

---

## Lecture 5: Subprograms

Dragan Mirkovic  
Department of Computer Science  
University of Houston

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Announcements

---

- Quiz #3 has been moved to 2/26/2004
  - Control structures (Ch. 5 in text)

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Introduction

---

- Subprograms are one of the most important structuring devices in your code
  - Simple interface for reusing the same piece of code in different places and with different data.
  - If well designed a subprogram should have a single purpose independent from the rest of the program.
  - Well structured programs are easier to write, maintain and debug.
  - We have used subprograms in the util library already.
- Interrupt services are special type of subprograms with a specific interface
  - Interrupt Service Routines (ISRs) in BIOS and DOS provide the lowest layer of software in MS-DOS operating system.

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Subprogram structure

---

- In the assembly language, the basic structure of a subprogram is very similar to the main program structure

```
SubProg  PROC
          ...
          subprogram code
          ...
          ret
SubProg  ENDP
```

- This subprogram is invoked by  
`Call SubProg`
- The `ret` instruction returns the control to the statement immediately following the the `call` instruction

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Writing subprograms

---

- Assembly language subprograms (PROCs) can be in the same source file as the main program or they can be stored in different files.
- The files just define the organization of the translation units.
- The ordering and naming of PROCs in the same file is immaterial
  - It does influence the scope in the usual way

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Writing subprograms...

---

- Subprograms don't need the code initializing the segments

```
.STACK 100h
.DATA
. . .
.CODE
Main PROC
    _Begin
    . . .
    call SubProg
    . . .
    _Exit
Main ENDP
SubProg PROC
    . . .
    ret
SubProg ENDP
END Main
```

No Initialization

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Writing subprograms...

---

- For readability, data which is used only by the subprogram should be physically associated with it.

```
    Main      ENDP
            .DATA
            . . .
            SubProg data
            . . .
            .CODE
    SubProg  PROC
            . . .
```

- In MASM (v. 6.0 and up) the labels are local to the PROC unless 'published' with a PUBLIC statement.
- In some other assemblers labels are visible to all procedures in a single source file (TASM)

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Example

---

- Interrupts are procedures which, in general, change the state of the registers
- A call to DOS INT 21h displays a string on the console and it uses AX and DX registers
- One can save the register values by using stack

```
.data
message db    'Hello!', 13,10,'$'
.code
push    ax
push    dx
mov     ah,9
mov     dx,offset message
int     21h
pop     dx
pop     ax
```

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# The STACK

---

- The stack is a LIFO structure directly related to subprograms
- When you allocate the stack, the stack pointer points to its top (highest address)
- Two main operations
  - `push mem/reg/const`
  - `pop mem/reg`
- The operand size must always be 16 or 32 bits
  - `push` will store a word or a dword at `ss:sp` and decrease the value of the `sp` by 2 or 4.
  - `pop` will copy a word or a dword from the memory position `ss:sp` and increase the value of `sp` by 2 or 4.

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## PUSH and POP Examples

---

- Only on 80286 and up can you push an immediate value.
- Examples:
  - `push ax` ; save a 16-bit register
  - `push ecx` ; save a 32-bit register
  - `push memval` ; save a memory operand
  - `push 1000h` ; save immediate value
- Examples:
  - `pop ax` ; restore a 16-bit register
  - `pop ecx` ; restore a 32-bit register
  - `pop memval` ; save a memory operand

