EC325 Microprocessors Procedures and the Stack

Yasser F. O. Mohammad

```
jmp     quit
        .
        .
quit:   INVOKE ExitProcess, 0
        .
        .
```

REMINDER 1: Unconditional Jmp

- Jmp statement
- Jmp offset
  - Offset = register, or memory location (signed)
  - Offset is added to the address of *next* instruction
- Jmp Types:
  - Relative Jump = Interasegment Jump = changes EIP
  - Far Jump = Intersegment Jump = changes CS, EIP
  - Task Switch = Jump to a different task (privileged)

Offset Type	Offset Size	Maximum offset
Relative short	4 bytes	-2147483648 → 2147483647
Relative near	Single byte	-128→127
Register indirect	4 bytes	-2147483648 → 2147483647
Memory indirect	4 bytes	-2147483648 → 2147483647

Why do we need relative short jmp?

REMINDER 2: Conditional Jump

- J* targetStatement
- * identifies the condition to take the jump

REMINDER 3: LOOP instruction

- loop   statement
  - Statetement must be short distance from the instruction (-128→ 127 bytes)
  - Does the following:
    - ECX=ECX-1
    - If ECX==0 then continue to next statement
    - If ECX ≠ 0 then jump to *statement*
  - Similar to a high level For-Loop with count in ECX

```
for(     ; ECX>0;  ECX--){
    .
    .
}
```

What is the stack?

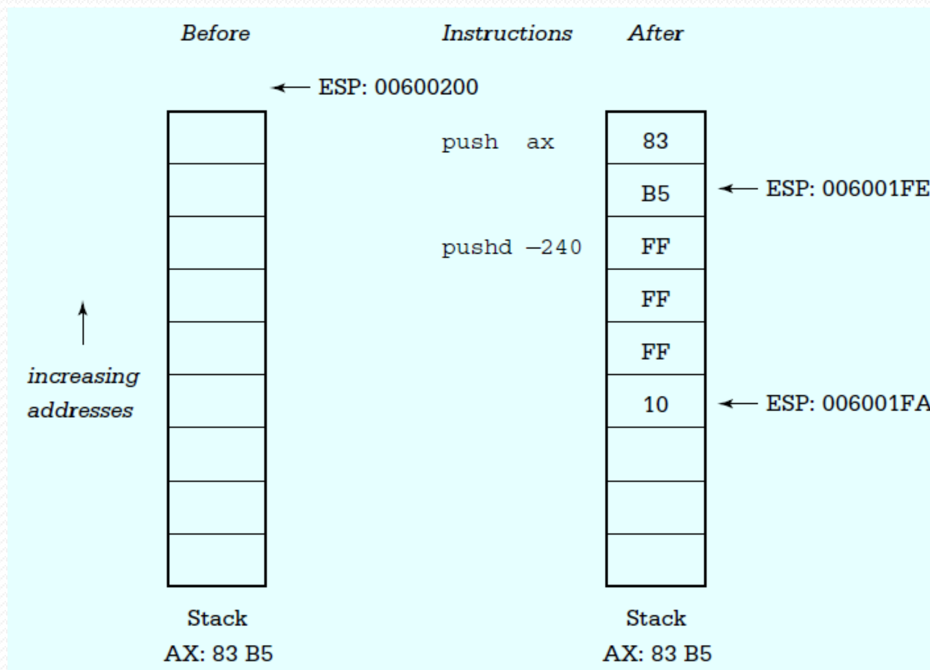
- A data structure with two operations:
  - push: adds on the top of the stack
  - Pop: pops from the top of the stack
- Allocated using .STACK in MASM
  - Of course the memory is still accessible as general memory
- Accessed by ESP (usually!!)
- Used for parameter passing during function calls
  - Automatically manage ESP
- Can be used as you see fit
  - You manage everything

.STACK

- Allocates a space in memory for the stack
- ESP points to the byte just above the allocated space for the stack.
- In general ESP points to the location of the last byte already written to the stack.

