Topics

- 80x86 Stack
- 32-bit Procedures with Value Parameters
- Additional 32-bit Procedure Options
- 64-bit Procedures
- Macro Definition and Expansion

32-bit Procedures with Value Parameters

Terminology

- *Procedure*
  - a subprogram
  - essentially a self-contained unit
  - Well-defined interface
- Called by
  - Main program
  - another subprogram
- Objective
  - perform a set of tasks
  - may or may not return a value

Procedure Concepts

- Transfer of control from calling program to procedure and back
- Passing parameter values to procedure and results back from the procedure
- Having procedure code that is independent of the calling program

The *cdecl* protocol provides one standard implementation scheme in the 32-bit environment

EAX-->RETURN

Call--> by default we push ret addr

ret---->by default pop Eip

```
.code

main proc
-------------------------
-------------------------
call addproc
---------------------
-----------------
main endp


Addproc Proc
-------------------------
-------------------------
ret
Addproc endp
```

Procedure Definition

- In a code segment with body statements bracketed by `PROC` and `ENDP` directives giving procedure name

```
.CODE
procName   PROC
; procedure body
            ...
procName   ENDP
```

Transferring Control to a Procedure

- In the "main"(or any other) proc , use

  `call` *procName*

- The next instruction executed will be the first instruction in the procedure.

Returning from a Procedure

- In the procedure, use

  `ret`

- The next instruction executed will be the one following the call in the "main"(or any other) proc.