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Example

```
-r
AX=1234 BX=5678 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000

        PUSH    AX ; SP=FFEC
-d ffe0
OB1A:FFE0 44 44 34 12 00 00 07 01-1A 0B 7E 05 34 12 00 00 DD4.....~.4...

        PUSH    BX ; SP=FFEA
-d ffe0
OB1A:FFE0 34 12 00 00 08 01 1A 0B-7E 05 78 56 34 12 00 00 4.....~.xV4...

        POP     BX
-d ffe0
OB1A:FFE0 34 12 00 00 08 01 1A 0B-7E 05 7E 05 34 12 00 00 4.....~.xV4...

        POP     AX
-d ffe0
OB1A:FFE0 34 12 34 12 34 12 00 00-0A 01 1A 0B 7E 05 00 00 4.4.4.....~....
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Comments

- The **push ax** instruction does not change the content of the register
- Stack make an excellent temporary save area for registers
- When a subroutine is called, the CPU saves a *return address* on the stack
- One can also push the subroutine arguments on the stack
- High-level languages create an area on the stack inside subroutines called *stack frame* for storing the local variables

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## The other stack related instructions

---

- The subroutine may change the state of the flags register.
- The `PUSHF` instruction pushes the Flags register on the stack.
- The `POPF` instruction is used to restore the flags original state.
- Starting with the 80286 processor there are two more stack related instructions
  - `PUSHA`: pushes AX, CX, DX, BX, SP, BP, SI, DI.
  - `POPA`: pops the same registers in reverse order.
  - `PUSHAD/POPAD`: the same for `EAX, EBX, ...`

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Using STACK

---

- The main use of the stack is to preserve a set of registers which define the state of the processor
- Subprograms should save and restore any registers they use
- The standard way to do that is to use the stack
- Subroutines should pop all (and only) the items they push on the stack
- Example

```
SubProg      PROC
              push ax
              push bx
              push cx
              . . .
              pop  cx
              pop  bx
              pop  ax
              ret
SubProg      ENDP
```
- **Note:** pop is in **REVERSE** order
- Example:

```
push ax
push bx
pop  ax
pop  bx
```

  - What happens here?

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# The stack, call and ret

---

- The stack is used to save the **IP** during the execution of the subprogram.
- The effect of the **call SubP** instruction:
  - Increase the value of the **IP**
  - Push the **IP** register (saves the return address on the stack).
  - Jump to **SubP**
- The effect of the **ret** instruction is to **pop** the **IP** from the stack.
  - The next instruction executed is the one following the **call**.

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Nested subprograms

---

- A subprogram may itself call other subroutines (including itself)
- The stack holds the list of return addresses so the CPU can find its way back to the original call.
- The **ret** at the end of the procedure pops the IP and execution resumes with the instruction following the call

```
main      .CODE
          PROC
          _Begin
          mov ax, A
          call subA
          _Exit 0
main      ENDP
subA      PROC
          call subB;
          ret
subA      ENDP
subB      PROC
          call subC;
          ret
subB      ENDP
subC      PROC
          ret
subC      ENDP
          END      main
```

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Debug trace

---

- Here is the debug trace of the nested subprogram example
- Only SP and IP registers are listed

```
SP=0110 IP=0013 : 0B91:0013 E80600      CALL    001C ; call to subA
SP=010E IP=001C : 0B91:001C E80100      CALL    0020 ; call to subB
SP=010C IP=0020 : 0B91:0020 E80100      CALL    0024 ; call to subC
SP=010A IP=0024 : 0B91:0024 C3          RET     ; return from subC
-d 10a
0B93:0100                                23 00 1F 00 16 00
SP=010C IP=0023 : 0B91:0023 C3          RET     ; return from subB
SP=010E IP=001F : 0B91:001F C3          RET     ; return from subA
SP=0110 IP=0016 : 0B91:0016 B000        MOV     AL,00
```

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Summary

---

- Subprograms are one of the most important structuring devices in your code
- The basic structure of subprograms resembles the main program structure
  - `call` and `return` instructions
- Translational units
  - Same file v.s. separate files
  - `EXTRN` and `PUBLIC` statements
- Use of stack for during the subroutine call
  - `push` and `pop` instructions
  - `call/ret` instructions

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Subprogram structure

- In the assembly language, the basic structure of a subprogram is very similar to the main program structure

```
SubProg  PROC
...
        subprogram code
...
        ret
SubProg  ENDP
```

- This subprogram is invoked by  
`Call SubProg`
- The `ret` instruction returns the control to the statement immediately following the the `call` instruction

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Separately translated subprograms

- If a subprogram is of general use it may be advantageous to save in a separate file.
- This requires a few extra statements.
- Example:

```
; ;MAIN.ASM
INCLUDE PCMAC.INC
.MODEL SMALL
.586
.STACK 100h
.DATA
; data for the main prog
.CODE
EXTRN SubProg : NEAR
main PROC
.Begin
.call SubProg
.Exit 0
main ENDP
END main
```

```
; ;SUB.ASM
INCLUDE PCMAC.INC; if necessary
.MODEL SMALL
.586
.STACK 100h
.DATA
; data for SubProg
.CODE
PUBLIC SubProg
SubProg PROC
; _Begin not needed
ret
SubProg ENDP
END ; no label!!!
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Separately translated subprograms

---

- The two programs must be assembled separately

```
masm main;
masm sub;
```

  - If macros are used in SubProg, they need to be included in it.
  - Normally, no stack is allocated in the sub. Any stack space specified is added to that in the main program
- Next, they need to be linked together

```
link main + sub,,,util.lib; (if needed)
```
- In some cases it is useful to store a collection of object files in a library  
lib command is used in MASM. (See text)

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Symbol tables

---

- Assembler creates the symbol table
  - Record programmer-defined symbols and information about them (location, storage, etc.)
  - This information is used inside the translational unit and it is discarded at the end of the translation
  - The **EXTRN** declaration causes the assembler to save each location where the external symbol is used so that the linker can later fill in the location
  - **.DATA** used by the two programs must be separated into
    - Data used by the main program only, which goes into MAIN.ASM
    - Data used by the subprogram only, which goes into SUB.ASM
    - Common data should use **EXTRN/PUBLIC** statements

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Global variables and data

---

- If there is data that is used both by the main program and the subprogram it needs to be specified through **EXTRN/PUBLIC** statements
  - **PUBLIC** statement creates a global variable
  - Any program can use it as long as it has the correct **EXTRN** statement
  - Note: global variables should be used sparingly
- Example

```
;MAIN.ASM
...
.DATA
; data for the main prog
PUBLIC COMN
COMN DW ?
.CODE
EXTRN SubProg : NEAR
main PROC
...
```

```
;SUB.ASM
...
.DATA
; data for SubProg
EXTRN COMN:WORD
.CODE
PUBLIC SubProg
SubProg PROC
...
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Creating the program libraries

---

- In many cases it is convenient to organize the subroutines into libraries
  - There should be a logical connection between the procedures
  - Simplifies the overall maintenance of the code
- The general command for manipulating a program library is

```
MASM: lib lib-file-name commands ;
lib-file-name - name without the extension
commands - list of .obj file names with or without the
extension prefixed by characters indicating the operations to
be performed
(lib in MASM and C++ are not compatible)
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Program libraries ...

---

Prefixes	Actions
+	Add file to the library
-	Delete file from the library
-+	Replace file in library

- Examples:
  - create a new library `NLIB.LIB` containing files `A.OBJ`, `B.OBJ`, `C.OBJ`  
`lib nlib +a +b +c`
  - Delete `B.OBJ` from `NLIB.LIB`  
`lib nlib -b`
  - Replace `A.OBJ` and `C.OBJ` in `NLIB.LIB` with new versions  
`lib nlib -+a -+c`

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## The Linking Process

---

- Assembler creates the symbol table
  - Record programmer-defined symbols and information about them (location, storage, etc.)
  - This information is used inside the translational unit and it is discarded at the end of the translation
  - The **PUBLIC** and **EXTRN** declarations require special treatment
  - Assembler saves two tables into the `.OBJ` file
    1. Table of **EXTRN** symbols with their reference points
    2. Table of **PUBLIC** symbols and its unique place of definition
- The linking process of attaching calls to their destinations is called **resolving external references**

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# The Linking Process...

---

- Example:

```
link P1+P2+...+Pn , , L1+L2+...+Lm;
```

- Links  $P1+P2+...+Pn$  and uses  $L1+L2+...+Lm$  to resolve any further external references
- It performs the following steps:
  1. Read each  $P_i$ .OBJ and merge it into three objects:
    - A file of object code oc. (If oc contained n bytes of code before,  $P_i$ 's code will start at offset n)
    - A table of external references  $XSR$ . For each **EXTRN** symbol in  $P_i$  add its references to the end of  $XSR$  table and add n to each of the reference locations.
    - Table of public symbol definitions  $PSD$ . For each **PUBLIC** symbol, check if it is already in  $PSD$  and generate an error message if necessary, otherwise add it to the end of  $PSD$  and add n to its location.