Push instruction

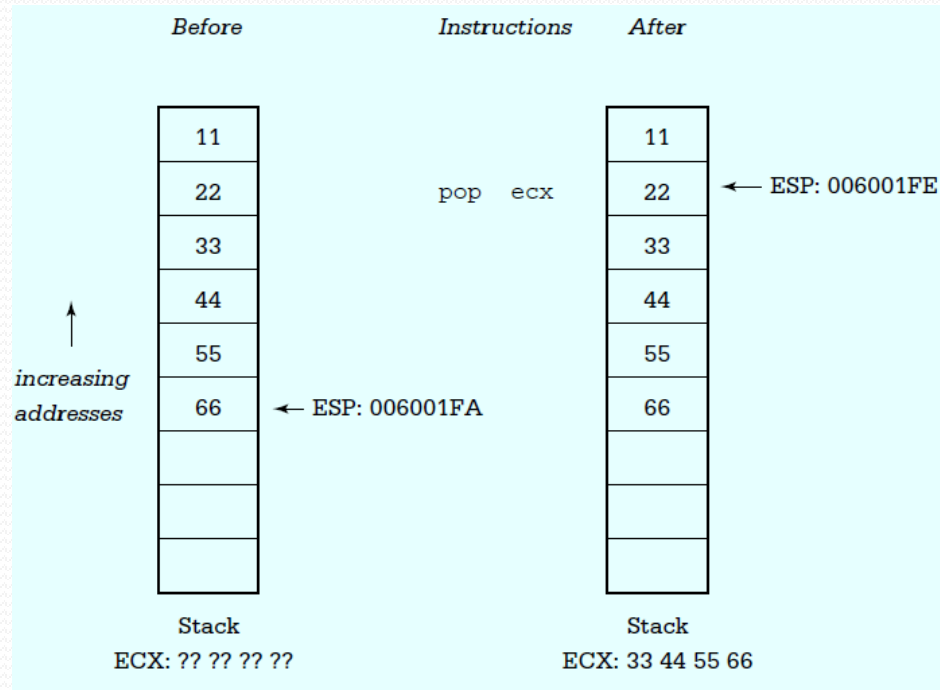
- push   source
  1. Decrements ESP by the size of *source.*
  2. Copies *source* to the location pointed to by ESP.



It grows downward !!!!

Pop instruction

- pop   source
  - Copies *source*-size bytes from [ESP] to source.
  - Increments ESP by the size of *source*.



What can we push and pop

- register
  - EAX or AX
  - ECX or CX
  - EDX or DX
  - EBX or BX
  - ESP or SP
  - EBP or BP
  - ESI or SI
  - EDI or DI
- segment register
  - DS
  - ES
  - SS
  - FS
  - GS
- memory word
- memory doubleword

Flags to/from the stack

- pushf/popf
  - Pushes/pops the flags register (2 bytes)
- pushfd/popfd
  - Pushes/pops the extended flags register (4 bytes)

All registers to/from the stack

- pusha
  - Pushes all registers in this order:
    - AX, CX, DX, BX, SP, BP, SI, DI
  - SP value pushed is the value BEFORE pushing AX

- popa
  - Pops all registers in this order:
    - DI, SI, BP, SP(Discarded), BX, DX, CX, AX
  - SP value is discarded after pushing not to modify current SP

All registers to/from the stack

- pushad
  - Pushes all registers in this order:
    - EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
  - ESP value pushed is the value BEFORE pushing EAX

- popad
  - Pops all registers in this order:
    - EDI, ESI, EBP, ESP(Discarded), EBX, EDX, ECX, EAX
  - ESP value is discarded after pushing not to modify current ESP

Note about pushing

- Some operating systems including Windows require that parameters for functions are double word-aligned.

- To be safe push and pop DWORDs not WORDs

Procedures

- The way to implement functions and function calls in IA32
- Always comes in the code segment (after .CODE)
- Has the following anatomy:

```
label PROC [[distance]] [[langtype]] [[visibility]] [[<prologuearg>]] [[USES
        reglist]] [[, parameter [[:tag]]]]...

        statements
        [ret]
label ENDP
```

