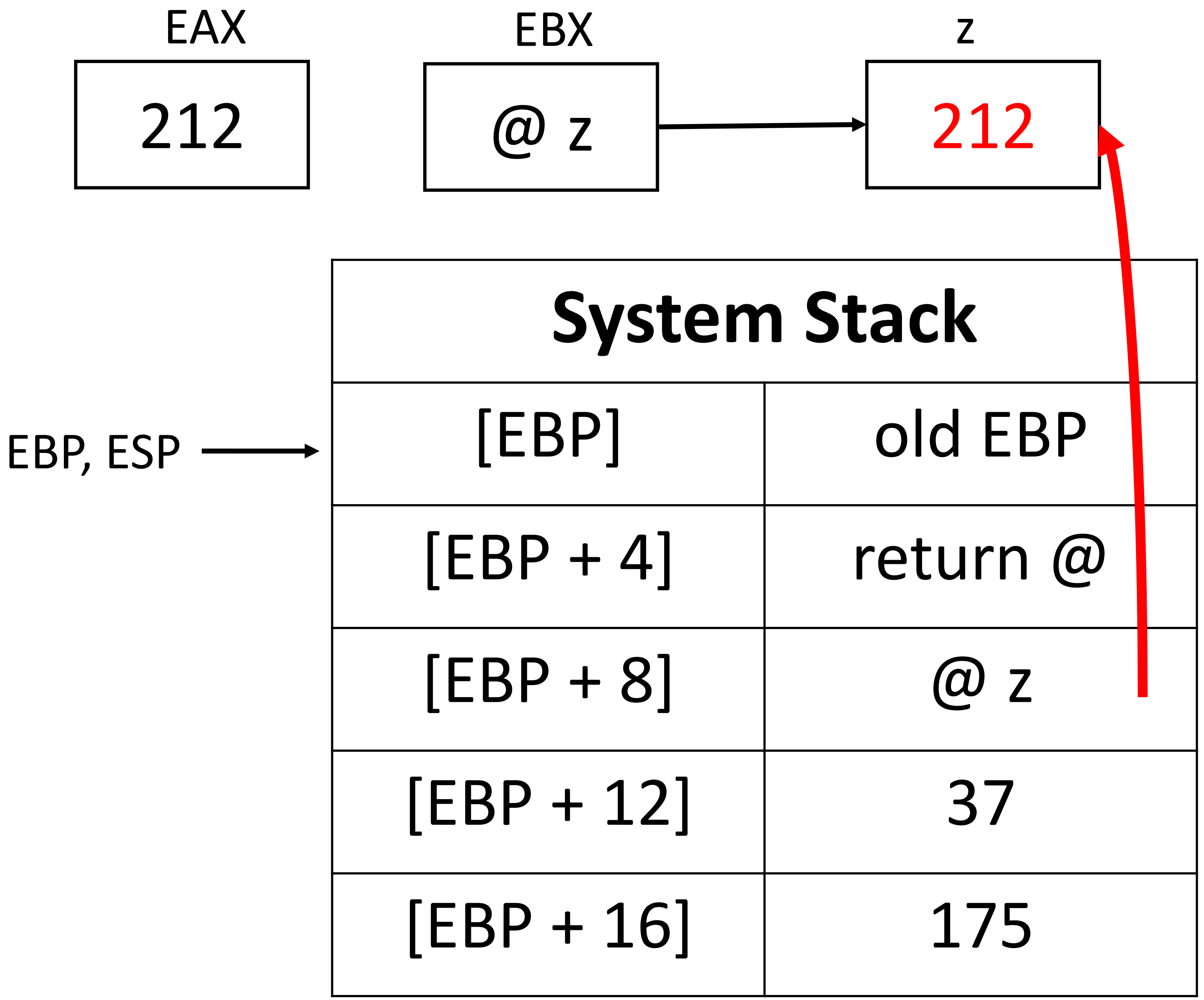
Stack Frame Example

```
SumTwo    PROC
    push  ebp
    mov   ebp, esp
    mov   eax, [ebp+16]
    ;175 in eax

    add   eax, [ebp+12]
    ;175+37 = 212 in eax

    mov   ebx, [ebp+8]
    ;@z in ebx
          ~~[ebp+8]~~
    mov   [ebx], eax
    ;store 212 in z

    pop   ebp
    ret   12
SumTwo    ENDP
```



