

*ECE390*  
*Computer Engineering II*  
*Lecture 7*



Dr. Zbigniew Kalbarczyk  
University of Illinois at Urbana- Champaign

*Lecture outline*



- Procedures (cont.)
- Recursion

## Local variable storage

- You may allocate stack space for use as local variables within procedures
- Subtract from the stack pointer the number of bytes you need to allocate after setting up the stack frame
- At the end of your procedure, just add the same amount you subtracted to free the memory you allocated just before you *pop bp*
- In the procedure, use the stack frame to address local variable storage

## Local variable storage

```
Q equ 0
R equ -2
MyProc
    ;setup stack frame
    push bp
    mov bp, sp
    sub sp, 4          ;allocate space for two words
    push ax
    push dx
    .....
    mov [bp+Q], ax    ;use of local variables
    mov dx, [bp+R]
    .....
    pop dx
    pop ax
    add sp, 4         ;release temporary space
    pop bp           ;restore original bp
    ret
```

## *Things to remember*

- Always make sure that your stack is consistent (a proc does not leave information on the stack that was not there before the procedure ran)
- Always remember to save registers you modify that do not contain return values

## *Passing Parameters on the Stack*

- Stack can be used to pass parameter(s) to a procedure
- The caller pushes the procedure parameter onto the stack and the callee finds the parameter there
- When the procedure completes its task, the parameter should be popped from the stack

```
LEN    EQU    80
CR     EQU    0dh
LF     EQU    0ah
```

```
Prompt1    DB    'Input a string', 0
Prompt2    DB    'Do another? ', 0
String     times (LEN +1) DB 0
```

## Passing Parameters on the Stack Example (cont.)

```
..start
MAIN:
.Begin:
    push    Prompt1
    call    putStr
    push    String
    call    getStr
    push    String
    call    putStr

    push    Prompt2
    call    putStr

    push    String
    call    getStr
    call    putStr

    mov     bx, String
    cmp    BYTE [bx], 'y'
    je     .Begin

    mov     ax, 4c00h
    int    21h
```

Z. Kalbarczyk

ECE390

## Passing Parameters on the Stack Example (cont.)

```
;/OFFSET of string to be printed must  
;be on the stack and the string must  
;be null terminated
```

```
putStr:
    push bp
    mov bp, sp
    push ax
    push bx
    push dx

    mov bx, [bp + 4] ;expect bx
    to point to string
    mov ah, 2h

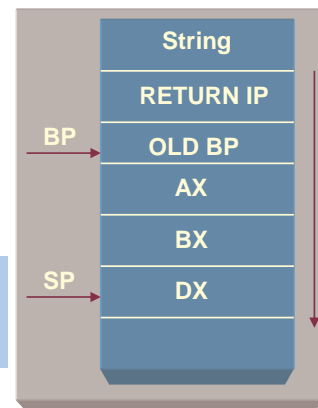
    ; prepare to print a char with 21h
.nextChar:
    cmp BYTE[bx], 0h ;check for
    null terminator
```

Z. Kalbarczyk

```
je .foundEnd ;when found exit
mov dl, [bx]
int 21h ; print with 21h
inc bx ;point to next char
jmp .nextChar

.foundEnd:
    pop dx
    pop bx
    pop ax
    pop bp
    ret 2
```

Removes passed parameters from the stack



ECE390

## Passing Parameters on the Stack Example (cont.)

*;OFFSET of large enough buffer must  
;have been pushed onto stack  
;string will be null terminated*

getStr:

```
push    bp
mov     bp, sp
push    ax
push    bx
```

```
mov bx, [bp + 4] ;base  
address of storing buffer
```

```
mov     ah, 01h
```

.getLoop:

```
int     21h
cmp     al, CR
        ;look for CR in al
```

```
je     .getEnd
```

```
mov [bx], al
        ;bx points to storage  
location
```

```
inc    bx
```

```
jmp    .getLoop
```

.getEnd:

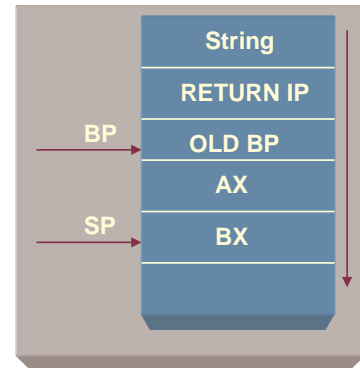
```
mov BYTE[bx], 0
        ;CR is converted in null  
term
```

```
pop    bx
```

```
pop    ax
```

```
pop    bp
```

```
ret    2
```



Z. Kalbarczyk

ECE390

## C style procedures

- Assembly procedures that can be called by C programs
- Must follow the same calling procedure that C compilers use when building C programs
- You can also call C functions from assembly programs using the same protocol