How to call a procedure

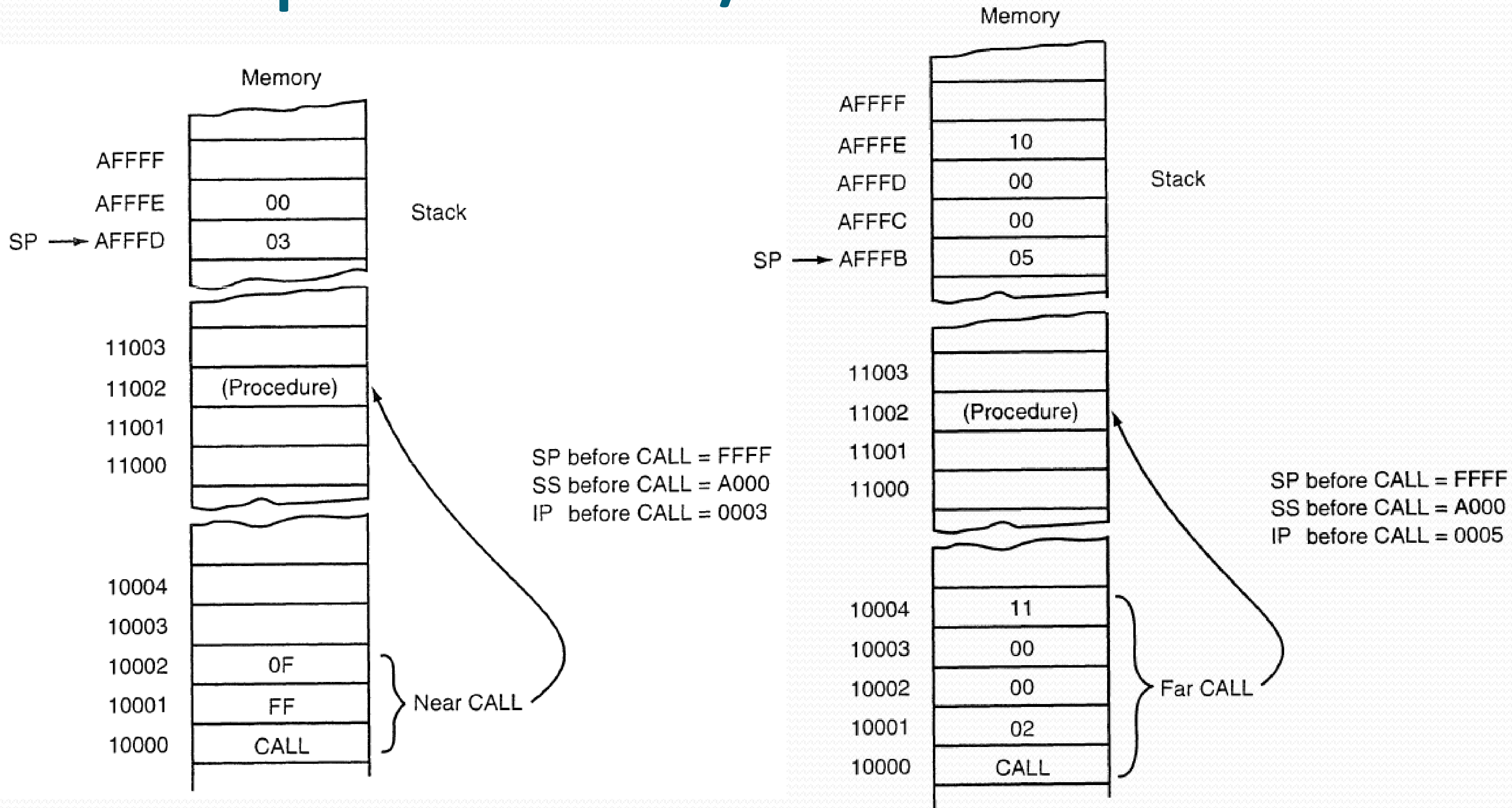
- call procedureLabel
  - Does not by itself do any parameter passing
  - You do parameter passing yourself!!!!!!
  - Does two things
    1. Pushes the return address to the stack
    2. Jumps to the address of the procedure

  - As in JMP, ±32K displacement is added to EIP/IP to do the jump

How long is the return address

- NEAR
  - IP (WORD)
- NEAR32
  - EIP (DWORD)
- FAR
  - 8086: CS:IP (2 WORDs)
  - 80386: CS:EIP (1 WORD+1 DWORD)

Examples NEAR/FAR



Example NEAR32

Before

```
C:\Documents\BOOK\Code\fig6-5.asm
Initialize  ENDP

_start:                                 ; program entry point
            call    Initialize          ; initialize variables

; -- other program tasks here

            call    Initialize          ; reinitialize variables

; -- more program tasks here

            INVOKE ExitProcess, 0       ; exit with return code
PUBLIC _start                         ; make entry point public
```

```
Registers
EAX  0040103e
EBX  00530000
ECX  0063ff68
EDX  bffc14a0
ESI  81636718
EDI  00000000
EIP  0040103e
ESP  0063fe3c
EBP  0063ff78
EFL  00000a82
CS   0157
DS   015f
ES   015f
SS   015f
```

```
Memory(0x0063fe30)
0x0063FE30 3a 93 f8 bf 00 05 00 00 4b 2d fc ff 49 93 f8 bf :.......K-..I...
0x0063FE40 00 00 00 00 18 67 63 81 00 00 53 00 46 69 67 36 .....gc...S.Fig6
```