System Stack	
[EBP]	old EBP
[EBP + 4]	return @
[EBP + 8]	@ z
[EBP + 12]	37
[EBP + 16]	175

Stack Frame Example

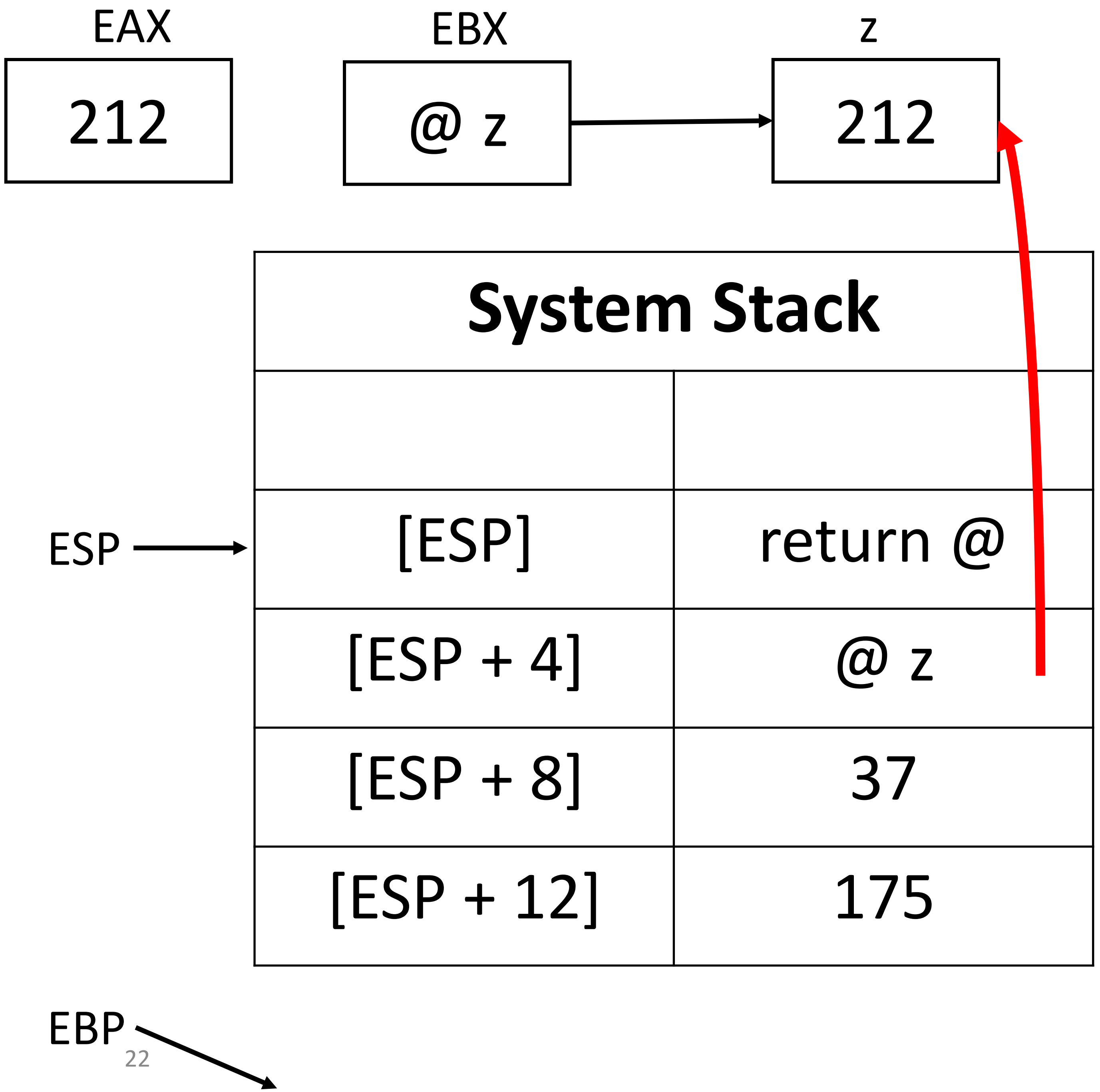
```
SumTwo    PROC
    push  ebp
    mov   ebp, esp
    mov   eax, [ebp+16]
    ;175 in eax

    add   eax, [ebp+12]
    ;175+37 = 212 in eax

    mov   ebx, [ebp+8]
    ;@z in ebx

    mov   [ebx], eax
    ;store 212 in z

    pop   ebp
    ret   12
SumTwo    ENDP
```