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Resolving external references

---

2. For each symbol in  $XSR$ , look it up in  $PSD$ . If it is there resolve all references for that symbol and delete it from  $XSR$ .
3. If the  $XSR$  is now empty, all references are resolved and we are done!
4. If not for each library  $L_i$ .LIB, read through each .OBJ file and see if it has any **PUBLIC** symbols referenced in  $XSR$ . If so, read in that .OBJ file as in step 1. Above.
5. If no new .OBJ files were added in 4., then there are unresolved external references and **linking fails!** Otherwise go back to step 2.

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Example

---

- Consider the following code:

```
;;
    ...
    .STACK 100h
    .DATA
    ...
    .CODE
Tst  EXTERN  PutDec : NEAR
    PROC
    _Begin
    ...
    call    PutDec
    ...
    call    PutDec
    _Exit  0
Tst  ENDP
    END    Tst
```

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Output

---

```
C:\Asm>masm jtst
Microsoft (R) MASM Compatibility Driver
...
Assembling: jtst.asm

C:\Asm>link jtst;
...
jtst.obj(jtst.asm) : error L2029: 'PUTDEC' : unresolved external

There was 1 error detected
C:\Asm>
```

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# External symbols

- Object file contains three types of records describing the **PUBLIC** and **EXTRN** names:
  - EXTDEF** External Names Definition Record
  - TYPDEF** Type Definition Record
  - PUBDEF** Public Names Definition Record

```
C:\Src\Aasm>debug jtst.obj
-d
0B1D:0100  80 0A 00 08 6A 74 73 74-2E 61 73 6D 3A 96 2B 00  ....jtst.asm:+.
...
0B1D:0160  03 59 8C 09 00 06 50 55-54 44 45 43 00 A0 A0 15  .Y....PUTDEC...
0B1D:0170  00 02 00 00 05 00 04 00-41 20 3D 20 24 42 20 3D  ....A = $B =
```

- EXTDEF—External Names Definition Record Format

1	2	1	<String Length>	1 or 2	1
8C	Record Length	String Length	External Name String	Type Index	Checksum

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Object file records

- PUBDEF**—Public Names Definition Record Format

1	2	1 or 2	1 or 2	2	1	<String Length>	2 or 4	1 or 2	1
90 or 91	Record Length	Base Group Index	Base Segment Index	Base Frame	String Length	Public Name String	Public Offset	Type Index	Checksum
				<conditional>					<repeated>

- The PUBDEF record contains a list of public names.
- It makes items defined in this object module available to satisfy external references in other modules with which it is bound or linked.
- See **OMF: Relocatable Object Module Format Specification** for more details.

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Procedures and High-Level Languages

---

- Up to now we have used only very simple procedures
  - Little space required for parameters and local variables
  - No recursion and reentrancy
- Here we discuss most general requirements for procedures and their use of the stack
- High-level languages require more functionality from the stack

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Procedures and the Stack

---

- Functionality required in high-level languages:
  - Passing of the parameters
    - Registers may not be enough
  - Storage for the local variables
  - Each recursive call requires its own copy of the local variable set.
  - Many procedures need to be reentrant (usable by several different programs simultaneously)
- Solution: the **stack frame**

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# The stack frame

---

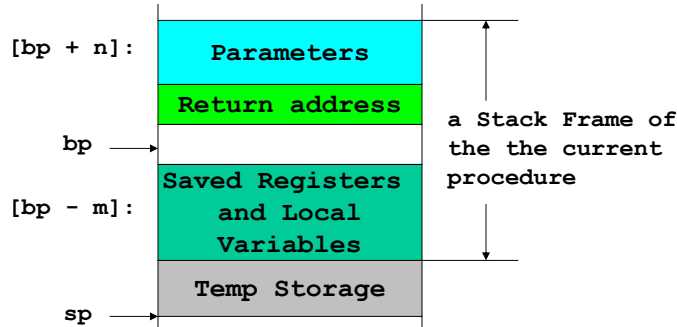
- A fixed-size block of memory on the stack for
  - Parameters, local variables, register storage, etc.
- When the procedure is called, its stack frame is pushed onto the stack
- The routine itself may then push and pop temp storage onto the stack and the routines it calls can do the same
- When the procedure finally exits, its stack frame is popped off the stack

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# The stack frame...