After

```
C:\Documents\BOOK\Code\fig6-5.asm
.CODE                                   ; program code

Initialize  PROC    NEAR32
            mov     Count1,0            ; zero first count
            mov     Count2,0            ; zero second count
            mov     Total1,0            ; zero first total
            mov     Total2,0            ; zero second total
            mov     ebx,0               ; zero balance
            ret                         ; return
Initialize  ENDP

_start:                                 ; program entry point
            call    Initialize          ; initialize variables
```

```
Registers
EAX  0040103e
EBX  00530000
ECX  0063ff68
EDX  bffc14a0
ESI  81636718
EDI  00000000
EIP  00401010
ESP  0063fe38
EBP  0063ff78
EFL  00000a82
CS   0157
DS   015f
ES   015f
SS   015f
```

```
Memory(0x0063fe30)
0x0063FE30 3a 93 f8 bf 00 05 00 00 43 10 40 00 49 93 f8 bf :.......C.@.I...
0x0063FE40 00 00 00 00 18 67 63 81 00 00 53 00 46 69 67 36 .....gc...S.Fig6
```

Indirect call

- CALL register
- CALL memaddress

  - Calls the procedure which address is referenced
  - Near version uses DWORD registers and addresses as new EIP
  - Far version can only use memory because it needs 6 bytes!!

How to pass parameters

- Push them to the stack before CALL
- Put them to known memory location before CALL
- Put them to registers before CALL

Returning from Procedures

- ret
  - Returns control to the caller
  - You must return the return value yourself!!!

  - Does one thing
    1. Pops the return address from the stack to IP, EIP, CS:IP

  - This is a JMP

Returning with cleaning

- ret *count*
  - Count is an immediate
  - Indicates how many bytes the ESP should be incremented with AFTER the return
  - Used to discard input parameters on the stack

How to return a value

- ~~Push it to the stack~~
- Leave it in a known memory location
- Leave it in a known register

Example

```
.STACK  4096               ; reserve 4096-byte stack

.DATA                      ; reserve storage for data
Count1       DWORD   11111111h
Count2       DWORD   22222222h
Total1       DWORD   33333333h
Total2       DWORD   44444444h
;            other data here

.CODE                              ; program code

Initialize  PROC   NEAR32
            mov    Count1,0            ; zero first count
            mov    Count2,0            ; zero second count
            mov    Total1,0            ; zero first total
            mov    Total2,0            ; zero second total
            mov    ebx,0               ; zero balance
            ret                        ; return
Initialize  ENDP

_start:                                ; program entry point
            call   Initialize          ; initialize variables

; — other program tasks here

            call   Initialize          ; reinitialize variables

; — more program tasks here

            INVOKE ExitProcess, 0   ; exit with return code 0
PUBLIC _start                      ; make entry point public

END                                ; end of source code
```

Declaration

Call

How to put procedures in a different file

- Declare them PUBLIC in the defining file
  - PUBLIC proc_name1, [proc_name1,....]
  - E.g. PUBLIC Initialize

- Declare them external in the calling file
  - EXTRN proc_name1:Type, [proc_name1:Type,....]
  - E.g. EXTRN Initialize:NEAR32

Example

```
; procedure to compute integer square root of number Nbr
; Nbr is passed to the procedure in EAX
; The square root SqRt is returned in EAX
; Other registers are unchanged.
; author:  R. Detmer    revised:  10/97

Root       PROC  NEAR32
           push  ebx            ; save registers
           push  ecx
           mov   ebx, 0         ; SqRt := 0
WhileLE:   mov   ecx, ebx       ; copy SqRt
           imul  ecx, ebx       ; SqRt*SqRt
           cmp   ecx, eax       ; SqRt*SqRt <= Nbr ?
           jnle  EndWhileLE     ; exit if not
           inc   ebx            ; add 1 to SqRt
           jmp   WhileLE        ; repeat
EndWhileLE:
           dec   ebx            ; subtract 1 from SqRt
           mov   eax, ebx       ; return SqRt in AX
           pop   ecx            ; restore registers
           pop   ebx
           ret                 ; return
Root       ENDP
```