1) EAX

2) call → Pushed → ret address

ret → POP EIP

.Code

Add Proc

===

ret

Add Endp

main Proc

===

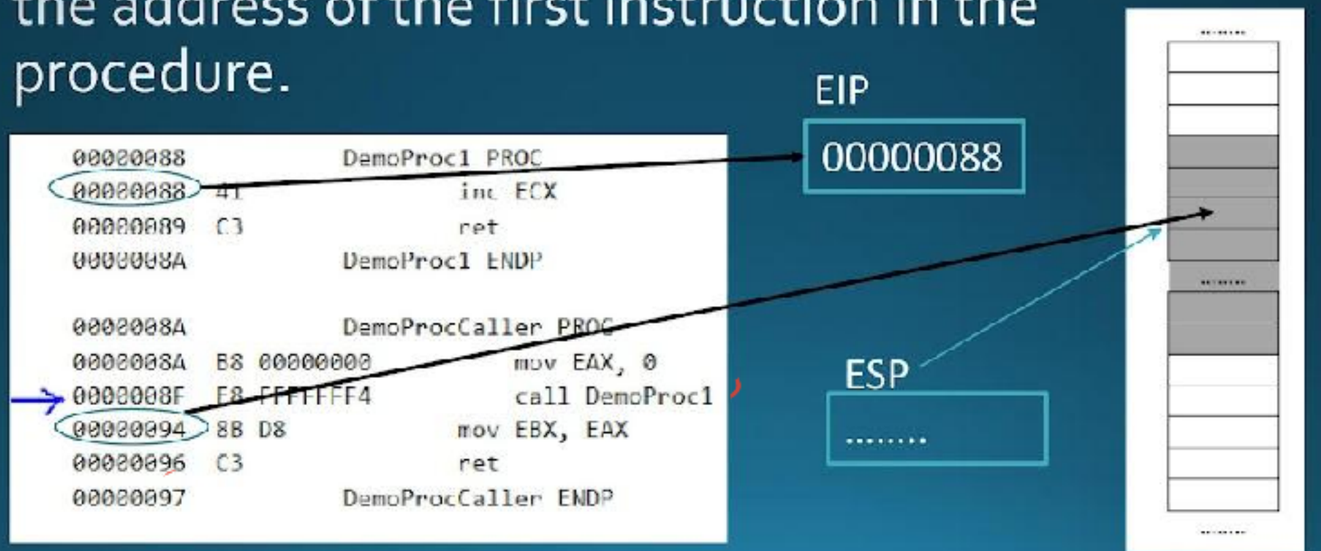
call Add

===

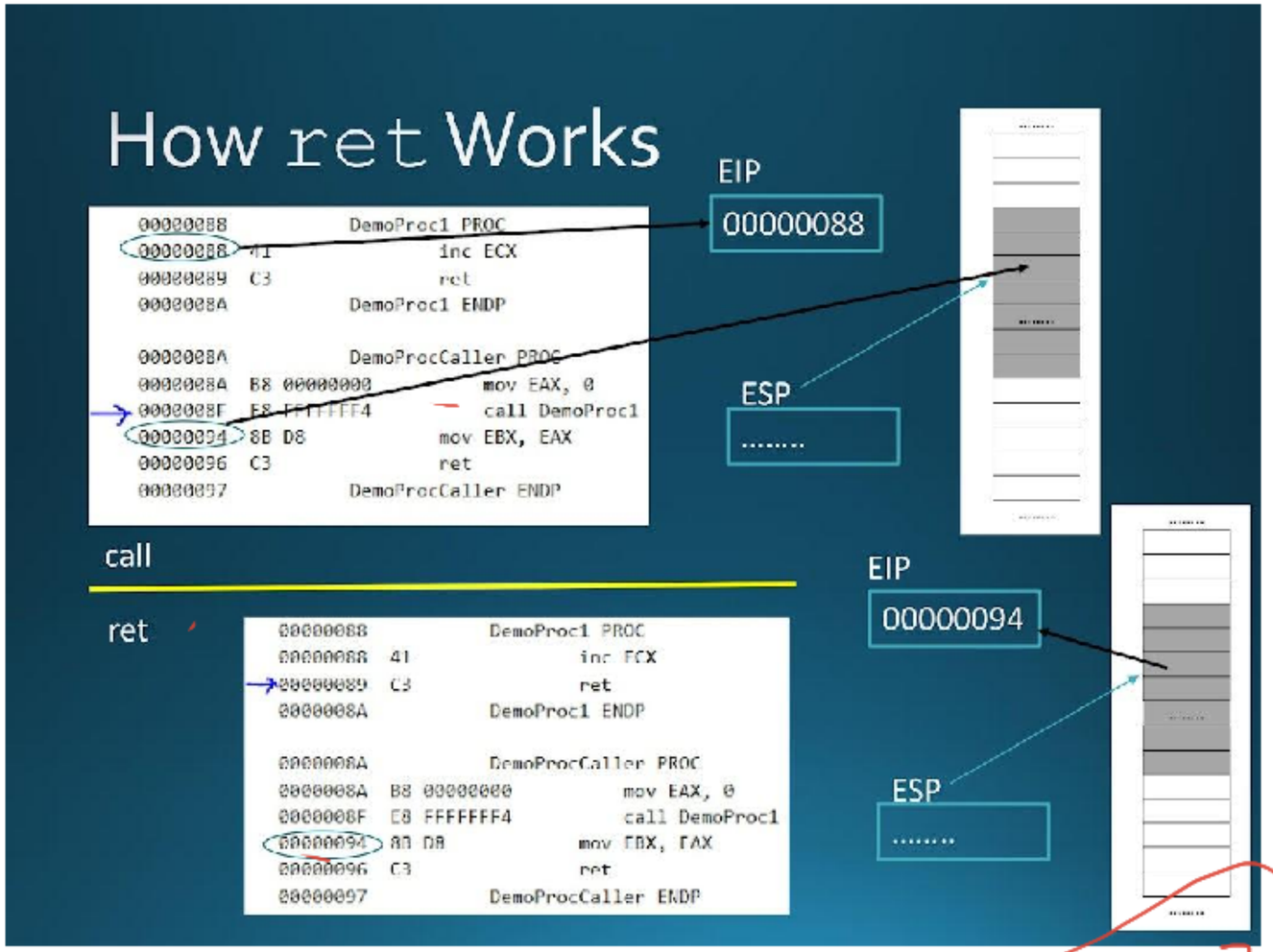
main Endp

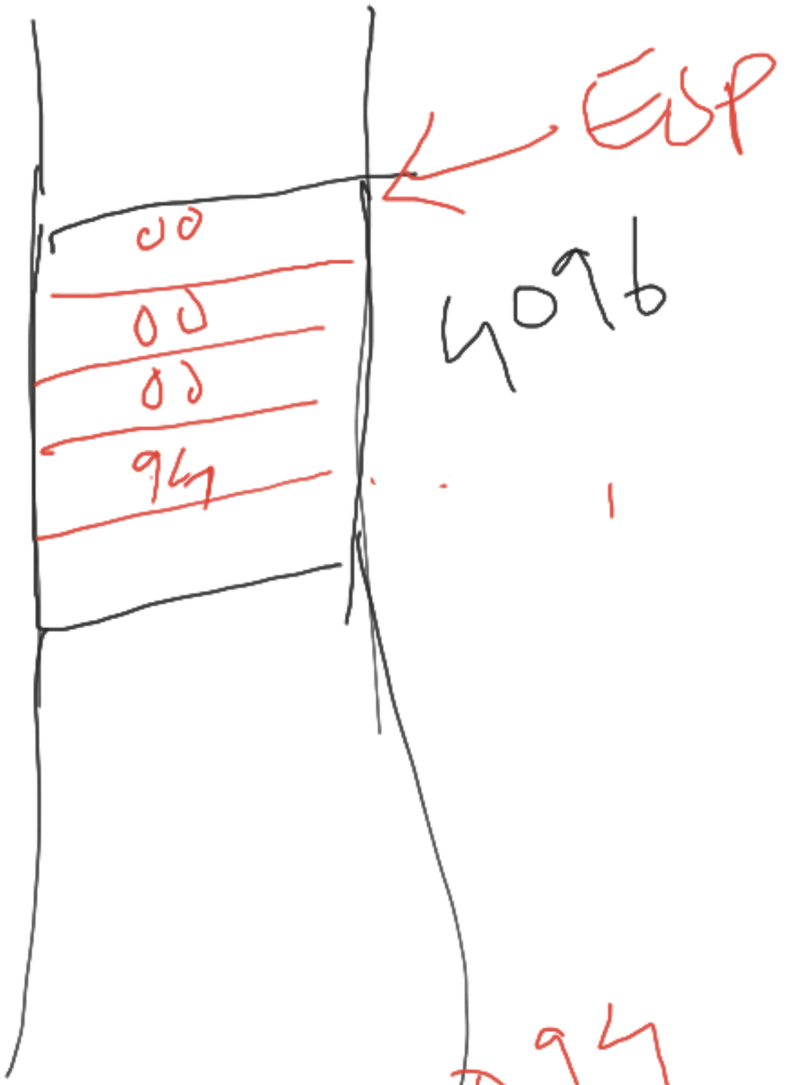
How `call` Works

- The address of the instruction following the `call` is pushed on the stack.
- The instruction pointer register EIP is loaded with the address of the first instruction in the procedure.



EIP-- has the address of next instruction to be executed





ret (POP EIP)

EIP 00000094

`ret` – instruction encoding

Type	Operand	Opcode	Byte length
Near	None	C3	1
Near	Immediate	C2	3
Far	None	CB	1
Far	Immediate	CA	3

```
00000897  C3              ret
00000898  C2 0014         ret 20
```

Parameter Terminology

- A procedure definition often includes *parameters* (also called *formal parameters*).
- These are associated with *arguments* (also called *actual parameters*) when the procedure is called.
- For a procedure's *in* (*pass-by-value*) parameters, values of the arguments are copied to the parameters when the procedure is called.
  - These values are referenced in the procedure using their local names (the identifiers used to define the parameters).

Add(value1, value2)