---



- The base pointer **bp** register points to a fixed position in the middle of the stack
- The stack pointer **sp** points to the top of the stack

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# The bp register

---

- Various parts of the stack frame can be referenced relative to `bp` register
- The caller creates the 1<sup>st</sup> part of the stack frame by pushing the parameters and the return address onto the stack.
- The called procedure creates the 2<sup>nd</sup> part
  - It saves and restores the `bp` register first
  - Next, it saves its local parameters and registers used inside the procedure

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Setting up the stack

---

- Saving and resetting `bp` in the **called procedure**:

```
SubProg  PROC
        push  bp
        mov   bp, sp
        ...
```

  - `bp` contains the value of the stack pointer before the local variables and registers are pushed onto the stack.
- Allocate space for the local parameters:

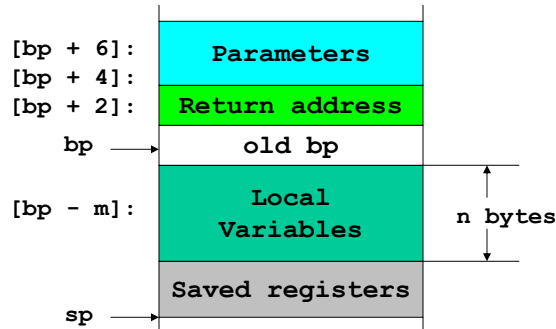
```
        ...
        sub   sp, n; Reserve n bytes of local
            ; storage
        push  reg1
        push  reg2
        ...
```

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Stack Structure in Sub

- The result is the following stack structure:



- This is assuming the small model (call pushes only the IP)
- In medium and large models far call pushes CS and IP

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Exit procedure

- Assuming that all temporary storage has been released (popped of the stack) sp points to the last saved register

```
...
pop  reg2
pop  reg1
mov  sp, bp;
pop  bp
```

- This code should always be used exactly in order to avoid errors
  - Some high-level languages need some additional structures

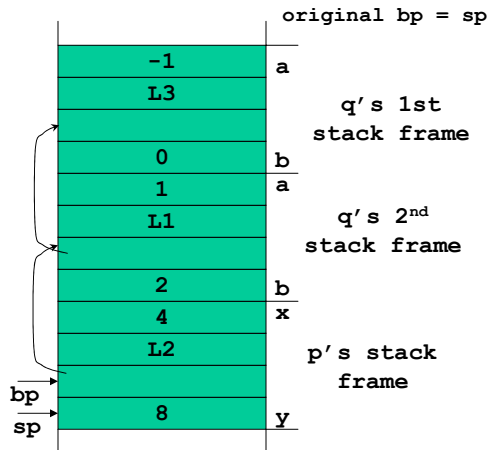
D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Example 13.1-1

- Consider the following pseudo-program:

```

procedure p(x)
  word    y
  y = 2 * x
  // stack snapshot
end p
procedure q(a)
  word    b
  b = a + 1
  if (a<0) then q(b+1)// L1:
             else p(b+2)// L2:
end q
// main program
q(-1)    // L3:
  
```



D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Factorial function

- The factorial algorithm calculates
 
$$n! = 1 * 2 * \dots * n,$$
 where  $n$  is an unsigned integer
- One can implement it in C using the following recursion
 

```

unsigned int Factorial(unsigned int n) {
  unsigned int f;
  if(n==0) f = 1; else f = n*Factorial(n-1);
  return f;
}
      
```
- Note:
  - This may not be the most efficient way of implementing factorials
  - The function value grows very rapidly!