Parameter passing

- Types of parameters:
  - In: Pass-by-Value
  - In-out: Pass-by-Reference

- Types of variables:
  - Local: specific to the procedure (visible only inside)
  - Global: visible outside

- Simplest parameter passing approach
  - Use registers
  - Use them as global variables

- Simplest local variable approach
  - Use registers

Example

- Passing two DWORDS:

```
push    Value1      ; first argument value
push    ecx         ; second argument value
call    Add2        ; call procedure to find sum
add     esp,8       ; remove parameters from stack
```

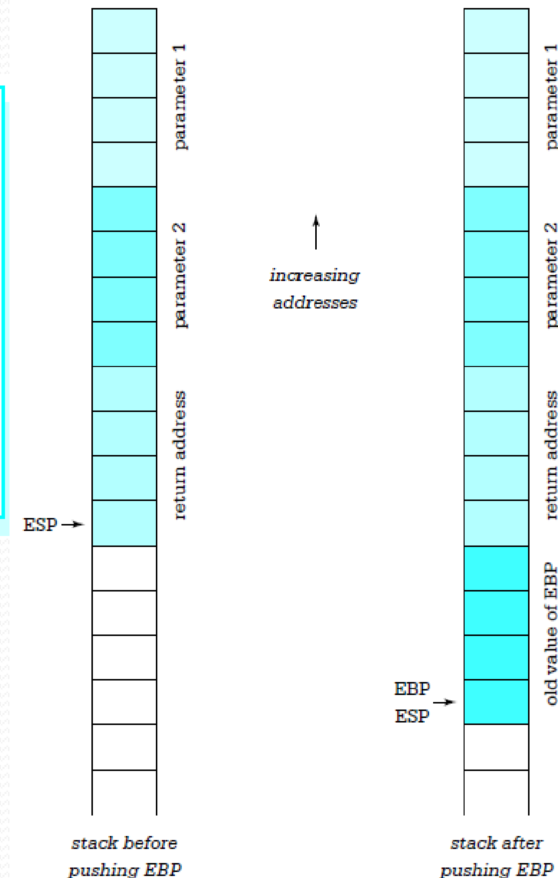
- You must readjust this ESP (by subtracting 8) before returning from the procedure. Why? How?

Stack for Parameter Passing

- Usually, we use EBP to access parameters/variables on the stack

```
Add2        PROC NEAR32   ; 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 second parameter
                                     ; value
            add    eax,[ebp+12]      ; add first parameter value
            pop    ebp               ; restore EBP
            ret                      ; return
Add2        ENDP
```

```
push    Value1        ; first argument value
push    ecx           ; second argument value
call    Add2          ; call procedure to find sum
add     esp,8         ; remove parameters from stack
```



Stack for Local Variables

int GCD(int , int);

Entrance Code

```
gcd := number1;
remainder := number2;

until (remainder = 0) loop
   dividend := gcd;
   gcd := remainder;
   remainder := dividend mod gcd;
end until;
```

Exit Code

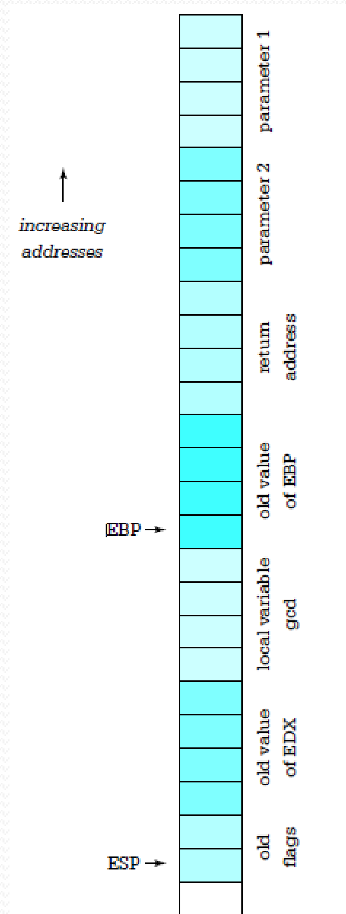
```
PUBLIC GCD
; Procedure to compute the greatest common divisor of two
; doubleword-size integer parameters passed on the stack.
; The GCD is returned in EAX.
; No other register is changed.  Flags are unchanged.
; Author:  R. Detmer    Revised:  10/97

GCD     PROC    NEAR32
        push    ebp             ; establish stack frame
        mov     ebp,esp
        sub     esp,4           ; space for one local doubleword
        push    edx             ; save EDX
        pushf                   ; save flags

        mov     eax,[ebp+8]     ; get Number1
        mov     [ebp-4],eax     ; GCD := Number1
        mov     edx,[ebp+8]     ; Remainder := Number2
until0: mov     eax,[ebp-4]     ; Dividend := GCD
        mov     [ebp-4],edx     ; GCD := Remainder
        mov     edx,0           ; extend Dividend to doubleword
        div     DWORD PTR [ebp-4]    ; Remainder in EDX
        cmp     edx, 0          ; remainder = 0?
        jnz     until0          ; repeat if not

        mov     eax,[ebp-4]     ; copy GCD to EAX
        popf                    ; restore flags
        pop     edx             ; restore EDX
        mov     esp,ebp         ; restore ESP
        pop     ebp             ; restore EBP
        ret     8               ; return, discarding parameters
GCD     ENDP
        END
```