Pass by value

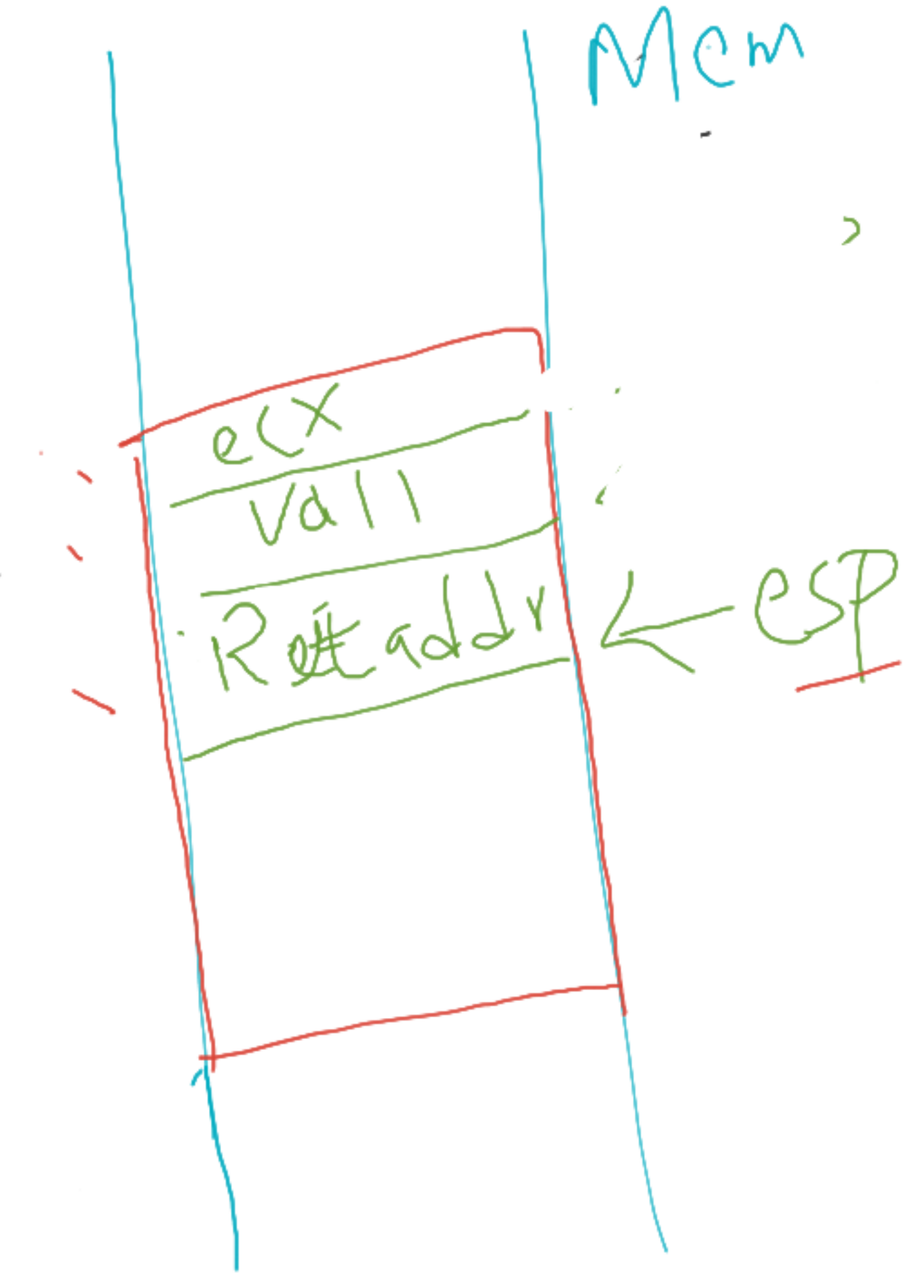
Pass by reference

Implementing Value Parameters

- Parameter values normally passed on the stack
  - Pushed in reverse order from argument list
- Example
  - Design or high-level code: *sum := add2(value1, value2)*
  - 80x86 implementation:

```
push    ecx         ; assuming value2 in ECX
push    value1      ; assuming value1 in memory
call    add2        ; call procedure to find sum
add     esp, 8      ; remove parameters from stack
mov     sum, eax    ; sum in memory
```

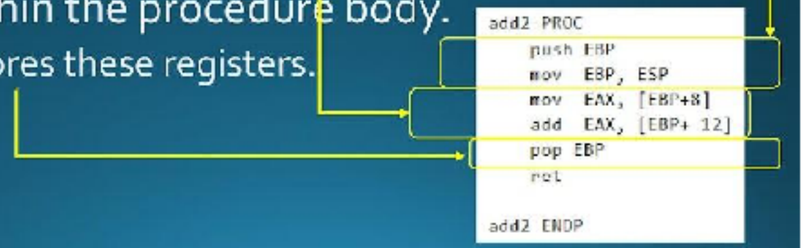
- With the *cdecl* protocol, the calling program must remove parameters from the stack.
- A single integer value is returned in EAX.



procedure result need to be stored in EAX

Procedure Entry Code

- Because the stack pointer ESP may change, a procedure starts with *entry code* to set the base pointer EBP to an address in the stack.
  - This location is fixed until *exit code* restores EBP right before returning.
- In the procedure body, parameters are located relative to the address in EBP.
- Entry code also saves contents of registers that are used locally within the procedure body.
  - Exit code restores these registers.