System Stack	
[ESP]	return @
[ESP + 4]	@ z
[ESP + 8]	37
[ESP + 12]	175

Stack Frame Example

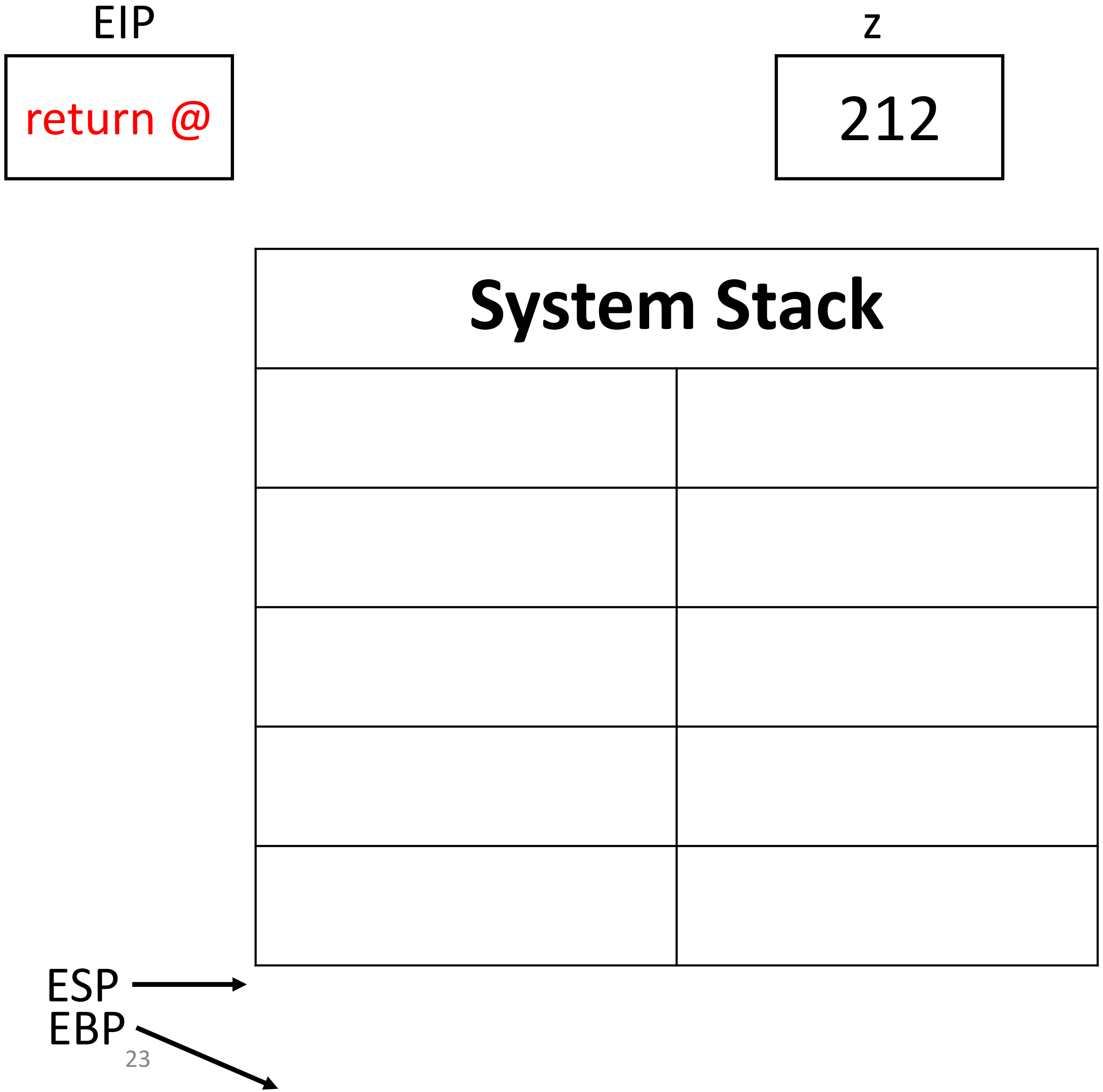
```
SumTwo   PROC
    push  ebp
    mov   ebp, esp
    mov   eax, [ebp+16]
    ;175 in eax

    add   eax, [ebp+12]
    ;175+37 = 212 in eax

    mov   ebx, [ebp+8]
    ;@z in ebx

    mov   [ebx], eax
    ;store 212 in z

    pop   ebp
    ret   12
SumTwo   ENDP
```