Point To Params

Allocate Local Variables

Save Flags

Stack usage with params and locals



Typical Function (PROC)

Entry code:

```
push    ebp             ; establish stack frame
mov     ebp,esp
sub     esp,n           ; n bytes of local variables space
push    ...             ; save registers
...
push    ...
pushf                   ; save flags
```

Exit code:

```
popf                    ; restore flags
pop     ...             ; restore registers
...
pop     ...
mov     esp,ebp         ; restore ESP if local variables used
pop     ebp             ; restore EBP
ret                     ; return
```

IA32 support for compilers

- enter   localBytes, nestingLevel

  - Nesting level = zero

```
push    ebp
mov     ebp,esp
sub     esp, localBytes
```

  - Nesting Level > zero
    - Push ESP from nestingLevel-1 to 0 to the stack
    - Allows nested blocks access to local variables of their parents

IA32 Support for compilers 2

- leave
  - Usually used just before returning (ret)
  - Does the following:

```
mov    esp,ebp      ; restore ESP
pop    ebp          ; restore EBP
```

  - Reverses the effects of *enter* on the stack

MASM support for you

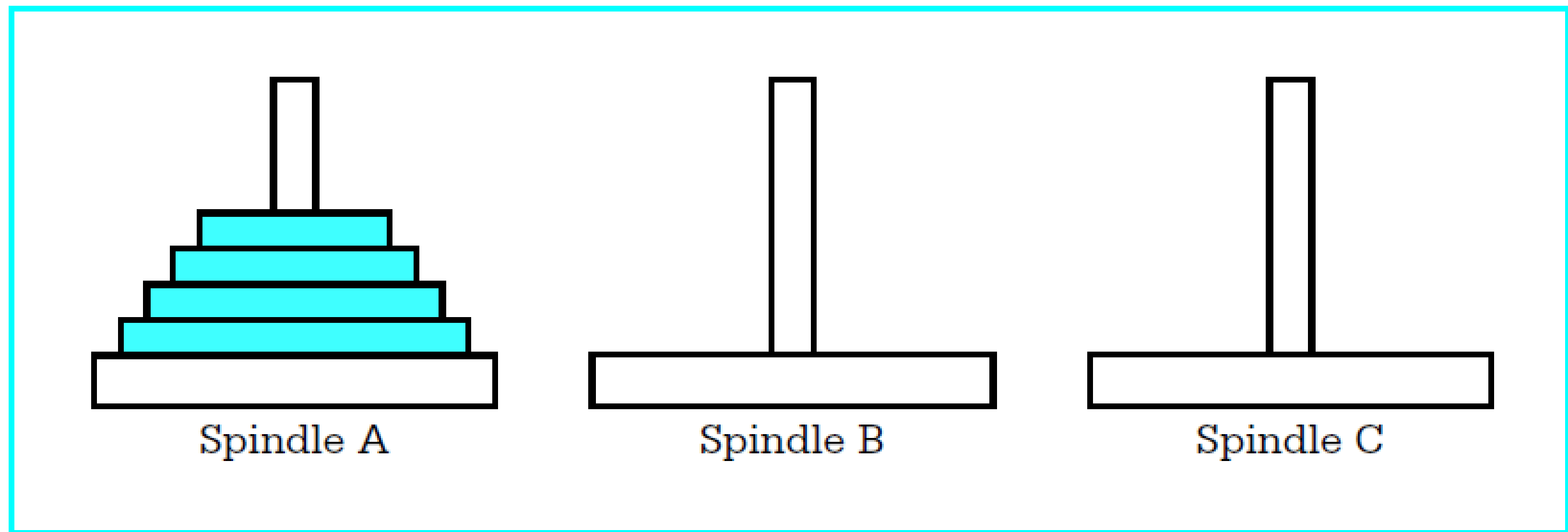
- INVOKE procName, param1, param2, ….
- A directive not an instruction. Even if it does not start with a '.'
- Does:

```
PUSH paramn
.
.
.
PUSH param1
CALL procName
```