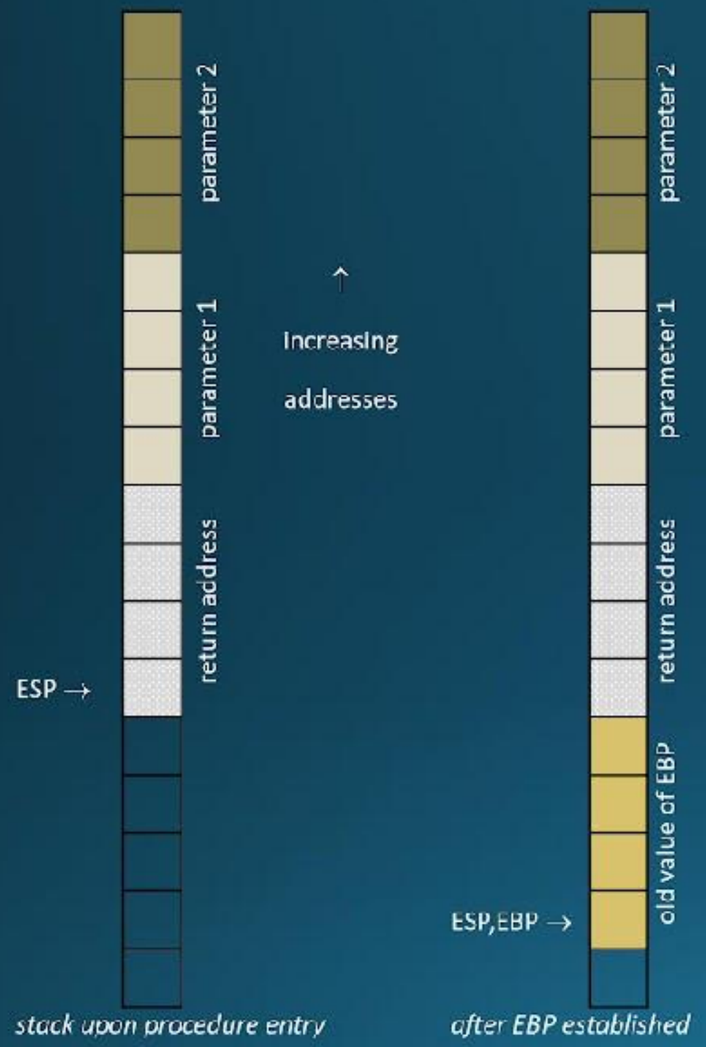


```
add2 PROC
    push EBP
    mov  EBP, ESP
    mov  EAX, [EBP+8]
    add  EAX, [EBP+ 12]
    pop EBP
    ret

add2 ENDP
```

Example Procedure

```
add2    PROC            ; add two words passed on the stack
                        ; return the sum in the EAX register
        push    ebp             ; save EBP
        mov     ebp,esp         ; establish stack frame
        mov     eax,[ebp+8]     ; copy first parameter value
        add     eax,[ebp+12]    ; add second parameter value
        pop     ebp             ; restore EBP
        ret                     ; return
add2    ENDP
```



Stack frame upon entry to procedure *add2* and after EBP established in entry code

Accessing Parameters in a Procedure

- Use *based addressing*
- Because the value is actually on the stack, [EBP+*n*] references the value
- Example:
  `mov eax,[ebp+8]`
  copies the last parameter pushed to EAX

Saving/Restoring Registers

- Push registers in entry code after EBP is established.
- Exit code pops registers in the opposite order.
- Calling program's flag values from can be saved with `pushfd/popfd`.

Entry and Exit Code Summary

- *Entry code:*

```
push    ebp         ; establish stack frame
mov     ebp, esp
push    ...         ; save registers
...
push    ...
pushfd              ; save flags
```

- *Exit code:*

```
popfd               ; restore flags
pop     ...         ; restore registers
...
pop     ...
pop     ebp         ; restore EBP
ret                 ; return
```

Procedure Call Summary

calling program code

- push arguments on stack in right-to-left order
- call procedure

procedure code

- save EBP and establish stack frame
- save registers used by procedure
- access parameter values using based addressing in procedure body
- return value, if any, goes in EAX
- restore saved registers and EBP
- return

calling program code

- add number of bytes of parameters to ESP
- for value-returning procedure, use value in EAX

```
.4096 stack

.data
value1 dword ?
value2 dword ?

.code

add proc

push ebp
mov ebp,esp
mov eax,[ebp+8]
add eax,[ebp+12]
pop ebp
ret

add endp

main proc

push value2
push value1
call add
add esp,8
mov sum,eax

main endp
```

pass by value example

Ebp is a global register it may have valuable information needed for another procedure

add(value1,value2)