- Why don’t we just use ESP instead of EBP?
    - Pushes and pops inside the procedure might cause us to lose the base of the stack frame.

Trouble-Avoidance Tips

- Save and restore registers when they are modified by a procedure.
  - Exception: a register that returns a function result

- Do not pass an immediate value or variable contents to a procedure that expects a reference pointer.
  - Dereferencing it as an address will likely cause a general-protection fault.

Demo

Lecture Topics:

- Passing Parameters on the System Stack

- Introduction to Arrays
- Arrays as Reference Parameters
- Display an Array Sequentially
- “Random” Numbers

Introduction to Arrays

Array in MASM

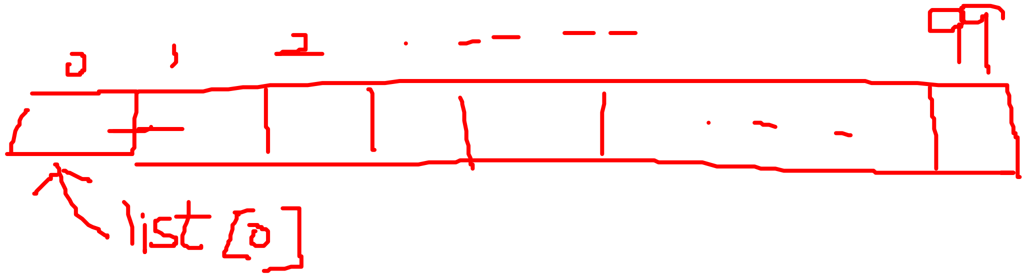
- Declaration (in data segment)

```
MAX_SIZE = 100
.data
list   DWORD     MAX_SIZE      DUP(?)
name   type   ?                initial
```

- Defines an uninitialized array named *list* with space for 100 32-bit integers

- Array elements are in contiguous memory

Array in MASM



- Declaration

```
MAX_SIZE = 100
.data
list    DWORD     MAX_SIZE       DUP(?)
count   DWORD        0
```

list[100] ?

0

- What happen (in HLL) if we reference list[100]?
  - Compile-time error
- What happens in MASM if we go beyond the end of the array?
  - Not easy to predict

Array Address Calculations

- Array declaration defines a name for the first element only
  - HLLs reference it as **list[0]** → *list

    list[k] → *(list + k)
- All other elements are accessed by calculating the actual address
- General formula for array address calculation:
  - Address of list[k] = list + (k * sizeof element)
- Example:

  12 4B

  - Address of 4th element (list[3]) is: address of *list* + (3 * sizeof DWORD)