Recursion

- A function directly or indirectly calling itself.
- This is one motivator to store local variables and parameters on the stack. Why?
- This is the most common reason for stack overflow problems

Towers of Hanoi puzzle



- Move all disks from A to B one at a time without ever having a disk under a larger one. C can be used as temporary location.

Towers of Hanoi Solution

- If N.Disks==1
  - Move it to B
- If N.Disks>1
  - Move largest N.Disks-1 to C
  - Move the remaining disk (on A) to B
  - Now solve the problem of moving N.Disks-1 from C to B using A as a temporary location

Pseudo code of Towers of Hanoi

```
procedure Move(NbrDisks, Source, Destination, Spare);
begin
        if NbrDisks = 1
        then
                display "Move disk from ", Source, " to ", Destination
        else
                Move(NbrDisks – 1, Source, Spare, Destination);
                Move(1, Source, Destination, Spare);
                Move(NbrDisks – 1, Spare, Destination, Source);
        end if;
end procedure Move;
begin {main program}
        prompt for and input Number;
        Move(Number, 'A', 'B', 'C');
end;
```

Assembly Solution Page 1

```
; program to print instructions for "Towers of Hanoi" puzzle
; author:  R. Detmer   revised:  10/97

.386
.MODEL FLAT

ExitProcess PROTO NEAR32 stdcall, dwExitCode:DWORD

include io.h            ; header file for input/output

cr      equu    0dh     ; carriage return character
Lf      equ     0ah     ; line feed

.STACK  4096            ; reserve 4096-byte stack

.DATA                   ; reserve storage for data
prompt      BYTE   cr,Lf,'How many disks?  ',0
number      BYTE   16 DUP (?)
message     BYTE   cr,Lf,'Move disk from spindle '
source      BYTE   ?
            BYTE   ' to spindle '
dest        BYTE   ?
            BYTE   '.',0
```

(continued)

Assembly Solution Page 2

```
.CODE
Move        PROC NEAR32
; procedure Move(NbrDisks : integer; { number of disks to move }
;           Source, Dest, Spare : character { spindles to use }  )
; parameters are passed in words on the stack

            push   ebp           ; save base pointer
            mov    ebp,esp       ; copy stack pointer
            push   eax           ; save registers
            push   ebx

            cmp    WORD PTR [ebp+14],1   ; NbrDisks = 1?
            jne    elseMore      ; skip if more than 1
            mov    bx,[ebp+12]   ; Source
            mov    source,bl     ; copy character to output
            mov    bx,[ebp+10]   ; destination
            mov    dest,bl       ; copy character to output
            output message       ; print line
            jmp    endIfOne      ; return
elseMore:   mov    ax,[ebp+14]   ; get NbrDisks
            dec    ax            ; NbrDisks - 1
            push   ax            ; parameter 1: NbrDisks-1
            pushw  [ebp+12]      ; parameter 2: source does not change
            pushw  [ebp+8]       ; parameter 3: old spare is new destination
            pushw  [ebp+10]      ; parameter 4: old destination is new spare
            call   Move          ; Move(NbrDisks-1,Source,Spare,Destination)
            add    esp,8         ; remove parameters from stack

            pushw  1             ; parameter 1: 1
            pushw  [ebp+12]      ; parameter 2: source does not change
            pushw  [ebp+10]      ; parameter 3: destination unchanged
            pushw  [ebp+8]       ; parameter 4: spare unchanged
            call   Move          ; Move(1,Source,Destination,Spare)
            add    esp,8         ; remove parameters from stack

            push   ax            ; parameter 1: NbrDisks-1
            pushw  [ebp+8]       ; parameter 2: source is original spare
            pushw  [ebp+10]      ; parameter 3: original destination
            pushw  [ebp+12]      ; parameter 4: original source is spare
            call   Move          ; Move(NbrDisks-1,Spare,Destination,Source)
```