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Factorial code

- Here is the subroutine code:

```

INCLUDE PCMAC.INC
.MODEL SMALL
.586
.STACK 100h
.CODE
PUBLIC Fact
Fact PROC
push bp
mov bp, sp
mov ax, [bp+4]
cmp ax, 0
ja Ll
mov ax, 1
jmp Lend
Ll: dec ax
push ax
call Fact
mov bx, [bp+4]
mul bx
Lend: mov sp, bp
pop bp
Fact ENDP
END
    
```

- Thee main program:

```

;; Factorial main program
;;
INCLUDE PCMAC.INC
.MODEL SMALL
.586
.STACK 100h
.DATA
n DW 6
CrLf DB 13,10,"$"
.CODE
EXTRN Fact : NEAR, PutDec : NEAR
FactM PROC
_begin
Ll: push n
call Fact
call PutDec
_putStr CrLf
dec n
jnz Ll
_exit 0
FactM ENDP
END FactM
    
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Output

- Here is the output of the program:
- And here is the snapshot of the stack segment at the last recursive call to the 'Fact' procedure:

C:\Src\Asm>factm

```

720
120
24
6
2
1
    
```

Last (partial) Stack frame

```

AX=0000 BX=0020 CX=00DB DX=0000 SP=01F8 BP=01FC SI=0000 DI=0000
DS=0B9C ES=0B80 SS=0B9C CS=0B90 IP=002E NV UP EI PL ZR NA PE NC
0B90:002E 55          PUSH  BP
-d 1f0
0B9C:01F0 FC 01 2E 00 90 0B 81 05 43 00 00 00 02 02 43 00 .....C.....C.
0B9C:0200 01 00 08 02 43 00 02 00 0E 02 43 00 03 00 14 02 .....C.....C.....
0B9C:0210 43 00 04 00 1A 02 43 00 05 00 00 00 17 00 06 00 C.....C.....
    
```

The diagram shows a stack segment with several frames. The top frame is labeled 'Last (partial) Stack frame'. Below it are other frames. Labels with arrows point to specific memory locations: 'Stack frames' points to the top of the stack; 'BP' points to the base pointer register value (01FC); 'Return IP in Fact' points to the instruction pointer of the fact procedure (002E); 'n-1' points to the value 05; 'Return IP in main' points to the instruction pointer of the main program (002E); 'BP' points to the base pointer register value (01FC); 'Initial n' points to the value 06; and another 'Return IP in main' points to the instruction pointer of the main program (002E).

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Parameters and the stack

---

- In what order are parameters pushed?

`p(a, b, c)` generates

```
push a           push c
push b   or     push b
push c           push a
```

- Who should pop the parameters off the stack, the calling or the called routine?
- Different set of choices depending on the language environment

---

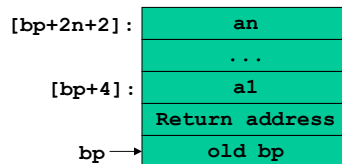
D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## C and STDCALL conventions

---

- C/C++ parameter conventions:
  - Some C and C++ functions (`printf`) can be passed a different number of parameters (`varargs`).
  - Parameters are pushed in the reverse order (why?)
  - The caller knows the number of parameters and pops the parameters using something like  
`add sp, n; pop n bytes  
from the stack`

- `call p(a1, ..., an)` results in



- STDCALL parameter conventions:
  - Reverse push
  - Called proc does the popping

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Parameters Example

---

- Inside of the the procedure one can define the parameters and local variables using the `EQU` declaration
- Remember: `EQU` is essentially an alias
- Example: Let the procedure `p` have three arguments and 2 local variables: `p(a, b, c)` and `d, e` are the local variables
- Then
  - a `EQU WORD PTR [BP + 4]`
  - b `EQU WORD PTR [BP + 6]`
  - c `EQU WORD PTR [BP + 8]`
  - d `EQU WORD PTR [BP - 2]`
  - e `EQU WORD PTR [BP - 4]`
  - Simplifies the writing of the code

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Assembler help with procedures

---

- MASM offers a lot of help in communicating with high-level languages
- Additional parameters for `.MODEL` directive
  - `.MODEL size, C; or STDCALL`
  - Affects all of the `PROC`s in the file
- Additional parameters for `PROC` statement
  - `Sub PROC C; or STDCALL`
- Consequences:
  - The name is automatically made `PUBLIC`
  - If `C` is specified: all symbols have a leading `'_'` added.
  - Code to save and restore the `bp` is added automatically.

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Assembler help with procedures...

---

- One can use the `USES` declaration to specify the registers used locally and the procedure parameters
  - Registers are saved and restored (`push/pop`),
  - Parameter names are equated to `[bp+n]` taking into account the memory model.
  - If the language type is `STDCALL` the `ret` instruction is automatically changed to `ret m`, where `m` is the number of bytes of parameters specified in `PROC` statement

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## USES Example

---

- The declaration