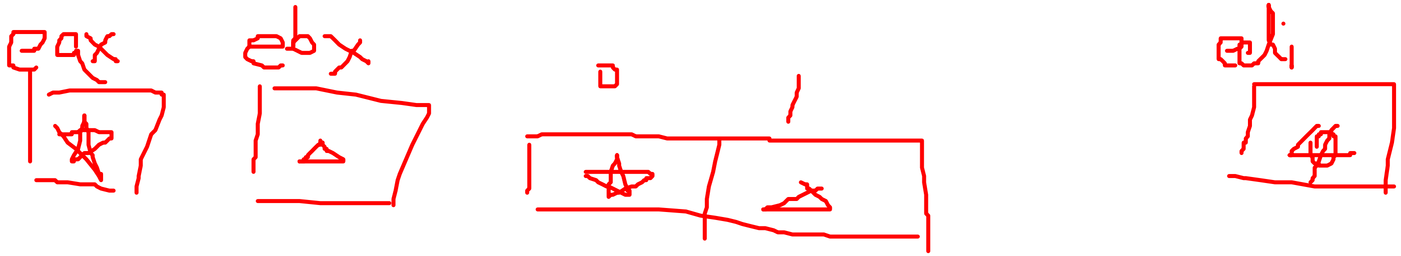
Addressing Modes

• Immediate	Constants, literal, absolute address
• Direct	Contents of referenced memory address
• Register	Contents of register
• Register indirect	Access memory through address in a register
• Indexed	Array name using element “distance” in register
• Base-indexed	Start address in one register; offset in another, add and access memory
• Stack	Memory area specified and maintained as a stack; Stack pointer in ESP register
• Offset	Memory address; may be computed

Array References in MASM

- Several methods for accessing specific array elements
  - Indexed
  - Register indirect
  - Base-indexed

Indexed Addressing



- Array name, with "distance" to element in a register
  - Used for global array references (not used in Program #5)
- Examples:

```
mov     edi,0               ;high-level notation
mov     list[edi],eax           ; is list[0]
add     edi,4                   ;* see note below
mov     list[edi],ebx           ;list[1]
```

index

- This means "add the value in [] to address of list"
- *Note: add 4 because these array elements are DWORD
  - If BYTE, add 1
  - If WORD, add 2
  - If QWORD, add 8
  - Etc.

Register Indirect Addressing

- Actual address of array element in register
  - Used for referencing array elements in procedures
- Examples:
  - In calling procedure…

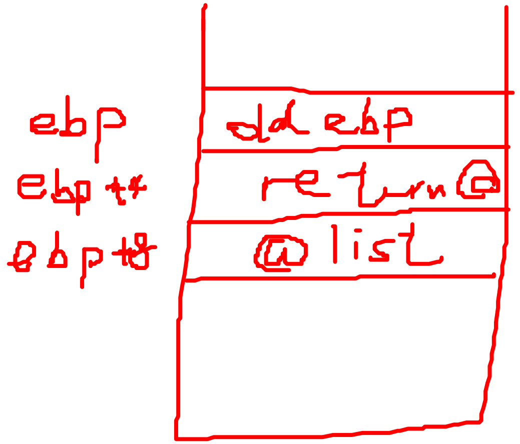
```
push                  OFFSET list
```

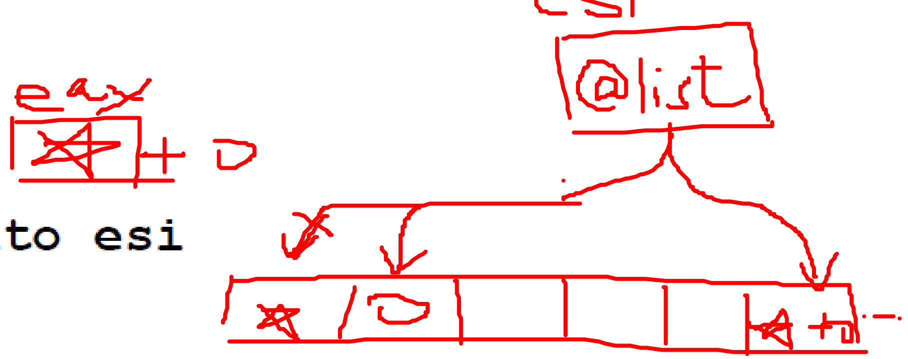
call

  - In called procedure… (example only)

push ebp
mov

```
... ;set up stack frame
mov esi,[ebp+8]          ; get address of list into esi
mov eax,[esi]            ; get list[0] into eax
add esi,4
add eax,[esi]            ; add list[1] to eax
add esi,16
mov [esi],eax            ; send result to list[5]
```





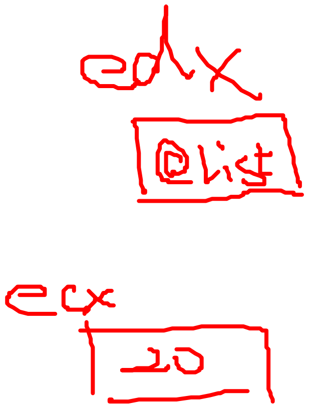
list[5] = list[0] + list[1]