Z. Kalbarczyk

ECE390

## *C style procedures*

- Suppose a C program calls an assembly procedure as follows
  - Proc1(a, b, c)
- Assume:
  - each argument is word-sized
  - the C compiler generates code to push a, b, and c onto the stack
- The assembly language procedure must be assembled with the correct ret (near or far) and stack handling of its procedures matching the corresponding values in the high-level module

## *C style procedures*

- Assume C program always makes far call
  - So far return (retf) must end the C style procedure
  - Return CS gets pushed onto the stack as well as return IP
  - Makes a difference when accessing arguments using the stack frame
- C compilers push arguments in reverse order, from right to left.
  - c is pushed first, then b, then a
  - Lowest index from BP corresponds to first argument

BP+10	c
BP+8	b
BP+6	a
BP+4	Return CS
BP+2	Return IP
BP	Saved BP

Proc1(a, b, c)

## *C style procedures*

---

- If the returned value needs four or fewer bytes, it is by default returned in registers
  - one or two bytes - returned in AX
  - three or four bytes - returned in AX (low word) and in DX (high byte or word), (in EAX in 32-bit mode)
- More than four bytes
  - the called procedure stores data in some address and returns the offset and segment parts of that address in AX and DX, respectively

## *C style procedures*

---

- Caller is responsible for clearing the arguments from the stack as soon as it regains control after the call
  - this is done by the compiler that generates the appropriate code
  - a far procedure called from C should end with RETF instead of RET

## Example

- Calling and NASM proc from a C program – the proc lets you display a string at a given row and column

```
# include <stdio.h>
extern void placeStr (char *, unsigned, unsigned);

void main (void)
{
    int    n;
    for (n = 10; n < 20; ++n)
        placeStr ("This is the string", n, 45);
}
```

## Example

```
GLOBAL _placeStr                                ;set cursor position
                                                MOV    AH, 02h
                                                INT    10h
SEGMENT code
_placeStr                                       ;point to string
; setup stack frame and save state             MOV    BX, [BP+6]
;call outAsc to disp string
PUSH    BP                                     call outAsc
MOV     BP, SP
PUSH    AX
PUSH    BX
PUSH    DX
                                                ;restore state
; get current page - returns in BH             POP    DX
                                                POP    BX
                                                POP    AX
MOV     AH, 0fh                                POP    BP
INT     10h
; read unsigned args 2 and 3
MOV     DL, [BP+10]
MOV     DH, [BP+8]
                                                RETF
```

## *Putting the two together*

- The C module must be compiled
- The assembly language module assembled
- The pair must be linked together into an executable
- **Extern** in C is exactly the same as **Extern** in assembly programs
- Notice that the procedure is named `_placeStr`, because C compilers preface all external variables with an underscore

## *Complete calling procedure*

Program writes function parameters to stack (C is right-pusher)

CALL saves program's return address on the stack [PUSH CS (Far Proc);  
PUSH IP]

Routine marks stack frame (PUSH BP; MOV BP, SP)

Routine allocates stack memory for local variables (SUB SP, n)

Routine saves registers it modifies (push SI, push BX, push CX)

Subroutine Code

Additional CALLs, PUSHs, POPs

Routine restores registers it modifies (pop CX, pop BX, pop SI)

Routine deallocates stack memory for local variables (ADD SP, n)

Routine restores original value of BP (POP BP)

Subroutine Returns (RETF)

Program clears parameters from stack (ADD SP,p)

# Recursion

- Recursion: procedure calls itself

```
RecursiveProc
    DEC     AX
    JZ     .QuitRecursion
    CALL   RecursiveProc
.QuitRecursion:
    RET
```

- Requires a termination condition in order to stop infinite recursion
- Many recursively implemented algorithms are more efficient than their iterative counterparts

# Recursion example

```
Factorial
; Input  AX = CX = Value
; Output AX = Value !

DEC  CX
;Test for base case
CMP  CX,0
JE   .FactDone
IMUL CX
; Recurs
Call Factorial

.FactDone:
RET
```

Stack contents  
Assume:  
AX = CX = 4

Return IP Iteration 1  
CX = 4; AX = 4

Return Iteration IP 2  
CX = 3; AX = 12

Return Iteration IP 3  
CX = 2; AX = 24

Return Iteration IP 4  
CX = 1; AX = 24

# *Recursion*

---

- Recursion must maintain separate copies of all pertinent information (parameter value, return address, local variables) for each active call
- Recursive routines can consume a considerable amount of stack space
- Remember to allocate sufficient memory in your stack segment when using recursion
- In general you will not know the depth to which recursion will take you
  - allocate a large block of memory for the stack