(continued)

```
            add    esp,8           ; remove parameters from stack
endIfOne:
            pop    ebx             ; restore registers
            pop    eax
            pop    ebp             ; restore base pointer
            ret                    ; return
Move        ENDP

_start:     output prompt         ; ask for number of disks
            input  number,16      ; read ASCII characters
            atoi   number         ; convert to integer
            push   ax             ; argument 1: Number
            mov    al,'A'         ; argument 2: ' A'
            push   ax
            mov    al,'B'         ; argument 3: ' B'
            push   ax
            mov    al,'C'         ; argument 4: ' C'
            push   ax
            call   Move           ; Move(Number,Source,Dest,Spare)
            add    esp,8          ; remove parameters from stack

            INVOKE  ExitProcess, 0  ; exit with return code 0

PUBLIC _start                     ; make entry point public
END                               ; end of source code
```

Interrupts*

- Hardware Interrupts
  - Requested by hardware to avoid polling
  - Controlled by the Programmable Interrupt Controller PIC
  - I flag controls if the processor accepts interrupts

- Software Interrupts
  - Requested in the program
  - Simulates interrupts
  - Has nothing to do with the PIC

How ALL Interrupts are handled*

- There are 256 different interrupt types (numbers).
- First 1 or 2K memory locations contain interrupt vectors.
- Interrupt vector of interrupt X: the address of the interrupt handling routine (IHR) to be called when X is raised
- Interrupt number → Interrupt vector (IV)
  - Real: Multiply by 4 → 00:[IV]=address of IHR
  - Protected: Multiply by 8 →00:[IV]= descriptor of the IHR address

INT*

- INT *number*
  1. Calculate IV=number * 4 or 8 (Real/Protected)
  2. Push flags
  3. Clear T and I flags (Traps and hardware interrupts)
  4. Push CS
  5. Read new CS from CS:[IV]
  6. Push IP/EIP onto the stack
  7. Read new IP/EIP from CS:[IV+2]
  8. Jump to new CS:IP/EIP

Used for system calls (2 bytes) instead of FAR calls (5 bytes)

Common Interrupts*

Number	Address	Microprocessor	Function
0	0H–3H	All	Divide error
1	4H–7H	All	Aingle-step
2	8H–BH	All	NMI pin
3	CH–FH	All	Breakpoint
4	10H–13H	All	Interrupt on overflow
5	14H–17H	80186–Pentium Pro	Bound instruction
6	18H–1BH	80186–Pentium Pro	Invalid opcode
7	1CH–1FH	80186–Pentium Pro	Coprocessor emulation
8	20H–23H	80386–Pentium Pro	Double fault
9	24H–27H	80386	Coprocessor segment overrun
A	28H–2BH	80386–Pentium Pro	Invalid task state segment
B	2CH–2FH	80386–Pentium Pro	Segment not present
C	30H–33H	80386–Pentium Pro	Stack fault
D	34H–37H	80386–Pentium Pro	General protection fault (GPF)
E	38H–3BH	80386–Pentium Pro	Page fault
F	3CH–3FH	—	Reserved
10	40H–43H	80286–Pentium Pro	Floating-point error
11	44H–47H	80486SX	Alignment check interrupt
12	48H–4FH	Pentium/Pentium Pro	Machine check exception
13–1F	50H–7FH	—	Reserved
20–FF	80H–3FFH	—	User interrupts

IRET/IRETD*

- IRET
  1. POP IP
  2. POP CS
  3. POP flags

- IRET=POPF+FAR RET
- IRET is used in real mode
- IRETD is used in protected mode (POPs EIP)

INTO*

- INTO
  - If OF=1 does INT 4 otherwise nothing

- Used to check for overflows

How to call without a CALL

- S/360 (1960)
  - 32 GPRs
  - Call is done as follows:
    - Allocate space to save 32 GPRs
    - Copy all GPRs to it
    - Put its address in R13
    - Copy ProcAddress to R15
    - Jump and Link (copies IP to R14 then JMP R15)
  - Return is done as follows:
    - JMP R14
  - Parameter Passing is done as follows:
    - Put parameters in memory
    - Make a list of pointers to parameters in memory at address ADDR
    - MOV R1,ADDR
    - Call
  - *NO recursive calls!!*