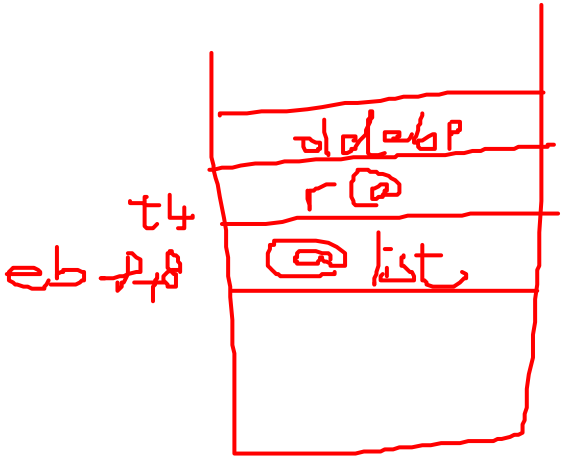
Base-indexed Addressing

- Starting address in one register, offset in another; add and access memory
  - Used for referencing array elements in procedures
- Examples:
  - In calling procedure …

```
    push            OFFSET list
```

  - In called procedure … (example only)





```
... ;set up stack frame
mov edx,[ebp+8]           ; get address of list into edx
mov ecx,20
mov eax,[edx+ecx] ; get list[5] into eax
mov ebx,4
add eax,[edx+ebx] ; add list[1] to eax
mov [edx+ecx],eax ; send result to list[5]
```

list[5] += list[0]

Passing Arrays by Reference

- Never pass an array by value!!!
- Suppose that an *ArrayFill* procedure fills an array with 32-bit integers
- The calling program passed the address of the array, along with *count* of the number of array elements:

```
COUNT = 100
.data
list DWORD COUNT DUP(?)
.code
   ...
   push OFFSET list
   push COUNT
   call ArrayFill
```

Passing Arrays by Reference

- *ArrayFill* can refence an array without knowing the array's name:

```
ArrayFill PROC
   push ebp
   mov  ebp,esp
   mov  edi,[ebp+12] ;@list in edi
   mov  ecx,[ebp+8]  ;value of count in ecx
   ; … etc.
```

- **`edi`** points to the beginning of the array, so it's easy to use a loop to access each array element.

- Style note: We use **`edi`** because the array is the "destination"

Passing Arrays by Reference

- This *ArrayFill* uses register indirect addressing:

```
ArrayFill   PROC
   push ebp
   mov  ebp,esp
   mov  edi,[ebp+12]    ;@list in edi
   mov  ecx,[ebp+8]     ;value of count in ecx
more:
   ; .
   ; Code to generate a random number in eax
   ;   goes here.
   mov   [edi],eax
   add   edi,4
   loop  more

   pop   ebp
   ret   8
ArrayFill   ENDP
```

Passing Arrays by Reference

- This *ArrayFill* uses base-indexed addressing, saves registers:

```
ArrayFill   PROC
   pushad                    ;save all registers
   mov  ebp,esp
   mov  edx,[ebp+40]     ;@list in edx
   mov  ebx,0                ;"index" in ebx
   mov  ecx,[ebp+36]     ;value of count in ecx
more:
   ;  .
   ; Code to generate a random number in eax
   ;   goes here.
   mov    [edx+ebx],eax
   add    ebx,4
   loop   more

   popad                     ;restore all registers
   ret    8
ArrayFill   ENDP
```

Lecture Topics:

- Passing Parameters on the System Stack

- Introduction to Arrays
- Arrays as Reference Parameters
- Display an Array Sequentially
- “Random” Numbers

Setup in Calling Procedure

```
.data
list        DWORD       100 DUP(?)
count       DWORD       0

.code
;...
            ;code to initialize list and count
;...
            ;set up parameters and call display
            push  OFFSET list       ;@list
            push  count       ;number of elements
            call  display
;...
```

Display: version 0.1 (register indirect)

```
display     PROC
        push ebp
        mov  ebp,esp
        mov   esi,[ebp+12]      ;@list
        mov   ecx,[ebp+8]       ;ecx is loop control
more:
        mov   eax,[esi]   ;get current element
        call  WriteDec
        call  Crlf
        add   esi,4       ;next element
        loop  more
endMore:
        pop   ebp
        ret   8
display     ENDP
```