```
SubP PROC C USES SI DI, A : WORD, B : DWORD, C : WORD
    ...
    ret
SubP ENDP
```

**NOTE:** 1<sup>st</sup> comma after all registers are listed
- Is equivalent to the code

```
PUBLIC _SubP
_SubP PROC
    push bp
    mov bp, sp
    push si
    push di
    A EQU WORD PTR [bp + 4]
    B EQU DWORD PTR [bp + 6]
    C EQU WORD PTR [bp + 10]
    ...
```

```
...
    pop di
    pop si
    mov sp, bp
    pop bp
    ret
_SubP ENDP
```

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## Declaring Procedure Prototypes

---

- Similar to C prototype declaration
- **PROTO** directive (pseudo-op) similar to **PROC** without **USES**  
`SubP PROTO C A : WORD, B : DWORD, C : WORD`
  - It generates an appropriate **EXTRN** statement:  
`EXTRN _SubP`
- The pseudo-op **INVOKE** can be used to call the procedure  
`INVOKE SubP, p1, p2, p3`
- This is equivalent to:  
`push p3  
push p2  
push p1  
call _SubP  
add sp, 8`
- The parameters: constants, variables, registers

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

## ENTER and LEAVE

---

- 80x86 CPUs from 80186 up include two additional instructions:  
**ENTER** and **LEAVE**
- The instruction  
`enter n, 0 ; Enter the subprocedure`
- Is equivalent to:  
`push bp  
mov bp, sp  
sub sp, n; Allocate n bytes of local storage`
- The instruction  
`leave ;`
- Is equivalent to:  
`mov sp, bp  
pop bp`
- The **ret** instruction is still required.

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Passing pointers

---

- Example:

- C procedure to swap two values.

```
void Swap(int *xp, int *yp) {
    int    tmp;
    tmp = *xp;
    *xp = *yp;
    *yp = tmp;
}
```

- The parameters should be passed by reference.

- The call in C is:

```
Swap(&a, &b),
where a and b are int variables
```

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Assembly code

---

```
xp    EQU    WORD PTR [bp+4]; C parameters pushed in rev.
yp    EQU    WORD PTR [bp+6];
Swap  PUBLIC Swap
      PROC
      push  bp
      mov   bp, sp
      push  ax
      push  si
      push  di;          save registers
      mov   si, xp;      si contains OFFSET of x
      mov   di, yp;      di contains OFFSET of y
      mov   ax, [si];    ax = x
      xchg  ax, [di];    ax <-> *y
      mov   [si], ax;    *x = *y
      pop   di
      pop   si
      pop   ax
      pop   bp
Swap  ENDP
```

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# The main program

---

- The call in C `Swap (&a, &b)` is equivalent to:  

```
push  OFFSET b
push  OFFSET a
EXTRN  Swap : NEAR
call  Swap
add   sp, 4
```
- One can use 'load address' instruction `lea` to simplify the call  

```
lea reg, memory reference
```
- The instruction:  

```
lea reg, Var
```
- Is equivalent to:  

```
mov reg, OFFSET Var
```

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004

# Homework

---

- Read Ch. 6 and Ch. 13.1-13.4 in Jones
- Problems: 1, 2 pp. 154 and 2, 3 pp. 402.

---

D. Mirkovic, COSC 2410: Computer Organization and Programming, Spring 2004