Display: version 0.2 (base-indexed)

```
display     PROC
      push  ebp
      mov   ebp,esp
      mov   esi,[ebp+12]      ;@list
      mov   ecx,[ebp+8] ;ecx is loop control
      mov   edx,0       ;edx is element “pointer”
more:
      mov   eax,[esi+edx]      ;get current element
      call  WriteDec
      call  Crlf
      add   edx,4        ;next element
      loop  more
endMore:
      pop   ebp
      ret   8
display     ENDP
```

Random Numbers

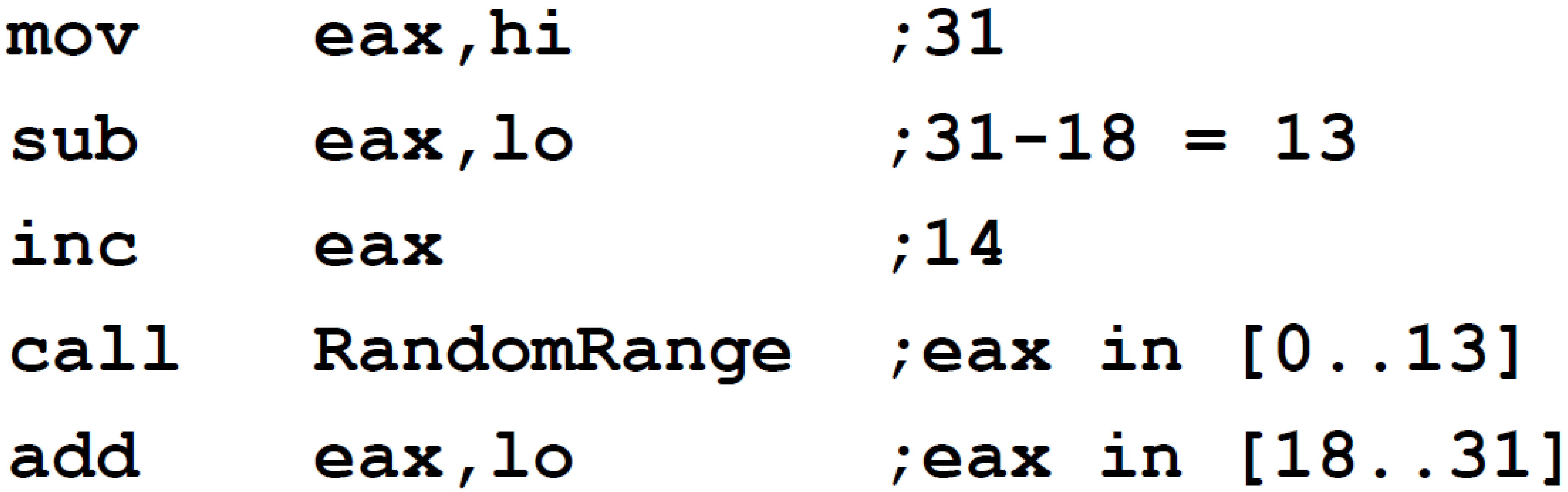
- Irving library has random integer generator
  - “pseudo-random” numbers
- *Randomize* procedure
  - Initialize sequence based on system clock (random seed)
  - Call once at the beginning of the program
  - Without *Randomize*, program gets the same sequence every time it is executed

Limiting Random Values

- *RandomRange* procedure
  - Accepts N>0 in **`eax`**
  - Returns random integer in [0 … N-1] in **`eax`**
- To generate a random number in [lo … hi]:
  - Find number of integer possible in [lo … hi]: range = hi – lo + 1
  - Put range in **`eax`**, and call RandomRange
  - Result in **`eax`** is in [0 … range -1]
  - Add lo to **`eax`**.

RandomRange Example

- Get a random integer in range [18 … 31]



```
mov     eax,hi          ;31
sub     eax,lo          ;31-18 = 13
inc     eax             ;14
call    RandomRange     ;eax in [0..13]
add     eax,lo          ;eax in [18..31]
```

Demo